PyXMRTool Documentation

Release 0.9

Yannic Utz

Jul 20, 2018

CONTENTS

CHAPTER

ONE

TODO: INTRODUCTION

to be written

CHAPTER

TWO

INSTALLATION

Dependencies: `PyXMRTool` depends on the package `Pythonreflectivity` from Martin Zwiebler. You need to install it first.

Download the PyXMRTool project as zip from https://github.com/malaclypseII/PyXMRTool and unpack it. Go to the folder and enter 'python setup.py install' to the command promt.

CHAPTER

THREE

TODO: USAGE

to be written

CHAPTER

FOUR

API

4.1 Module *Parameters*

Provides an easy and transparent handling of fit parameters.

Fitparameters can be handled as instances of *Fitparameter* which are collected and managed by an instance of *ParameterPool*. To allow for transparent modelling, *Fitparameter* is derived from *Parameter* which can hold a constant value. I.e. the code using a parameter does not have to know if it is a constant *Parameter* or a variable *Fitparameter*. The values of both are obtained in the same way with **getValue**.

Both parameter classes support also the creation of of dependent paramters by arithmetic operations:

```
>>> import Parameters
>>> const=Parameters.Parameter(30.7)
>>> const.getValue()
30.7
>>> pp=Parameters.ParameterPool()
>>> par_real=pp.newParameter('par_real',start_val=0,lower_lim=-10,upper_lim=10)
>>> par_complex=pp.newParameter('par_complex',start_val=0+0j,lower_lim=-10-1j,upper_
↪lim=100.3+20j)
>>> par_real.getValue([1,2,3])
1
>>> par_complex.getValue([1,2,3])
(2+3j)
>>> dep_par = ((par_complex + 5*par_real)*0.68)**const
>>> dep_par.getValue([1,2,3])
(8.367641368810413e+21-1.1467109965804664e+21j)
```

class Parameters.**Parameter**(*value=None*)

Bases: object

Base class for parameters. It contains a (complex) value which is set as instantiation and cannot be changed.

This class (and all derived classes) supports creation of dependent parameters by arithmetic operations.

__init__(*value=None*)

Initialize with a number as (complex) **value**.

getValue(*fitpararray=None*)

Returns the **value**.

fitpararray is not used and only there to make this function forward compatible with the *Fitparameter* class and with dependent parameters.

class Parameters.**Fitparameter**(*name*, *fixed=False*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)

Bases: *Parameters.Parameter*

Contains name, starting value and limits of a fitting parameter and knows how to get the parameter value out of an array of values of fitparameters if it is attached to an instance of *ParameterPool*.

__init__(*name*, *fixed=False*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)
Initialize a fitparameter at least with a **name**.

Usually, you (the user) should not instantiate a fitparameter directly. Instead use *ParameterPool.newParameter()*.

Parameters

- **name** (*str*) – Name of the fit parameter. Use names without whitespaces. Otherwise there can be problems when using a parameter file.
- **fixed** (*bool*) – If *True*,the parameter will not be varied during a fit routine and the limits are not necessary.
- **start_val** (*number*) – Start value of the fit parameter.
- **upper_lim** (*number*) – Lower limit of the fit parameter.
- **upper_lim** – Upper limit of the fit parameter.

fix()
Fix parameter during fitting.

unfix()
Set parameter as variable during fitting.

getValue(*fitpararray*)
Return the value of the parameter corresponding to the given array of values.

This method only works if the fitparameter is connected to an instance of *ParameterPool*. Therefore, create instances of *Fitparameter* always using *ParameterPool.newParameter()*.

name
(*str*) Name of the parameter. Read-only.

fixed
(*bool*) Determines if fitparameter is fixed (*True*) or unfixed (*False*) during fitting procedures.

start_val
(*number*) Start value of the fitparameter for fitting procedures.

lower_lim
(*number*) Lower limit of the fitparameter for fitting procedures.

upper_lim
(*number*) Upper limit of the fitparameter for fitting procedures.

index
(*int*) Determines which entry in the fitparameter array is interpreted as the value of this fitparameter. Usually this property should not be changed by you (the user) but by the instance of *ParameterPool* where the fitparameter is connected to.

complex
(*bool*) Signals if fitparameter is a complex number (*True*) or not (*False*). Read-only.

class Parameters.**ParameterPool**(*parfilename=None*)
Bases: object

Collects a pool of Parameter objects and connects them with a parameter file.

__init__(*parfilename=None*)
Initialize a new ParameterPool.

Read parameter initialisation from file **parfilename** if given. As soon as you connect a parameter file to the pool, its initialisation values have priority over local initialisations with *newParameter()*.

newParameter(*name*, *fixed=False*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)
Create a new *Fitparameter* inside the parameter pool and return a reference to it or return the reference to an already existing one with the same **name**. This is the usual way to create instances of *Fitparameter*.

