PyXRD Documentation

Mathijs Dumon

Jun 14, 2018

Contents

PyXRD is a python implementation of the matrix algorithm for computer modeling of X-ray diffraction (XRD) patterns of disordered lamellar structures. It’s goals are to:

1. provide an easy user-interface for end-users
2. provide basic tools for displaying and manipulating XRD patterns
3. produce high-quality (publication-grade) figures
4. make modelling of XRD patterns for mixed-layer clay minerals ‘easy’
5. be free and open-source

CHAPTER 1

Motivation

PyXRD was written with the multi-specimen full-profile fitting method in mind. The direct result of this is the ability to 'share' parameters among similar phases.

This allows for instance to have an air-dry and a glycolated illite-smectite share their coherent scattering domain size, but still have different basal spacings and interlayer compositions for the smectite component.

Other features are (incomplete list):

- Import/export several common XRD formats (.RD, .RAW, .CPI, ASCII)
- simple background subtraction/addition (linear or custom patterns)
- smoothing patterns and adding noise to patterns
- peak finding and annotating (markers)
- custom line colors, line widths, pattern positions, . . .
- goniometer settings (wavelengths, geometry settings, . . .)
- specimen settings (sample length, absorption, . . .)
- **automatic parameter refinement using several algorithms, e.g.:**
    - L BFGS B
    - Brute Force
    - Covariation Matrix Adapation Evolutionary Strategy (CMA-ES; using DEAP)
    - Multiple Particle Swarm Optimization (MPSO; using DEAP)
    - scripting support

CHAPTER 2

Contents

2.1 Library API Reference

A python implementation of the matrix algorithm developed for the X-ray diffraction analysis of disordered lamellar structures

2.1.1 Atoms module

AtomType

Atom

2.1.2 Probabilities module

The probabilities module contains a classes that allow the calculation of weigth and probability matrixes for mixed-layer minerals.

Theory

Mixed-layer probabilities

These probability classes use the Reichweite (= R) concept and Markovian statistics to calculate how the layer stacking sequence is ordered (or disordered).

The value for R denotes what number of previous layers (in a stack of layers) still influence the type of the following component. With other words, for:

- R=0; the type of the next component does not depend on the previous components,
- R=1; the type of the next component depends on the type of the previous component,
- R=2; the type of the next component depends on the type of the previous 2 components,

- ...

We can describe the stacking sequence using two types of statistics: weight fractions and probabilities. Some examples:

- the fraction of A type layers would be called W_A
- the probability of finding an A type layer in a stack would be called P_A
- the fraction of A type layers immediately followed by a B type layer would be called W_{AB}
- the probability of finding an A type layer immediately followed by a B type layer would be called P_{AB}

There exist a number of general relations between the weight fractions W and probabilities P which are detailed below. They are valid regardless of the value for R or the number of components G. Some of them are detailed below. For a more complete explanation: see Drits & Tchoubar (1990). For stacks composed of G types of layers, we can write (with N the number of layers):

$$W_i = \frac{N_i}{N_{max}} \qquad \forall i \in [1, 2, \ldots, G]$$
$$W_{ij} = \frac{N_{ij}}{N_{max} - 1} \qquad \forall i, j \in [1, 2, \ldots, G]$$
$$W_{ijk} = \frac{N_{ijk}}{N_{max} - 2} \qquad \forall i, j, k \in [1, 2, \ldots, G]$$

etc.

$$W_{ij} = W_i \cdot P_{ij}$$
$$W_{ijk} = W_{ij} \cdot P_{ijk}$$

etc.

$$\sum_{i=1}^{G} W_i = 1$$
$$\sum_{i=1}^{G}\sum_{j=1}^{G} W_{ij} = 1$$

etc.

$$\sum_{j=1}^{G} P_{ij} = 1$$
$$\sum_{k=1}^{G} P_{ijk} = 1$$

etc.

Because of these relationships it is not neccesary to always give all of the possible weight fractions and probability combinations. Each class contains a description of the number of 'independent' variables required for a certain combination of R ang G. It also details which ones were chosen and how the others are calculated from them.

More often than not, ratios of several weight fractions are used, as they make the calculations somehwat easier. On the other hand, the actual meaning of these fractions is a little harder to grasp at first.

