

A short manual for GenX

M. Björck, G. Andersson

Version 0.91

August 2, 2007

1 Introduction

This manual is intended as an introduction to the GenX program. The program is an implementation of the Differential Evolution algorithm for constrained fitting of a general function to data. The Differential Evolution algorithm has proven to be a robust method for multidimensional optimization of non-linear functions.

By separating the function to be fitted and the model it is readily extensible to many problems. At present it has been used to fit x-ray reflectivity and diffraction data. The models are written in the script language Python together with the SciPy package. Using a script language makes the program a bit slower a compilable programming language such as C, but instead it permits a fast implementation of new ideas. The main idea of the program is to provide a flexible and extensible environment for fitting a model to data.

The manual starts with an explanation of the user interface. It continues by explaining how to write models. Finally it briefly explains how to use the existing package for simulating x-ray reflectivity.

2 Installation

The program will run on all operating systems on which Python and wxPython is implemented. These are at present, according to wxPython homepage: 32-bit Microsoft Windows, most Unix or Unix-like systems, and Macintosh OS X. Currently GenX has been tested on Windows XP, Mac OS X and Linux. To install the program the following Python components need to be installed:

- Python 2.3.5, www.python.org
- SciPy 0.3.2, www.scipy.org
- Numeric 23.5, numpy.scipy.org
- wxPython 2.6, www.wxpython.org

The versions in this list are the most up-to-date versions the program has been tested with. For Windows user there exists a package called Python Enthought Edition, code.enthought.com, which contains all the packages mentioned in the

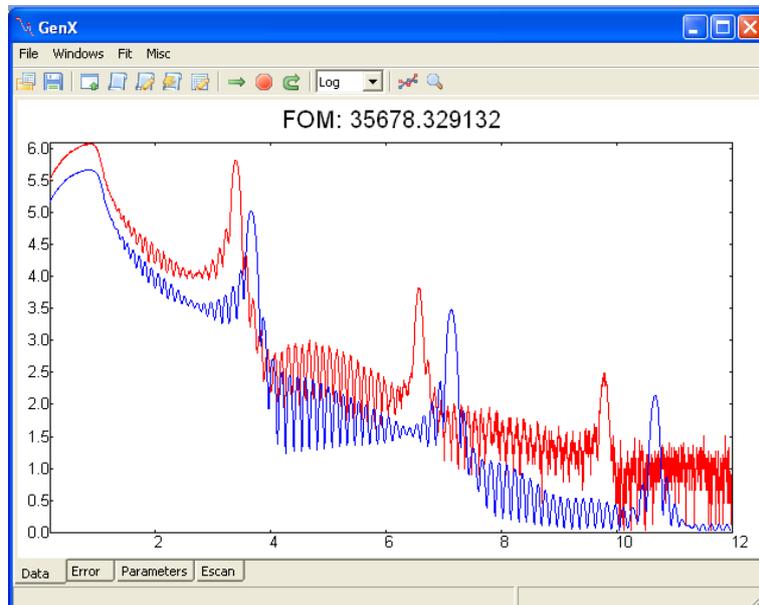


Figure 1: The main window of GenX, showing the menus and the different folders.

list above and even some more. It should be said that SciPy might need to be reinstalled on top of that installation to get it working properly.

When all the packages are installed. Unzip the program and run the file `GenX.py` with the Python interpreter. For Windows users it should be enough to double click on the file.

3 The User Interface

This section will briefly explain what the different menus and buttons in the user interface do.

3.1 Main window

The main window, figure 1, is controlled by menus at the top of the window or using the buttons on the toolbar. There are also different plot folders which are chosen by clicking on the respective folder at the bottom of the window.

3.1.1 File menu

Open Session This opens a previously saved session. The user is asked to give a file with the extension `*.dbm` and the session file contains the values in the Parameters window, the data and a path to the Python file containing the model. The command also loads the model file. The path in the session file is relative to the session file, i.e. moving the model file relative

to the session file will make it unloadable¹.

Save Session This saves the current session, see Open session.

Load Data Opens a window for importing and processing data. See section Data window.