Parameters

- **name** (*str*) – Name of the fit parameter. Use names without whitespaces. Otherwise there can be problems when using a parameter file.
- **fixed** (*bool*) – If *True*,the parameter will not be varied during a fit routine and the limits are not necessary.
- **start_val** (*number*) – Start value of the fit parameter.
- **upper_lim** (*number*) – Lower limit of the fit parameter.
- **upper_lim** – Upper limit of the fit parameter.

getParameter(*name*)
Return existing parameter with name **name** or return *None* if not existing.

readFromFile(*parfilename*)
Read parameters and there initialisation values from file *parfilename**, append them to the pool or overwrite existing once.

Lines in the parameter file should be in the format:
`<start_value> <fixed> <lower_limit> <upper_limit> <name> <comments/other stuff>`
Lines starting with # are ignored.
Values of <fixed> have to be either *0* or *1* (not *False* or *True*).

writeToFile(*parfilename*, *fitpararray=None*)
Write parameters and there initialisation value to file **parfilename**.

Can be used to create a template for a parameter initialisation file or to store parameters after fitting. If **fitpararray** is given, these values are written as start values. Else the stored start values are used.

getStartLowerUpper()
Return a tupel of *fitpararrays* of parameter start values, lower and upper limits for usage with *Fitparameter.getValue()*.

Each of the arrays contains the values in the order of occurence of the corresponding parameter in the pool (order of parameter creation), but only of those parameters which are not fixed. Real parameters first, then the complex ones.

setStartValues(*fitpararray*)
Set the start values of all parameters which are not fixed using **fitpararray** in the order of occurence in the pool (order of parameter creation). First real and then complex ones.

Can be used e.g. after finding goog start values with an Evolutionary fitter.

getFitArrayLen()
Return length of the *fitpararray* which is needed for *setStartValues()*, *writeToFile()* and *Fitparameter.getValue()* and which is given by *getStartLowerUpper()*.

getNames()
Return the names of all registered Fitparameters as a list in the same order they should be in the *fitpararray*.

4.2 Module *SampleRepresentation*

Deals with the sample representation for simulation of the reflectivity.

A multilayer sample is represented by a *Heterostructure* object. Its main pupose is to deliver the list of layers (Layer type of the Pythonreflectivity package from Martin Zwiebler) with defined susceptibilities at certain energies via :meth.'Heterostructure.getSingleEnergyStructure'. The layers within this heterostructure are represented by instances of *LayerObject* or of derived classes, which alows for a very flexibel modelling of the sample.

So far the following layer types are implemented:

- *LayerObject*: Layer with a constant (over energy) but fittable electric susceptibility tensor.
- *ModelChiLayerObject*: This layer type holds the electric susceptibility tensor as a user-defined function of energy.
- *AtomLayerObject*: This layer deals with compositions of atoms with different formfactors. The densities of the atoms can be varied during fitting procedures and plotted with using *plotAtomDensity()*. The formfactors are represented by instances of classes which are derived from *Formfactor* (the base class is abstract and cannot be used directly).

So far the following formfactor types are implemented:

- *FFfromFile*: Reads an energy-dependent formfactor as data points from a textfile. For energies between the data points the formfactor is linearly interpolated.

class SampleRepresentation.**Heterostructure**(*number_of_layers=0*, *multilayer_structure=None*)

Bases: object

Represents a heterostructure as a stack of instances of *LayerObject* or of derived classes. Its main pupose is to model the sample in a very flexibel way and to get the list of layers (Layer type of the Pythonreflectivity package from Martin Zwiebler) with defined susceptibilities at certain energies.

In contrast to Martin's list of Layer-type objects, this class contains all information also for different energies.

__init__(*number_of_layers=0*, *multilayer_structure=None*)
Create heterostructructure object.

Parameters

- **number_of_layers** (*int*) – gives the number of different layers
- **multilayer_structure** (*list*) – Makes it possible to define multilayers which contain identical layers several times. This can be done by passing a list containing the indices of layers from the lowest (e.g. substrate) to the highest (top layer, hit first by the beam). Default is [0,1,2,3, ...,number_of_layers-1]. Multilayer syntax is e.g.''[0,1,2,[100,[3,4,5,6]],7,.,1,..]'' which repeats 100 times the sequence of layers 3,4,5,6 in between 2 and 7 and later on layer 1 is repeated once.

setLayout(*number_of_layers*, *multilayer_structure=None*)
Change the layout of the heterostructure.