Class functionality

The classes all inherit from an 'abstract' base class which provides a number of common functions. One of the 'handy' features are its indexable properties *mW* and *mP*. These allow you to quickly get or set an element in one of the matrixes:

```
>>> from pyxrd.probabilities.models import R1G3Model
>>> prob = R1G3Model()
>>> prob.mW[0] = 0.75 # set W1
>>> print prob.mW[0]
0.75
>>> prob.mW[0,1] = 0.5 # set W12
>>> print prob.mW[0,1]
0.5
```

Note however, that doing so might produce invalid matrices and produce strange X-ray diffraction patterns (or none at all). It is therefore recommended to use the attributes of the selected 'independent' parameters (see previous section) as setting these will trigger a complete re-calculation of the matrices.

If however, you do want to create a matrix manually, you can do so by setting all the highest-level elements, which are:

- for an R0 class only the Wi values
- for an R1 class the Wi and Pij values

- for an R2 class the Wij and Pijk values
- for an R3 class the Wijk and Pijkl values

After this you can call the *solve* and *validate* methods, which will calculate the other values (e.g. for an R2 it will calculate Wi, Wijk and Pij values).

An example:

```
>>> from pyxrd.probabilities.models import R1G2Model
>>> prob = R1G2Model()
>>> prob.mW[0] = 0.75 # set W1
>>> prob.mW[1] = 0.25 # set W2 (needs to be 1 - W1 !)
>>> prob.mP[1,1] = 0.3 # set P22
>>> prob.mP[1,0] = 0.7 # set P21 (needs to be 1 - P22 !)
>>> prob.mP[0,1] = 0.7 / 3.0 # set P12 (needs to be P21 * W2 / W1!)
>>> prob.mP[0,0] = 2.3 / 3.0 # set P11 (needs to be 1 - P12 !)
>>> prob.solve()
>>> prob.validate()
>>> print prob.get_distribution_matrix()
[[ 0.75  0.  ]
 [ 0.    0.25]]
>>> print prob.get_probability_matrix()
[[ 0.76666667  0.23333333]
 [ 0.7         0.3       ]]
```

Note that at the end we print the validation matrixes to be sure that we did a good job: if all is valid, we should see only "True" values. For more details on what elements produced an invalid results, you can look at the W_valid_mask and P_valid_mask properties.

The exact same result could have been achieved using the independent parameter properties:

```
>>> from pyxrd.probabilities.models import R1G2Model
>>> prob = R1G2Model()
>>> prob.W1 = 0.75
>>> prob.P11_or_P22 = 0.3
>>> print prob.get_distribution_matrix()
[[ 0.75  0.  ]
 [ 0.    0.25]]
>>> print prob.get_probability_matrix()
[[ 0.76666667  0.23333333]
 [ 0.7         0.3       ]]
```

For more information see the `_AbstractProbability` class

Models

Base Models

R0 Models

R0 models have $G - 1$ independent parameters, G being the number of components.

Partial weight fractions were chosen as independent parameters, as this approach scales very well to a large number of components:

If we define a partial weight fraction as $F_i = \frac{W_i}{\sum_{j=i}^{G} W_j} \forall i \in [1, G]$, and keep in mind the general rule $\sum_{i=1}^{G} W_i = 1$, we can calculate all the weight fractions from these partial weight fractions progressively, since:

- F_1 will acutally be equal to W_1.
- the denominator of every fraction F_i is equal to $1 - \sum_{j=1}^{i-1} W_j$, and you are able to calculate this:
  - for F_2, it would be $1 - W_1$, and you know W_1 from the first fracion
  - for F_3 it would be $1 - W_1 - W_2$, and you can get W_1 and W_2 from the previous two fractions.
- once the weight fractions of the first $G - 1$ components are known, then the weight fractions of the last component can be calculated as $W_g = 1 - \sum_{i=1}^{G} W_i$.

R1 Models

R2 Models

R3 Models

2.1.3 Phases module

The phases module contains a number of classes that allow to create complex mixed-layer phases.

TODO: add example code on how to use them!

Phase

CSDS

Unit-cell properties

Component

Atom Relations

2.1.4 Goniometer module

2.1.5 Mixture module

The mixture module contains a number of classes that manage 'mixtures'. Mixtures combine multiple specimens and phases with each other. Mixtures are part of the project, which also holds a reference to the phases and specimens (and possible others as well) in the mixture.