Load Model This loads a Python file that should contain the model. The Python file will be executed in the global namespace. Any errors during the execution will be reported. In addition the file must contain a function called `Sim` which takes a data structure as argument and returns a list of calculated values. See section Making a Model for more details.

Export Data This exports the data to a three-column format. The first column is the x-data and the two remaining columns is the loaded data and the simulation, respectively.

Page Setup A native (i.e depends on the operating system) dialog box for setting up the printing. These changes only apply to the printing of the different plots.

Print Preview Print a preview of the graph in the active tab.

Print Prints the active graph in the notebook. Uses the settings defined in Page Setup

Print Parameters Prints the parameters from the Parameters window as a spreadsheet. This does not use the settings in Page Setup.

Exit Exits the program.

3.1.2 Windows Menu

This menu controls the auxiliary windows used for creating the model, loading the data and defining the parameters.

Edit Opens an editor. Its main use is to edit the model file. It should be noted that saving the file only saves the file and does not load the model into the program. This has to be done using Load Model in the file menu or the button on the toolbar.

Parameters Opens the Parameter window. This window defines which parameters should be fitted and their minimum values, maximum values and the guessed start/refined value. See section Parameter window.

Data Opens a window for importing and processing data. See section Data window.

¹Note it is possible to load a broken file by firstly load the model file and then open the session and after that load the model again. This makes it possible to load moved sessions and relink the model file.

3.1.3 Fit Menu

This menu controls the fitting process.

Start Starts and initializes the fitting. Initialization means that any changes made in the Parameter window are transferred to the fitting algorithm. It will also create a new (random) population of parameter vectors.

Stop Stops the fitting.

Resume Resumes the fit without the initialization. The fitting will proceed with the *same population*. Note that if the minimum and maximum values in the spreadsheet has changed it will not be loaded into the fitting routine.

3.1.4 Misc Menu

Nice Graph This is intended for producing publication quality plots. At present this is under development. Using this requires the package matplotlib, matplotlib.sourceforge.net. It is at present prone to crash the program!

Enable Zoom Turns the zoom on and off in the active plot in the notebook.

About.. Shows some information about the program.

3.1.5 The Toolbar

The toolbar allows fast access to the functions most used. The description that follows lists the different buttons starting from the left.

Open Session Opens a Session. See description in the File menu.

Save Session Saves a Session. See description in the File menu.

Data Opens the Data window.

Load Model Loads a model. See description in the File menu.

Edit Opens the editor. See description in the File menu.

Reload Model Reloads the modelfile.

Parameters Shows the parameterwindow.

Start Starts the fitting. See description in the Fit menu.

Stop Stops the fitting. See description in the Fit menu.

Resume Resumes the fitting. See description in the Fit menu.

Choiche list Changes the y-axis scale. The different scaling functions are defined in the file Settings.py

Line/Points Changes between lines and points for plotting the data.

Enable Zoom Enables the zoom. See description in the Misc menu.

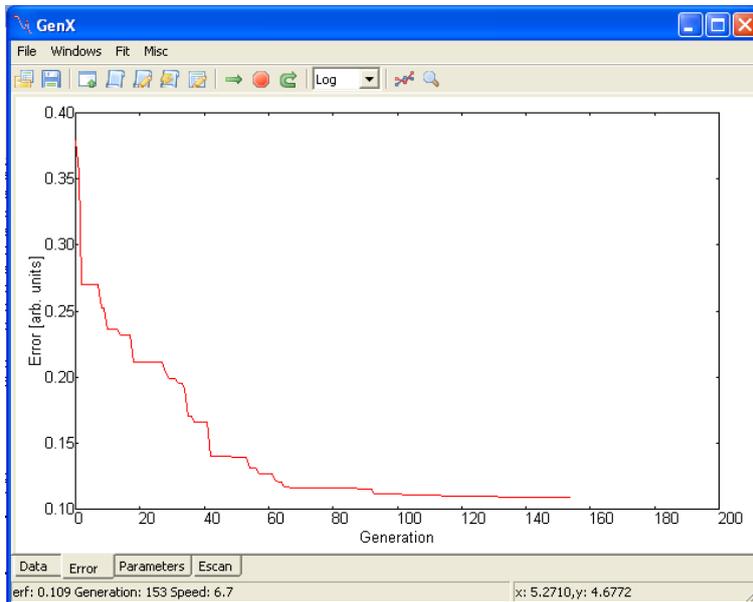


Figure 2: A picture of the Error folder. The red curve is the error as a function of the number of generations.