See *__init__()* for details. Only difference is: you cannot make changes which would remove layers.

setLayer(*index*, *layer*)
Place **layer** (instance of *LayerObject*) at position **index** (counting from 0, starting from the bottom or according to indices defined by **multilayer_structure**).

getLayer(*index*)
Return the instance of *LayerObject* which is placed at position **index** (counting from 0, starting from the bottom or according to indices defined by **multilayer_structure**).

getTotalLayer(*index*)
Return the instance of *LayerObject* which is placed at position **index** (counting from 0, starting from the bottom, repeated layers are counted repeatedly).

removeLayer(*index*)
Remove the instance of *LayerObject* which is placed at position **index** (counting from 0, starting from the bottom or according to indices defined by **multilayer_structure**).

index can also be a list of indices. BEWARE: The instance of *LayerObject* itself and the corresdonding instances of *Parameters.Fitparameter* are not deleted! So in a following fitting procedure, these parameters might still be varied even though they don't have any effect on the result.

getSingleEnergyStructure(*fitpararray*, *energy=None*)
Return list of layers (Layer type of the `Pythonreflectivity` package from Martin Zwiebler) which can be directly used as input for `Pythonreflectivity.Reflectivity()`.

energy in units of eV

N
(*int*) Number of different layers. Read-only.

N_total
(*int*) Total number of layers counting also multiple use of the same layer according to **multilayer_structure**. Read-only.

class SampleRepresentation.**LayerObject**(*chitensor=None*, *d=None*, *sigma=None*, *magdir='0'*)

Bases: `object`

Base class for all layer objects as the common interface. Speciallized implementation should inherit from this class.

__init__(*chitensor=None*, *d=None*, *sigma=None*, *magdir='0'*)

Parameters

- **d** (*Parameters.Parameter*) – Thickness. Unit is the same as for every other length used throughout the project and is not predefined. E.g. wavelength.
- **sigma** (*Parameters.Parameter*) – The roughness of the upper surface of the layer. Has dimension of length. Unit: see **d**.
- **chitensor** (list of *Parameters.Parameter*) – Electric susceptibility tensor of the layer. | *[chi]* sets *chi_xx = chi_yy = chi_zz = chi* | *[chi_xx,chi_yy,chi_z]* sets *chi_xx,chi_yy,chi_zz*, others are zero | *[chi_xx,chi_yy,chi_z,chi_g]* sets *chi_xx,chi_yy,chi_zz* and depending on **magdir** *chi_yz=-chi_zy=chi_g* (if *x*), *chi_xz=-chi_zx=chi_g* (if *y*) or *chi_xz=-chi_zx=chi_g* (if *z*) | *[chi_xx,chi_xy,chi_xz,chi_yx,chi_yy,chi_yz,chi_zx,chi_zy,chi_zz]* sets all the corresdonding elements
- **magdir** (*str*) – Gives the magnetization direction for MOKE. Possible values are *"x"*, *"y"*, *"z"* and *"0"* (no magnetization).

getChi(*fitpararray*, *energy=None*)
Return the electric susceptibility tensor as a list of numbers for a certain **energy** corresponding to the parameter values in **fitpararray** (see *Parameters*).

The returned list can be of length 1,3,4 or 9 (see *__init__()*). For the base implementation of *LayerObject* the parameter **energy** is not used. But it may be used by derived classes like *AtomLayerObject* and therefore needed for compatibility.l **energy** is measured in units of eV.

getD (*fitpararray*)
Return the thickness of the layer as a number corresponding to the parameter values in **fitpararray** (see *Parameters*).

The thickness is given in the unit of length you chose. You are free to choose whatever unit you want, but use the same for every length troughout the project.

getSigma (*fitpararray*)
Return the roughness of the upper surface of the layer as a number corresponding to the parameter values in **fitpararray** (see *Parameters*).

The thickness is given in the unit of length you chose. You are free to choose whatever unit you want, but use the same for every length troughout the project.

getMagDir (*fitpararray=None*)
Return magnetization direction

fitpararray is not used and just there to give a common interface. Maybe a derived classes will have a benefit from it.

d
Thickness of the layer. Unit is the same as for every other length used throughout the project and is not predefined. E.g. wavelength.

chitensor
Electric susceptibility tensor of the layer. See *__init__()* for details.

sigma
Roughness of the upper surface of the layer. Unit is the same as for every other length used throughout the project and is not predefined. E.g. wavelength.

magdir
Magnetization direction for MOKE. Possible values are *"x"*, *"y"*, *"z"* and *"0"* (no magnetization).

class SampleRepresentation.**ModelChiLayerObject** (*chitensorfunction*, *d=None*, *sigma=None*, *magdir='0'*)

Bases: *SampleRepresentation.LayerObject*

Speciallized layer to deal with an electrical suszeptibility tensor (Chi) which is modelled as function of energy.

BEWARE: The inherited property *chitensor* is now a function.