The combination of phases and specimens is achieved using a kind of combination 'matrix', in which rows are phases and columns are specimens. In other words, each column gets a specimen asigned to it, and each slot in the matrix gets a phase asigned to it. This way it is possible to have the same phase for different specimens of your sample if that phase is believed to be 'immune' to the treatments, or to have different (or at least partially different) phases when it is believed to be affected by the treatment in some way.

For an explanation on how to create and link phases see the documentation on *Phases module*.

TODO: add example code on how to use mixtures, optimizers and refiners

Mixture

Optimizer

Refiner

RefineContext

2.1.6 Project module

Project

2.1.7 Calculations

This module contains the basic implementation of the matrix formalism as detailed in Drits and Tchoubar (1990) and Plançon (2001).

It was chosen to implement this using 'loose' function calls. The disadvantage of this approach is that the functions are no longer bound to class instances, which makes them less intuitive to use. The advantage is we can more easily call these functions asynchronously (e.g. using `Pool`)

Despite all this, most function calls in this module do expect to be passed a `DataObject` sub-class, which wraps all the data in a single object. These `DataObject` s map onto the different models used. As such this module is also largely independent from the MVC framework used.

Drits, V.A., and Tchoubar, C., 1990. X-Ray Diffraction by Disordered Lamellar Structures: Theory and Applications to Microdivided Silicates and Carbons. Springer-Verlag, Berlin, Germany. Plançon, A., 2001. Order-disorder in clay mineral structures. Clay Miner 36, 1–14.

Atoms

Components

Phases and CSDS

Goniometer

Specimen

Statistics

Improve

Exceptions

exception `pyxrd.calculations.exceptions.`**`WrapException`**

A wrapped exception used by the *`wrap_exceptions()`* decorator.

`pyxrd.calculations.exceptions.`**`wrap_exceptions`**(*func*)

Function decorator that allows to provide useable tracebacks when the function is called asynchronously and raises an error.

Data Objects

The following classes are not meant to be used directly, rather you should create the corresponding model instances and retrieve the DataObject from them.

The rationale behind not using the model instances directly is that they are difficult to serialize or pickle (memory-)efficiently. This is mainly due to all of the boiler-plate code that takes care of references, saving, loading, calculating properties from other properties etc. A lot of this is not needed for the actual calculation. The data objects below, on the other hand, only contain the data needed to be able to calculate XRD patterns.

class `pyxrd.calculations.data_objects.`**`DataObject`**`(`***kwargs*`)`

The base class for all DataObject instances.

The constructor takes any number of keyword arguments it will set as attributes on the instance.

class `pyxrd.calculations.data_objects.`**`AtomTypeData`**`(`***kwargs*`)`

The DataObject describing an AtomType.

`par_a = None`

a numpy array of *a* scattering factors

`par_b = None`

a numpy array of *b* scattering factors

`par_c = None`

the *c* scattering constant

`debye = None`

the debye-waller temperature factor

class `pyxrd.calculations.data_objects.`**`AtomData`**`(`***kwargs*`)`

The DataObject describing an Atom.

`atom_type = None`

an *`AtomTypeData`* instance

`pn = None`

the # of atoms projected to this z coordinate

`default_z = None`

the default z coordinate

`z = None`

the actual z coordinate

class `pyxrd.calculations.data_objects.`**`ComponentData`**`(`***kwargs*`)`

The DataObject describing an Atom

`layer_atoms = None`

a list of *`AtomData`* instances

`interlayer_atoms = None`

a list of *`AtomData`* instances

`volume = None`

the component volume

`weight = None`

the component weight

`d001 = None`

the d-spacing of the component

default_c = None
the default d-spacing of the component

delta_c = None
the variation in d-spacing of the component

lattice_d = None
the height of the silicate lattice (excluding the interlayer space)

class `pyxrd.calculations.data_objects.`**CSDSData**(***kwargs*)
The DataObject describing the CSDS distribution.

average = None
average CSDS

maximum = None
maximum CSDS

minimum = None
minimum CSDS

alpha_scale = None
the alpha scale factor for the log-normal distribution

alpha_offset = None
the alpha offset factor for the log-normal distribution

beta_scale = None
the beta scale factor for the log-normal distribution

beta_offset = None
the beta offset factor for the log-normal distribution

class `pyxrd.calculations.data_objects.`**GonioData**(***kwargs*)
The DataObject describing the Goniometer setup.

