PyXMRTool Documentation

Release 0.9

Yannic Utz

Jul 17, 2018

CONTENTS:

CHAPTER

ONE

TODO: INTRODUCTION

to be written

CHAPTER

TWO

TODO: INSTALLATION

to be written

CHAPTER

THREE

TODO: USAGE

to be written

CHAPTER

FOUR

API

4.1 Module *Parameters*

Provide an easy and transparent handling of fit parameters

class Parameters.**Parameter**(*value=None*)
Bases: object

Base class for parameter. Contains read-only property 'value'. Supports creation of dependent parameters by arithmetic operations.

__init__(*value=None*)
Initialize and check if numeric type.

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=0*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)
Bases: *Parameters.Parameter*

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

__init__(*name*, *fixed=0*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)
Initialize a fitparameter at least with a name.

If 'fixed=1',the parameter will not be varied during a fit routine and the limits are not necessary.

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.

name

fixed

start_val

lower_lim

upper_lim

index

complex

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'.

As soon as you connect a parameter file to the pool, its initialisation values have priority over local initialisations with self.newParameter("name",fixed,start_value, lower_lim, upper_lim).

newParameter(*name*, *fixed=0*, *start_val=None*, *lower_lim=None*, *upper_lim=None*)
Create a New Parameter and return a reference to it or return the reference to an existing one.

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

readFromFile(*parfilename*)
Read parameters from file, 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> Lines starting with '#' are ignored.

writeToFile(*parfilename*, *fitpararray=None*)
Write parameters to file.

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

getStartLowerUpper()
Return a tupel of arrays of parameter start values and limits in the order of occurence in the pool (order of parameter creation) of parameters which are not fixed. Real parameters first, then the complex ones.

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

Can be used befor writeToFile() if you want to write out results of a fit to a file.

getFitArrayLen()

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

4.2 Module *SampleRepresentation*

Deals with the sample representation for simulation of the reflectivity.

class SampleRepresentation.**Heterostructure**(*number_of_layers=0*, *multilayer_structure=None*)
Represents a heterostructructure as a stack of layers.

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.

'number_of_layers' gives the number of different layers. With 'multilayer_structure' multilayers which contain identical layers several times can be defined. This can be defined by as a list containing the indices of layer 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 constructor 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 '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 'multilayer_structure').

getTotalLayer(*index*)
Return the instance of LayerObject which is placed at position 'index' (counting from 0, starting from the bottom, even if a latter sequence is repeated with help of 'multilayer_structure').

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

'index' can also be a list of indices. BEWARE: The instance of LayerObject itself and the corresdonding Parameters are not deleted!

getSingleEnergyStructure(*fitpararray*, *energy=None*)
Return list of layers (layer type from Pythonreflectivity) which can be directly used as input for 'Pythonreflectivity.Reflectivity( )'

'energy' in units of eV

N
Return number of different layers (i.e. number of different indices).

N_total
Return total number of layers (counting also multiple use of the same layer according to 'multilayer_structure').

class SampleRepresentation.**LayerObject**(*chitensor=None*, *d=None*, *sigma=None*, *magdir='0'*)
Base class for all layer objects as the common interface. Speciallized implementation should inherit from this class.

It handels the basic properties of

__init__(*chitensor=None*, *d=None*, *sigma=None*, *magdir='0'*)
Creates a new Layer.

'd' is its thickness, 'sigma' is the roughness of its upper surface, 'chitensor' is its electric susceptibility tensor, and 'magdir' gives the magnetization directrion for MOKE 'd', 'sigma' ,and the entries of 'chitensor' are expected to be an instances of a "Parameter" class (also "Fitparamter"). 'd' and 'sigma' are both of dimension length. You are free to choose whatever unit you want, but use the same for every length troughout the project. 'chitensor' is a list of either 1,3, 4 or 9 elements (see also documentation for Pythonreflectivity). Each of these elements is supposed to be of type Parameter or of a derived class. [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

getChi (*fitpararray*, *energy=None*)
Return the chitensor as a list of numbers for a certain energy.

For the base implementation of class 'LayerObject' the parameter 'energy' is not used. But it may be used by derived classes like 'AtomLayerObject'. 'energy' in units of eV.

getD (*fitpararray*)
Return the thickness d as a an actual number.

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 sigma as a an actual number.

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 magdir.