__init__ (*chitensorfunction*, *d=None*, *sigma=None*, *magdir='0'*)

Parameters

- **d** (*Parameters.Parameter*) – Thickness. Unit is the same as for every other length used throughout the project and is not predefined. E.g. wavelength.
- **sigma** (*Parameters.Parameter*) – The roughness of the upper surface of the layer. Has dimension of length. Unit: see **d**.
- **chitensorfunction** (*callable*) – Energy-dependent electric susceptibility tensor of the layer. It is supposed to be a function of two parameters (**fitpararray**, **energy**) which returns a list of either 1,3,4 or 9 real or complex numbers. See also documentation of Pythonreflectivity.
  - *[chi]* sets *chi_xx = chi_yy = chi_zz = chi*
  - *[chi_xx,chi_yy,chi_z]* sets *chi_xx,chi_yy,chi_zz*, others are zero

- *[chi_xx,chi_yy,chi_z,chi_g]* sets *chi_xx,chi_yy,chi_zz* and depending on **magdir** *chi_yz=-chi_zy=chi_g* (if *x*), *chi_xz=-chi_zx=chi_g* (if *y*) or *chi_xz=-chi_zx=chi_g* (if *z*)
- *[chi_xx,chi_xy,chi_xz,chi_yx,chi_yy,chi_yz,chi_zx,chi_zy,chi_zz]* sets all the corresdonding elements

- **magdir** (*str*) – Gives the magnetization direction for MOKE. Possible values are *"x"*, *"y"*, *"z"* and *"0"* (no magnetization).

getChi (*fitpararray*, *energy*)

Return the electric susceptibility tensor as a list of numbers for a certain **energy** corresponding to the parameter values in **fitpararray** (see *Parameters*).

energy is measured in units of eV.

class SampleRepresentation.**AtomLayerObject**(*densitydict={}*, *d=None*, *sigma=None*, *magdir='0'*, *densityunitfactor=1.0*)

Bases: *SampleRepresentation.LayerObject*

Speciallized layer class to deal with compositions of atoms and their energy dependent formfactors (which can be obtained from absorption spectra).

Especially usefull to deal with atomic layers, but can also be used for bulk. The atoms and their formfactors have to be registered a the class (with registerAtom) before they can be used to instantiate a new AtomLayerObject. The atom density can be plotted with *plotAtomDensity()*. Density is measured in mol/cm^3 (as long as no **densityunitfactor** is applied)

__init__(*densitydict={}*, *d=None*, *sigma=None*, *magdir='0'*, *densityunitfactor=1.0*)

Parameters

- **densitydict** – a dictionary which contains atom names (strings, must agree with before registered atoms) and densities (must be instances of *Parameters.Parameter* or of a derived class).
- **d** (*Parameters.Parameter*) – Thickness. Unit is the same as for every other length used throughout the project and is not predefined. E.g. wavelength.
- **sigma** (*Parameters.Parameter*) – The roughness of the upper surface of the layer. Has dimension of length. Unit: see **d**.
- **chitensor** (list of *Parameters.Parameter*) – Electric susceptibility tensor of the layer. | *[chi]* sets *chi_xx = chi_yy = chi_zz = chi* | *[chi_xx,chi_yy,chi_z]* sets *chi_xx,chi_yy,chi_zz*, others are zero | *[chi_xx,chi_yy,chi_z,chi_g]* sets *chi_xx,chi_yy,chi_zz* and depending on **magdir** *chi_yz=-chi_zy=chi_g* (if *x*), *chi_xz=-chi_zx=chi_g* (if *y*) or *chi_xz=-chi_zx=chi_g* (if *z*) | *[chi_xx,chi_xy,chi_xz,chi_yx,chi_yy,chi_yz,chi_zx,chi_zy,chi_zz]* sets all the corresdonding elements
- **magdir** (*str*) – Gives the magnetization direction for MOKE. Possible values are *"x"*, *"y"*, *"z"* and *"0"* (no magnetization).
- **densityunitfactor** (*float*) – If the densities in densitydict are measured in another unit than mol/cm^3, state this value which translates your generic density to the one used internally. I.e.:

```
rho_in_mol_per_cubiccm = densityunitfactor * rho_in_
↪whateverunityouwant
```

getDensitydict (*fitpararray=None*)
Return the density dictionary either with evaluated paramters (needs **fitpararray**) or with the raw *Parameters.Parameter* objects (use **fitparraray** = *None*).

getChi (*fitpararray*, *energy*)
Return the electric susceptibility tensor as a list of numbers for a certain **energy** corresponding to the parameter values in **fitpararray** (see *Parameters*).

energy is measured in units of eV.

classmethod registerAtom(*name*, *formfactor*)
Register an atom called **name** for later use to instantiate an AtomLayerObject.

formfactor as to be an instance of *Formfactor* or of a derived class.

classmethod getAtom(*name*)
Return the *Formfactor* object registered for atom **name**.

classmethod getAtomNames()
Return a list of names of registered atoms.

class SampleRepresentation.**Formfactor**
Bases: object

Base class to deal with energy-dependent atomic form-factors.

This base class is an abstract class an cannot be used directly. The user should derive from this class if he wants to build his own models.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

getFF (*energy*, *fitpararray=None*)
Return the formfactor for **energy** corresponding to **fitpararray** (if it depends on it) as 9-element list of complex numbers.

energy is measured in units of eV.

class SampleRepresentation.**FFfromFile**(*filename*, *linereaderfunction=None*, *energyshift=<Parameters.Parameter object>*)
Bases: *SampleRepresentation.Formfactor*

Class to deal with energy-dependent atomic form-factors which are tabulated in files.