3.1.6 The Plots

In the bottom of the window there are several different tabs, see figure 1, that are used to display the data produced by the fitting routine. They are chosen by clicking on the respective tab.

Data Plot to show the data and the simulation. The scale of the y-axis is changed by choice list in the toolbar. The red line corresponds to the data and the blue is the simulation. This plot is updated after every iteration/generation.

Error The plot, see figure 2, shows the evolution of the Figure of Merit (FOM) as a function of the iteration/generation.

Parameters The y-axis corresponds to the normalized parameter space. The plot, see figure 3, shows the best value of each parameter, red dot, and the maximum and minimum of the parameter in the population, blue bar. Each parameter has its own red dot and blue bar, and the x-axis correspond to the row number in the spreadsheet. A normalized parameter space means that the plot is normalized between 0 and 1 where 0 is the minimum allowed value for the parameter and 1 is the maximum allowed value. This is quite useful during fitting since it allows the user to check that the minimum and maximum values do not limit the fitting routine. It can also be used for seeing how fast certain parameters converge. In addition it can be used to determine that all the parameters have converged when the blue bars have vanished or are small. In that case more generations/iterations will not improve the fit.

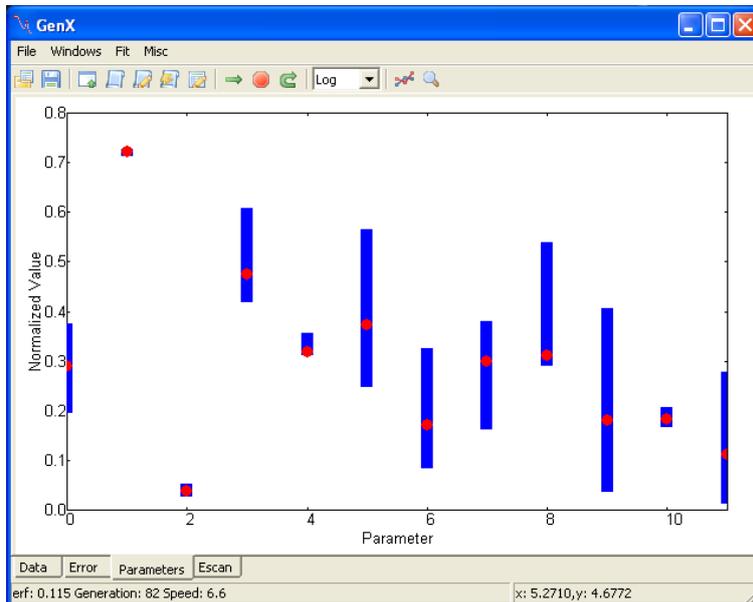


Figure 3: A picture of the Parameters plot. The red dots represents the best value so far and the blue bars represents the population spread. See text for details.

Escan This plot is used by the Escan button in the parameter window. It is used to display the dependence of the error function on one parameter. The y-axis corresponds to the error function and the x-axis to the parameter value which is represented by a blue line. The point that corresponds to the value in the grid is represented by a red dot and a red line is also included to indicate a 5 % increase in the errorfunction. See the section for the parameter window for more details and figure 5.

3.2 Parameter Window

The parameter window, figure 4, is used for defining which parameters are to be fitted and for setting the allowed minimum values, maximum values and the best guess values. In addition it contains some auxiliary functions such as simulation of the values and scanning of a parameter.