min_2theta = None
Lower 2-theta bound for calculated patterns

max_2theta = None
Upper 2-theta bound for calculated patterns

steps = None
The number of steps in between the lower and upper 2-theta bounds

soller1 = None
The first soller slit size

soller2 = None
The second soller slit size

divergence = None
The divergence size

has_ads = None
Whether and Automatic Divergence Slit correction should be performed

ads_fact = None
ADS Factor

ads_phase_fact = None
ADS phase factor

ads_phase_shift = None
ADS phase shift

ads_const = None
ADS constant

radius = None
The goniometer radius

wavelength = None
The goniometer wavelength

wavelength_distribution = None
The goniometer wavelength distribution

class pyxrd.calculations.data_objects.**ProbabilityData**(***kwargs*)
The DataObject describing the layer stacking probabilities

valid = None
Whether this probability is really a valid one

G = None
The number of components this probability describes

W = None
The weight fractions matrix

P = None
The probabilities matrix

class pyxrd.calculations.data_objects.**PhaseData**(***kwargs*)
The DataObject describing a phase

apply_lpf = True
A flag indicating whether to apply Lorentz-polarization factor or not

apply_correction = True
A flag indicating whether to apply machine corrections or not

components = None
A list of *ComponentData* instances

probability = None
A *ProbabilityData* instance

sigma_star = None
The sigma start value

csds = None
A *CSDSData* instance

class pyxrd.calculations.data_objects.**SpecimenData**(***kwargs*)
The DataObject describing a specimen

goniometer = None
A *GonioData* instance

sample_length = None
The sample length

absorption = None
The sample absorption

phases = None
A list of *PhaseData* instances

observed_intensity = None
A numpy array with the observed intensities

total_intensity = None
A numpy array with the calculated intensity

phase_intensities = None
A nummpy array with the calculated phase profiles

class pyxrd.calculations.data_objects.**MixtureData**(***kwargs*)
The DataObject describing a mixture

specimens = None
A list of *SpecimenData* instances

fractions = None
A numpy array with the phase fractions

bgshifts = None
A numpy array with the specimen background shifts

scales = None
A numpy array with the specimen absolute scales

parsed = False
Whether this MixtureData object has been parsed (internal flag)

n = 0
The number of specimens

m = 0
The number of phases

2.2 Script Tutorial

2.2.1 Introduction

It is possible to write scripts for PyXRD (projects). This allows anybody to make PyXRD do things it wasn't really intended to do or to automate certain tasks. Parts of the official PyXRD code are scripts themselves. This tutorial will provide an introduction on how to setup such a script.

We assume the interested reader has already made himself familiar with Python.

2.2.2 Hello World script

Fire up your favorite text editor and copy the following piece of code:

```
#!/usr/bin/python
# coding=UTF-8

import logging
logger = logging.getLogger(__name__)

def run(args):
```

(continues on next page)

(continued from previous page)

```
    """
        Run as
         python core.py -s path/to/hello_world.py
    """
    logging.info("Creating a new project")

    from pyxrd.project.models import Project
    project = Project(name="Hello World", description="This is a hello world project")

    from pyxrd.scripts.tools import reload_settings, launch_gui
    reload_settings()
    launch_gui(project)      # from this point onwards, the GUI takes over!
```

What this script does is very simple: it will create a new project, with it's name and title set to "Hello World" and "This is a hello world project" respectively. Then it will launch the gui as it would normally start but pass in this newly created project. What you should see is PyXRD loading as usual but with this new project pre-loaded.

2.2.3 Running the script

Save the script somewhere (e.g. on your desktop) and name it "hello_world.py".

To run this script you have to tell PyXRD where to find it first. So instead of starting PyXRD as you would usually do, open up a command line (Windows) or terminal (Linux), and follow the instructions below.

Windows

On windows the following command should start PyXRD with the script:

```
C:\Python27\Scripts\PyXRD.exe -s "C:\path\to\script\hello_world.py"
```

Replace the path\to\script part with the actual path where you saved the script. The above example also assumes you have installed python in C:\Python27 (the default).

Linux

On linux the following command should start PyXRD with the script:

```
PyXRD -s "/path/to/script/hello_world.py"'
```

Replace the /path/to/script/ part with the actual path where you saved the script. This assumes you have installed PyXRD using pip so that the PyXRD command is picked up by the terminal. If you get an error like 'PyXRD: command not found', you will need to find out where PyXRD was installed and use the full path instead.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index