__init__ (*filename*, *linereaderfunction=None*, *energyshift=<Parameters.Parameter object>*)
Initializes the FFfromFile object with an energy-dependent formfactor given as file.

Parameters

- **filename** (*str*) – Path to the text file which contains the formfactor.
- **linereaderfunction** (*callable*) – This function is used to convert one line from the text file to data. It should be a function which takes a string and returns a tuple or list of 10 values: `(energy,f_xx,f_xy,f_xz,f_yx,f_yy,f_yz,f_zx,f_zy,f_zz)`, where *energy* is measured in units of *eV* and formfactors in units of *e/atom* (dimensionless). It can also return *None* if it detects a comment line. You can use *FFfromFile.createLinereader()* to get a standard function, which just reads this array as whitespace seperated from the line.
- **energyshift** (*Parameters.Parameter*) – Species a fittable energyshift between the energy-dependent formfactor from **filename** and the *real* one in the reflectivity measurement. So the formfactor delivered from *FFfromFile.getFF()* will not be *formfactor_from_file(E)* but *formfactor_from_file(E+energyshift)*.

static createLinereader(*complex_numbers=True*)
Return the standard linereader function for usage with *FFfromFile.__init__()*.

This standard linereader function reads energy and complex elements of the formfactor tensor as a whitespace-seperated list (i.e. 10 numbers) and interpretes "#" as comment sign. If **complex_numbers** = *False* then the reader reads real and imaginary part of every element seperately, i.e. every line has to consist of 19 numbers seperated by whitespaces:

```
energy f_xx_real ff_xx_im ... f_zy_real f_zy_im f_zz_real f_zz_im
```

getFF(*energy*, *fitpararray=None*)
Return the (energy-shifted)formfactor for **energy** as an interpolation between the stored values from file as 9-element 1-D numpy array of complex numbers.

Parameters

- **energy** (*float*) – Measured in units of eV.
- **fitpararray** – Is actually only needed when an energyshift has been defined.

maxE
Upper limit of stored energy range. Read-only.

minE
Lower limit of stored energy range. Read-only.

SampleRepresentation.**plotAtomDensity**(*hs*, *fitpararray*, *colormap=[]*, *atomnames=None*)
Convenience function. Create a bar plot of the atom densities of all instances of *AtomLayerObject* contained in the *Heterostructure* object **hs** corresdonding to the **fitpararray** (see *Parameters*) and return the plotted information as dictionary.

You can define the colors of the bars with **colormap**. Just give a list of matplotlib color names. They will be used in the given order. You can define which atoms you want to plot or in which order. Give **atomnames** as a list of strings. If **atomnames** is not given, the bars will have different width, such that overlapped bars can be seen.

SampleRepresentation.**KramersKronig**(*energy*, *absorption*)
Convinience funtion. Performs the Kramers Kronig transformation. It is just a wrapper for `Pythonreflectivity.KramersKroning()` from Martins Zwieblers `Pythonreflectivity` package.

Parameters

- **energy** – an ordered list/array of L energies (in eV). The energies do not have to be envenly spaced, but they should be ordered.
- **absorption** – a list/array of real numbers and length L with absorption data

4.3 Module *Experiment*

Deals with the description of the experiment and brings experimental and simulated data together. It contains currently only the class *ReflDataSimulator*, which does this job.

class Experiment.**ReflDataSimulator**(*mode*, *length_scale=1e-09*)
Bases: `object`

Holds the experimental data, simulates it according to the settings and fitparameters and can directly deliver the sum of squared residuals (*getSimData()*) and the residuals themselfs (*getResidualsSSR()*), which both describe the difference between data and simulation at a certain parameter set. It can be in different modes which determins which data or which derived data is stored and simulated.

__init__(*mode*, *length_scale=1e-09*)
Initialize the ReflDataSimulator with a certain mode.

Parameters

- **mode** (*string*) – The following modes are implemented so far:
  - 'l' - for linear polarized light, only reflectivity for sigma and pi polarization will be stored and simulated
  - 'c' - for circular polarized light, only reflectivity for left circular and right circular polarization will be stored and simulated
  - 'x' - for xmcd, only the difference between the reflectivity for right circular and left circular polarization will be stored and simulated
  - 'cx<xfactor>' - for the reflections of circular pol. light and the xmcd signal (which should usually been calculated from the left and right circ. pol.) simultaniously '<xfactor>' is optional and can be used to multiply the xmcd signal with this value. This can be usefull to give the xmcd more or less weight during fitting e.g.'cx20' or 'cx0.1'
  - 'lL', 'cL', 'xL', 'cLx<xfactor>', - as before, but instead of the corresponding reflectivities themselfs their logarithms are stored and simulated.
- **length_scale** (*float*) – Defines in which unit lengths are measured in your script. The unit is then **length_scale** * meters. Default is **length_scale** = *1e-9* which means *nm*. It is important to define it here due to conversion between energies and wavelength.