3.2.1 The Spreadsheet

The spreadsheet consists of 6 columns and an arbitrary number of rows. Each row represents one parameter and the values associated with it. The first column defines the parameter by a **function** that sets the parameter. This function has to take a single float number as input. The use of a function instead of supplying a parameter may seem a bit tedious, but it improves the flexibility and makes less room for hard to find reference errors in the programming. Also, if the right mouse button is clicked on a cell in the first column a pop-up menu will appear allowing the user to choose from member functions in the Layer, Stack, Sample

	Parameter	Value	Fit	Min	Max
1	MLFe.setD	8.000000	<input checked="" type="checkbox"/>	7.000000	20.000000
2	setLambda	29.000000	<input checked="" type="checkbox"/>	20.000000	30.000000
3	inst.setI0	1000000.000000	<input checked="" type="checkbox"/>	1000000.000000	1000000.000000
4	MLFe.setSigmar	2.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
5	MLPt.setSigmar	2.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
6	bufPt.setD	45.000000	<input checked="" type="checkbox"/>	40.000000	50.000000
7	bufFe.setD	5.000000	<input checked="" type="checkbox"/>	2.000000	10.000000
8	bufFe.setSigmar	2.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
9	bufPt.setSigmar	2.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
10	TopPt.setD	4.000000	<input checked="" type="checkbox"/>	0.050000	5.000000
11	TopPt.setSigmar	2.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
12	sub.setSigmar	5.000000	<input checked="" type="checkbox"/>	0.050000	8.000000
13		0.000000	<input type="checkbox"/>	0.000000	0.000000

Buttons: Simulate, Escan, Eproject

Figure 4: The parameter window with the spreadsheet and the auxiliary buttons at the bottom.

and Instrument objects defined in the model file². The second column, Value, represents either the first guess value as well as the best fit value supplied by the fitting algorithm. The third column, Fit, is a tick box. If this is ticked the parameter will be fitted. The fourth and fifth columns contain the allowed minimum and maximum values the parameter is allowed to take. The last column represents the error of the parameter for a 5% increase in the FOM or if the Chi2Fom, Chi2BarsFom or the Log2Fom a "real" confidence interval with a certain confidence set by the user in the menu Settings-Conf level. See section on the figure of merit function for more information. The rows in spreadsheet grows automatically as soon as the last row has a value.

If the first column is left blank the program will ignore the line. But if there is something written, and it is not a python object the program will give an error message in the form of a dialog box. Also when starting the fit all the variables (functions) in the spreadsheet are set to the corresponding value in the Value column whether or not the tick box in the Fit column is ticked.

3.2.2 Buttons

At present there are three buttons at the bottom of the window for extra function, see figure 4. The first from the left is Simulate. This executes a simulation by first setting all the parameters defined to the value in the value column. Then it calls the Sim function and plots the simulation together with the data.

The second button is the Escan button. This will scan a selected parameter and plot the error function versus the parameter value. The limits of the scan are determined by the minimum and maximum values. Before the error function is scanned all the defined parameters will be set to the values in Value column. The output is plotted in the Escan folder of the notebook. Before clicking on

²Note that user defined functions have to be entered manually

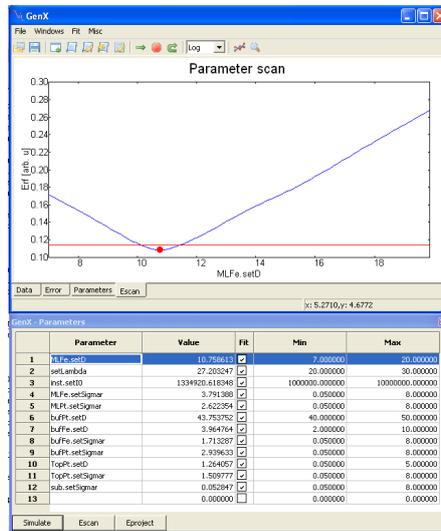


Figure 5: A screenshot of the procedure for an EScan run. Note that a row is selected in the Parameter window.

the Escan button a row has to be selected by clicking on the row numbering to the right of the spreadsheet, see figure 4. This function can be used to estimate the uncertainty of the fit by using the above mentioned 5% level. However, this needs a word of caution: it is not strictly the uncertainty in the parameter that is measured, since it is only a one-dimensional scan through the parameter space! If a correlation between the parameters exists, and it most probably does, this method will give a wrong number for the uncertainty.

The third button, Eproject, works similar to the Escan button. It will plot all evaluated FOM values from the fitting algorithm as a function of the selected parameter. It can be used to view the projected FOM as a function of one parameter.