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

class SampleRepresentation.**ModelChiLayerObject**(*chitensorfunction*, *d=None*, *sigma=None*, *magdir='0'*)
Speciallized Layer to deal with a Suszeptibility Tensor (Chi) which is modelled as function of energy.

Beware: The property chitensor is here supposed to be a function.

__init__ (*chitensorfunction*, *d=None*, *sigma=None*, *magdir='0'*)
Creates a new ModelChiLayer.

'd' is its thickness, 'sigma' is the roughness of its upper surface, and 'magdir' gives the magnetization directrion for MOKE 'd', 'sigma' ,and the entries of 'chitensor' are expected to be an instances of a "Parameter" class (also "Fitparamter"). 'd' and 'sigma' are both of dimension length. You are free to choose whatever unit you want, but use the same for every length troughout the project. 'chitensorfunction' should be a function of two parameters (fitpararray,energy) and should return a list of either 1,3, 4 or 9 elements (see also documentation for Pythonreflectivity). Each of these elements is supposed to be a number (real or imaginary). [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

getChi (*fitpararray*, *energy*)
Return the chitensor as a list of numbers for a certain energy.

'energy' in units of eV.

class SampleRepresentation.**AtomLayerObject**(*densitydict={}*, *d=None*, *sigma=None*, *magdir='0'*, *densityunitfactor=1.0*)
Speciallized Layer 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() (see convenience functions). Density in mol/cm^3 (as long as no densityunitfactor is applied)

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

'densitydict' ist a dictionary which contains atom names (strings, must agree with before registered atoms) and densities (must be instances of the Parameter class or derived classes). For the other parameters see 'LayerObject'.

If the densities in densitydict are measured in another unit than mol/cm^3, state the 'densityunitfactor' 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 'Parameter' objects (use fitparraray=None).

getChi (*fitpararray*, *energy*)
Return the chitensor as a list numbers of for a certain energy.

'energy' in units of eV.

classmethod registerAtom(*name*, *formfactor*)
Register an atom 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**
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).

'energy' in units of eV.

class SampleRepresentation.**FFfromFile**(*filename*, *linereaderfctn=None*, *energyshift=<Parameters.Parameter object>*)
Class to deal with energy-dependent atomic form-factors which are tabulated in files.

__init__ (*filename*, *linereaderfctn=None*, *energyshift=<Parameters.Parameter object>*)
Initializes the FFfromFile object with the data from filename.

The 'linereaderfctn' 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) 'energy' in units of eV, 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' has to be an instance of 'Paramater' or of a derived class. It can be used to specify a fittable energyshift between the energy-dependent formfactor from filename and the 'real' one in the reflectivity measurement. So the formfactor delivered from getFF() will not be formfactor_from_file(E) but formfactor_from_file(E+energyshift).

static createLinereader(*complex_numbers=True*)
Return the standard linereader function.

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==0 then the reader reads real and imaginary part of every element sepeately, i.e. every line has to consist of 19 numbers seperated by whitespaces.

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

'energy' in units of eV. 'fitpararray' is actually only needed when an energyshift has been defined.

SampleRepresentation.**plotAtomDensity**(*hs*, *fitpararray*, *colormap=[]*, *atomnames=None*)
Make a plot of the atom densities of all AtomLayerObjects contained in the 'Heterostructure' 'hs' corresdonding to the 'fitpararray' and return the plotted information.

You can define the colors of the bars. 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*)
Perform the Kramers Kronig transformation. Just a wrapper for the function Pythonreflectivity.KramersKroning(energy,absorption) from Martins Pythonreflectivity package.

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

4.3 Module *Experiment*

Deals the description of the experiment and brings experimental and simulated data together.

class Experiment.**ReflDataSimulator**(*mode*, *length_scale=1e-08*)
Holds the experimental data, simulates it according to the settings and fitparameters and can directly deliver chi_square (which measures the difference between the data and the simulation with a certain parameter set.

__init__(*mode*, *length_scale=1e-08*)
Initialize the ReflDataSimulator with a certain mode.

'mode' can be: 'l' - for linear polarized light, only reflectivity for sigma and pi polarization will be stored and simula
'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 there logarithms are stored and simulated.

Define with 'length_scale' in which units lengths are measured in your script. The unit is then 'length_scale'*meters. Standard is 'length_scale=10e-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.

This function enables a very flexible reading of the data files. Logically this function uses data point 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 this information comes from can differ.