ReadData(*files*, *linereaderfunction*, *energies=None*, *angles=None*, *filenamereaderfunction=None*, *pointmodifierfunction=None*, *headerlines=0*)
Read the data files and store the data corresponding to the **mode** specified with instanciation (see *ReflDataSimulator.__init__()*)

This function enables a very flexible reading of the data files. Logically, this function uses data points which consist of the independent variables energy and angle, and the reflectivities as dependent variables (rsigmag,rpi,rleft,rright,xmcd). So one point is specified by (energy,angle,rsigmag,rpi,rleft,rright,xmcd) with energies in eV and angles in degrees. Where the values for the independent variables comes from can differ: either from lists (**energies**,**angles**), from the filenames (**filenamereaderfunction**) or from the lines in the data file (**linereaderfunction**).

Parameters

- **files** (*str or list of str*) – Specifies the set of data files. Either a list of filenames or one foldername of a folder containing all the data files (and only them!).
- **linereaderfunction** (*callable*) – A function given by the user which takes one line of an input file as string and returns a list/tuple of real numbers *(energy,angle,rsigma,rpi,rleft,rright,xmcd)*. Entries can also be 'None'. Exceptions will only be trown if the needed information for the specified **mode** is not delivered. An easy way to create such a function is to use the method *createLinereader()*.
- **energies** (*list of floats*) – Only possible to be different from *None* if **files** is a list of filenames and **angles** is *None*. Gives the energies which belong to the corresponding files (same order) as floats.
- **angles** (*list of floats*) – Only possible to be different from *None* if **files** is a list of filenames and **energies** is *None*. Gives the angles which belong to the corresponding files (same order) as floats.
- **filenamereaderfunction** (*callable*) – A user-defined function which reads energies and/or angles from the filenames of the data files. This function should take a string (a filename without path), extract energy and/or angle out of it and return this as a tuple/list

(energy,angle). Both entries can also be set to *None*, but their will be an exception if the needed information for the data points can also not be obtained from the **linereaderfunction**.

- **pointmodifierfunction** (*callable*) – A user-definde function which is used to modify the obtained information. It takes the tuple/list of independent and dependent variables of a single data point and returns a modified one. It can be used for example if the data file contains qz values instead of angles. In this case you can read the qz values first as angles and replace them afterwards with the angles calculated out of it with the **pointmodifierfunction**. Of course you can also use a adopted **linereaderfunction** for this purpose (if all necessary information can be found in one line of the data files).
- **headerlines** (*int*) – specifies the number of lines which should be ignored at the top of each file.

setModel (*heterostructure*, *reflmodifierfunction=None*, *MultipleScattering=True*, *MagneticCutoff=1e-50*)

Set up the model for the simulation of the reflectivity data.

The simulation of the reflectivities is in prinicple done by using the information about the sample stored in **heterostructure** (of type *SampleRepresentation.Heterostructure*). The calculated reflectivities are then given to the **reflmodifierfunction** (takes one number or numpy array and the fitpararray; returns one number or a numpy array). This function has to be defined by the user and can be used for example to multiply the reflectivity by a global number and/or to add a common background. To make these numbers fittable, use the fitparameters registerd at an instance of `Paramters.ParamterPool`. Example:

```
pp=Paramters.ParameterPool("any_parameterfile")
...
b=pp.NewParameter("background")
m=pp.NewParameter("multiplier")
reflmodifierfunction=lambda r, fitpararray: b.getValue(fitpararray) + r * m.
↪getValue(fitpararray)
```

and give this function to *setModel()*.

BEWARE: The reflmodifierfunction is called very often during fitting procedures. Make it performant!

With **MultipleScattering** you can switch on (*True*) and off (*False*) the simulation of multiple scattering. *False* is 20 percent faster. Default is *True*. Has no effect on calculations that require the full matrix.

MagneticCutoff: If an off-diagonal element of chi (chi_g) fulfills abs(chi_g)<MagneticCutoff, it is set to zero. It defaults to 10e-50.

The last two parameters are directly passed to `Pythonreflectivity.Reflectivity()`. See also the Documentation of `Pythonreflectivity`.

getLenDataFlat ()

Return length of the flat data representation.

It will be the number of measured data points times 2 for mode “l” and “c”, only the number of measured data points for mode “x” and the number of measured data points times 3 for mode “cx”

getSimData (*fitpararray*)

Return simulated data according to the bevor set-up model and the parameter values given with **fitpararray** (see also *Parameters*).