3.3 Data Window

The data window represents a collection of routines for loading and processing of the data. The data that can be loaded is column based ASCII data. The program also supports the loading of multiple data sets. This can be used to fit multiple data sets or to cut out different regions of an existing data set. The window, see figure 6, is divided into two parts. The upper part is a toolbar and the lower part is called working area in this manual.

3.3.1 Toolbar

The toolbar consists of four buttons and a drop-down choice list. The choice list is used to choose on which data set the operations should be conducted. The first button from the left loads data from a file to the active data set. The second button adds a new empty data set to the list. The third button deletes the active data set. The button furthest to the right plots the data in the main window.

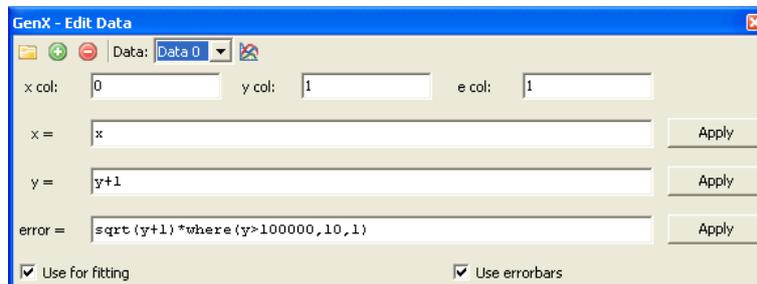


Figure 6: The Data window with the toolbar, top, and the working area, the rest of the window. **To be changed**

3.3.2 Working area

The two upper input fields in the working area are labeled **x col**, **y col** and **e col**³. The values in these fields represent which columns of the file contains the x data, y data and errors of the y data, respectively. This information is used when new data is loaded. The first column in the data file is denoted by 0. The text fields labeled **x=**, **y=**, **error=** is used for writing Python expressions that operate on the raw data, as read from the file. By clicking on the Apply button the typed command is executed. At the end of this section some useful expressions are presented. There is a tick box labeled **use for fitting** at the bottom right of the window. If this box is checked the active data set will be plotted and used for fitting. At the bottom right there is another tick box that controls whether or not to plot the error bars of the data.

In this paragraph some useful expressions for treating the raw data are presented. Any Python expression will work and, in addition, if a special function is written in the model file this can be called to process the data. First of all to reset the data to the raw data write, **x** in the x field and **y** in the y field. The general syntax for selecting data from an array is

```
x[start:end:stride]
```

If a special interval of the data needs to be fitted;

```
x[20:-300]
```

where the first value is the starting point (number of elements from the beginning) and the last is the end point. A negative value means that the end point is calculated from the end of the array. In addition, if the number of data points has to be decreased the expression above can be extended to include the stride.

```
x[20:-300:2]
```

Consequently with this expression only every second data point is included. The operations shown here also need to be performed on the y values. This is done by typing the same expression into the text field for the y values and exchanging x against y. It is also possible to conduct simple arithmetic operations on the data. For example transforming the x data from degrees to Q_z (scattering vector in reciprocal Ångstrom), assuming that a wavelength of 1.54 Å is used:

³If no errors for the data are available set **e col=y col**

```
4*pi/1.54*sin(x*pi/180)
```

This would then be typed into the text field for the x data. The examples presented here are rather limited but hopefully it shows the flexibility of treating the data. For a more detailed list of functions and syntax the reader is referred to the tutorials and manuals found at www.scipy.org.

4 Making a model

As said before, the only mandatory thing the model file has to contain is a function called `Sim` taking a member of the class `Data` as input parameter. However, to make the model useful, functions for setting the values have to be incorporated. Since writing a model actually involves writing a script in Python it is good to have some basic knowledge of the syntax. However, if you have some basic knowledge about programming it should be fairly easy to just look at the examples and write your own models without having to learn to program in Python. On the other hand, there exists a number of *free* introductory books as well as tutorials on the internet for the interested reader, see below.