At first, there are two different ways to specify the data files: Either a list of filenames (strings) or one foldername (string) of a folder containing all the data files (and only them!).

One possibility is that all information comes from the 'linereaderfunction'. This function can be defined by the user (or created with createLinereader()). It takes one line as a string and returns a list/tuple of real numbers (energy,angle,rsigma,rpi,rleft,rright,xmcd). Entries can also be 'None'. The function will complain only if the needed information for the specified 'mode' is not delivered.

Sometimes, not all the information on independent variables can be obtained from single lines of the file. To specify an independent variable which is valid for complete files there are 3 different possibilities, which cannot be mixed: Set the list 'energies': Only possible if 'files' is a list of filenames. Gives the energies which belong to the corresponding files (same order) as floats. Set the list 'angles': Only possible if 'files' is a list of filenames. Gives the angles which belong to the corresponding files (same order) as floats. Set 'filenamereaderfunction': Give a user-specified function to the function 'ReadData()'. It 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 information for the data points can also not be obtained from the linereaderfunction.

With the parameter 'pointmodifierfunction' you can hand over a functions which takes the list of independent and dependent variables of a single data point and returns a modified one. Can be used e.g. if the data file contains qz values instead of angles. The 'pointmodifierfunction' can calculate the angles. Of course you can also use a adopted linereaderfunction for this (if all necessary information can be found in one line of the data files).

'headerlines' specifies the number of lines which should be ignored in 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 reflectivities calculated by this are then given to the 'reflmodifierfunction' (takes one number or numpy array and the fitpararray; returns one number or numpy array). This funktion has to be defined by the user and can be used e.g. 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 the ParamterPool e.g

> pp=Paramters.ParameterPool("any_parameterfile") ... b=pp.NewParameter("background") m=pp.NewParameter("multiplier") reflmodifierfunction=lambda r, fitpararray: b.getValue(fitpararra) + 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 to 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.

getExpData ()
Return experimental data in the shape in which it is stored internally.

getSSR (*fitpararray*)
Return sum of squared residuals.

getResidualsSSR (*fitpararray*)
Return 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.

'simcolor' and 'expcolor' are supposed to be strings which specify a color for the plotting with pyplot (see https://matplotlib.org/users/colors.html). Defaults are red and blue. 'simlabel' and 'explabel' are the labels shown in the legend of the plot.

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

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

'mode' can be: 'l' - for linear polarized light, only reflectivity for sigma and pi polarization will be stored and simula
'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 there 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).

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

4.4 Module *Fitters*

Contains different optimization algorithms designed for 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 to PyXMRTool. More information can be found in the PhD thesis of Martin Zwiebler.

Fitters.**Evolution**()

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

'costfunction' should usally be the method 'getSSR' of an instance of 'ReflDataSimulator' (returns the sum of squared res

Can also be any other function which takes the array of fit parameters and returns one real value should be minimized by 'Evolution'.

('startfitparameters','lower_limits','upper_limits): tuple/list of arrays/lists of start values/lower limits/upper limits for the fit parameters. 'iterations': number of iterations 'number_of_cores': number of jobs used in parallel. Best performance when set to the number of available cores on your computer. 'generation_size': Use this many individual fit parameter sets in each step 'mutation_strength' Mutates by adding this factor times (upper_limit - lower_limit) –> use rather small values 'elite' Remember the best individuals for the next generation 'parent_percentage' #Take this subset for reproduction

If 'control_file' is given, you can abort the optimization routine by writing "terminate 1" to the beginning of its first line.

If 'plotfunction' is given, it will be used to plot the current state of fitting (simulated data with currently best parameter set) after every iteration. 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**()

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.

'residualandcostfunction' should usally be the method 'getResidualsSSR' of an instance of 'ReflDataSimulator'. Can also

1.) a list of residuals (will be used to determine derivatives) 2.) a value of the costfunction which should be minimized (usually the sum of squared residuals)

('startfitparameters','lower_limits','upper_limits): tuple/list of arrays/lists of start values/lower limits/upper limits for the fit parameters. 'parallel_points': 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': number of jobs used in parallel. Best performance when set to the number of available cores on your computer. 'strict': 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.

If 'control_file' is given, you can abort the optimization routine by writing "terminate 1" to the beginning of its first line.

If 'plotfunction' is given, it will be used to plot the current state of fitting (simulated data with currently best parameter set) after every iteration. It should take only one parameter: the array of fitparameters.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

E

F

G