The retured data is a list and has on of the following or similar shapes:

```
[[energy1,[angle1,....angleN], [rsigma1, .... rsigmaN], [rpi1,...rpiN]], ...
↪[energyL,[angle1,....angleK], [rsigma1, .... rsigmaK], [rpi1,...rpiK]]
[[energy1,[angle1,....angleN], [rleft1, .... rleftN], [rright1,...rrightN]], .
↪..[energyL,[angle1,....angleK], [rleft1, .... rleftK], [rright1,...rrightK]]
[[energy1,[angle1,....angleN], [xmcd1, .... xmcdN]], ...[energyL,[angle1,....
↪angleK], [xmcd1, .... xmcdK]]
[[energy1,[angle1,....angleN], [rleft1, .... rleftN], [rright1,...rrightN],␣
↪[xmcd1, .... xmcdN]]], ...[energyL,[angle1,....angleK], [rleft1, ....␣
↪rleftK], [rright1,...rrightK],[xmcd1, .... xmcdK]]
```

getExpData()
Return stored experimental data.

The retured data is a list and has on of the following or similar shapes:

```
[[energy1,[angle1,....angleN], [rsigma1, .... rsigmaN], [rpi1,...rpiN]], ...
↪[energyL,[angle1,....angleK], [rsigma1, .... rsigmaK], [rpi1,...rpiK]]
[[energy1,[angle1,....angleN], [rleft1, .... rleftN], [rright1,...rrightN]], .
↪..[energyL,[angle1,....angleK], [rleft1, .... rleftK], [rright1,...rrightK]]
[[energy1,[angle1,....angleN], [xmcd1, .... xmcdN]], ...[energyL,[angle1,....
↪angleK], [xmcd1, .... xmcdK]]
[[energy1,[angle1,....angleN], [rleft1, .... rleftN], [rright1,...rrightN],␣
↪[xmcd1, .... xmcdN]]], ...[energyL,[angle1,....angleK], [rleft1, ....␣
↪rleftK], [rright1,...rrightK],[xmcd1, .... xmcdK]]
```

getSSR(*fitpararray*)
Return sum of squared residuals as float according to the parameterset given by **fitpararray** (see also *Parameters*).

getResidualsSSR(*fitpararray*)
Return the residuals and the sum of squared residuals according to the parameterset given by **fitpararray** (see also *Parameters*).

The information is returned as tuple: array of differences between simulated and measured data, sum of squared residuals.

plotData(*fitpararray*, *simcolor='r'*, *expcolor='b'*, *simlabel='simulated'*, *explabel='experimental'*)
Plot simulated and experimental data.

This function generates a plot at the first call and refreshes it if called again.

Parameters

- **simcolor** (*str*) – Specifies the color of the simulated data for the plotting with pyplot (see https://matplotlib.org/users/colors.html). Default is red.
- **expcolor** (str are supposed to be strings which specify a color for the plotting with pyplot (see https://matplotlib.org/users/colors.html).) – Specifies the color of the experimental data for the plotting with pyplot (see https://matplotlib.org/users/colors.html). Default is blue.
- **simlabel** (*str*) – Label shown in the legend of the plot for the simulated data. Default is *"simulated"*.
- **explabel** (*str*) – Label shown in the legend of the plot for the experimental data. Default is *"experimental"*.

setMode(*mode*)
Change the mode after instantiation.

Be carefull with this function. Errors can occur if the mode does not fit to the available information in the data files.

Parameters **mode** (*string*) – The following modes are implemented so far:

- 'l' - for linear polarized light, only reflectivity for sigma and pi polarization will be stored and simulated
- 'c' - for circular polarized light, only reflectivity for left circular and right circular polarization will be stored and simulated
- 'x' - for xmcd, only the difference between the reflectivity for right circular and left circular polarization will be stored and simulated
- 'cx<xfactor>' - for the reflections of circular pol. light and the xmcd signal (which should usually been calculated from the left and right circ. pol.) simultaniously '<xfactor>' is optional and can be used to multiply the xmcd signal with this value. This can be usefull to give the xmcd more or less weight during fitting e.g.'cx20' or 'cx0.1'
- 'lL', 'cL', 'xL', 'cLx<xfactor>', - as before, but instead of the corresponding reflectivities themselfs their logarithms are stored and simulated.

static createLinereader(*energy_column=None*, *angle_column=None*, *rsigma_column=None*, *rpi_column=None*, *rleft_column=None*, *rright_column=None*, *xmcd_column=None*, *commentsymbol='#'*)

Return a linereader function which can read lines from whitespace-seperated files and returns lists of real numbers *[energy,angle,rsigma,rpi,rleft,rright,xmcd]* (or *None* for a uncommented line).

With the parameters ... *_column* you can determin wich column is interpreted how. Column numbers are starting from 0.

mode

The current mode. See :meth:'.__init__'for possible modes. Read-only.

hcfactor

Planck constant times the speed of light in units of *eV* times the unit of length which was defined by **length_scale** with *__init__()*. Read-only.

4.4 Module *Fitters*

Contains different optimization algorithms designed to fit reflectivity data. They take advantage of parallelization to be used on multiprocessor system.

The algorithms are developed by Martin Zwiebel and I just adopted them with slight changes to PyXMRTool. More information can be found in the PhD thesis of Martin Zwiebler.

Fitters.**Evolution**(*costfunction*, *parameter_settings*, *iterations*, *number_of_cores=1*, *generation_size=300*, *mutation_strength=0.01*, *elite=2*, *parent_percentage=0.25*, *control_file=None*, *plotfunction=None*)

Evolutionary fit algorithm. Slow but good in finding the global minimum. Return the optimized parameter set and the coresponding value of the costfunction.

Parameters

- **costfunction** (*callable*) – A function which returns a measure (cost) for the difference between measurement and simulated data according to the paramter set given as list of values. Usually the sum of squared residuals (SSR) is used as cost. It should usally be the method `SampleRepresentation.ReflDataSimulator.getSSR()` of an instance of `SampleRepresentation.ReflDataSimulator` wrapped in a function.

The wrapping is necessaray due to some implemetation issues connected to the parallelization. Example for the wrapping:

```
simu = SampleRepresentation.ReflDataSimulator("l")
...
def cost(fitpararray):
    return simu.getSSR(fitpararray)
```

Pass then the function *cost* as **costfunction**. It can also be any other function which takes the array of fit parameters and returns one real value which should be minimized by `Evolution()`.

- **parameter_settings** (*tuple of lists of floats*) – Sets start values, lower and upper limit of the parameters as *(startfitparameters, lower_limits, upper_limits)*, where each of the entries is an list/array of values of same length.
- **iterations** (*int*) – number of iterations/generations
- **number_of_cores** (*int*) – Number of jobs used in parallel. Best performance when set to the number of available cores on your computer.
- **generation_size** (*int*) – Generate this many individual fit parameter sets in each generation.
- **mutation_strength** (*float*) – Mutates children by adding this factor times (upper_limit - lower_limit) –> use rather small values
- **elite** (*int*) – Remember the best individuals for the next generation.
- **parent_percentage** (*flota*) – Use this fraction of a gereneration (the best) for reproduction.
- **control_file** (*str*) – Filename of a control file. If it is given, you can abort the optimization routine by writing "terminate 1" to the beginning of its first line.
- **plotfunction** (*callable*) – Function which is used to plot the current state of fitting (simulated data with currently best parameter set) after every iteration if given. It should take only one parameter: the array of fitparameters.

This Evolutionary algorithm is mainly the same as Martins. Only the rule for mutation has changed:

Martin: `children[i]=children[i] * (1 + s * random float(-1,1))`
I: `children[i]=children[i] + s * random float(-1,1)*(upper_limits-lower_limits)`

Fitters.**Levenberg_Marquardt_Fitter**(*residualandcostfunction*, *parameter_settings*, *parallel_points*, *number_of_cores=1*, *strict=True*, *control_file=None*, *plotfunction=None*)

Modified Levenberg-Marquard algorithm (see PhD thesis of Martin Zwiebler). Good convergence, but might end up in a local mininum. Return the optimized parameter set and the coresponding value of the costfunction.

Parameters

- **residualandcostfunction** (*callable*) – A function which returns the differences between simulated and measured data points (residuals) as list and a scalar measure (cost) for these differences in total according to the paramter set given as list of values. Usually the sum of squared residuals (SSR) is used as cost. It should usally be the method `SampleRepresentation.ReflDataSimulator.getResidualsSSR()` of an

instance of `SampleRepresentation.ReflDataSimulator` wrapped in a function. The wrapping is necessaray due to some implemetation issues connected to the parallelization. Example for the wrapping:

```
simu = SampleRepresentation.ReflDataSimulator("1")
...
def rescost(fitpararray):
    return simu.getResidualsSSR(fitpararray)
```

Pass then the function *rescost* as **costfunction**. It can also be any other function which takes the array of fit parameters and returns a tuple of 1.) a list of residuals (will be used to determine derivatives) 2.) a value of the costfunction which should be minimized by `Levenberg_Marquardt_Fitter()`.

- **parameter_settings** (*tuple of lists of floats*) – Sets start values, lower and upper limit of the parameters as *(startfitparameters, lower_limits, upper_limits)*, where each of the entries is an list/array of values of same length.
- **parallel_points** (*int*) – This should be something like the number of threads that can run in parallel/number of cores. The algorithm will first find a direction for a good descent and then check this number of points on the line. The best one will yield the new fit parameter set.
- **number_of_cores** (*int*) – Number of jobs used in parallel. Best performance when set to the number of available cores on your computer.
- **strict** (*bool*) – Usually this algorithm fails if the residuals are locally independent of one of the parameters. If you set **stict** = *False* this parameter will be neglected locally.
- **control_file** (*str*) – Filename of a control file. If it is given, you can abort the optimization routine by writing "terminate 1" to the beginning of its first line.
- **plotfunction** (*callable*) – Function which is used to plot the current state of fitting (simulated data with currently best parameter set) after every iteration if given. It should take only one parameter: the array of fitparameters.

Hint for API documentation: Inherited members are not repeated in the documentaion of derived classes.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

F

G