- Python's homepage contain most of the available tutorials online, www.python.org.
- *A Byte of Python* is an introductory text for the absolute beginner, www.byteofpython.info:8123.
- *How to Think Like a Computer Scientist: Learning with Python* is a textbook for the beginner written for computer science students, www.greenteapress.com/thinkpython.
- *Dive Into Python* is an introduction to Python book for the more experienced programmer, diveintopython.org.
- In addition there are a number of tutorials on the SciPy homepage, www.scipy.org, which deal with numerical computations. There is also a migration guide for those who are familiar with MatLab®

In order to write the `Sim` class it is necessary to know the structure of the class `Data` which is taken as a parameter. The variables which could be useful in the `Sim` function are:

- x** A list of 1-D arrays (vectors) containing the x-values of the processed data
- y** A list of 1-D arrays (vectors) containing the y-values of the processed data
- xraw** A list of 1-D arrays (vectors) containing the raw x-values (the data loaded from the data file)
- yraw** A list of 1-D arrays (vectors) containing the raw y-values (the data loaded from the data file)
- use** A list of booleans (True or False) denoting if the data should be fitted

Knowing what the `Data` class contains we will start with a simple example, making a model that fits one Gaussian to the first data set. The free parameters of the Gaussian are; the center of the peak, `Xc`, the peak width, `W`, and the amplitude of the peak, `A`. Writing a model for it would produce a code as shown below. Note that a `#` produce a comment.

```

1  # Create a class for user variables
2  MyVar=UserVars()
3  # Create your variables + set the initial values
4  MyVar.newVar('A',1.0)
5  MyVar.newVar('W',2.0)
6  MyVar.newVar('Xc',0.0)
7
8  # Define the function for a Gaussian
9  # i.e. definition of the model
10 def Gaussian(x):
11     return MyVar.A*exp((x-MyVar.Xc)**2/MyVar.W**2)
12
13 # Define the function Sim
14 def Sim(data):
15     # Calculate the Gaussian
16     I=Gauss(data.x[0])
17     # The returned value has to be a list
18     return [I]
19

```

The following is a brief description of the code above. First an object of the class `UserVars` is created. This object is used to store user defined variables. Then the variables are initialized (created) with their names given as strings. After that a function for calculating a Gaussian variable is created. The function takes an array of `x` values as input parameters and returns the calculated `y`-values. At last the `Sim` function is defined. The function `Gauss` is called to calculate the `y`-values with the `x`-data as the input argument. The `x`-values of the first data set are extracted as `data.x[0]`, and those of the second data set would be extracted by `data.x[1]`. Note that a list is returned by taking the array (vector) `I` and making a list with one element. Note that this requires that only one data set has been loaded. In order to fit the parameters created in by `MyVar` the user only has to right click on a cell in the grid of the Parameter Window and choose the `MyVar.set[Name]` function, i.e. `MyVar.setA`.

The code above is usually sufficient for prototyping and simple problems. For more complex models it is recommended to write a library. This is what has been done for the simulation of x-ray reflectivity data. Also, instead of writing a lot of functions for each model, a class, or several, can be written to make the model simple to use. As a more elaborate example the previous simple example can be transformed into a class:

```

1  # Definition of the class
2  class Gauss:
3      # A class for a Gaussian
4      # The creator of the class
5      def __init__(self,w=1.0,xc=0.0,A=1.0):

```

```

6         self.w=w
7         self.xc=xc
8         self.A=A
9
10        # The set functions used in the parameters column
11        def setW(w):
12            self.w=w
13
14        def setXc(xc):
15            self.xc=xc
16
17        def setA(A):
18            self.A=A
19
20        # The function to calculate the model (A Gaussian)
21        def Simulate(x):
22            return A*exp((x-self.xc)**2/self.w**2)
23
24        # Make a Gaussian:
25        Peak1=Gauss(w=2.0,xc=1.5,A=2.0)
26
27        def Sim(data):
28            # Calculate the Gaussian
29            I=Peak1.Simulate(data.x[0])
30            # The returned value has to be a list
31            return [I]
32

```

This code is quite similar to the first version with only functions. It starts with the definition of the class `Gauss`. This class has a constructor, `__init__`, to initialize the parameters of the object and functions to set the member variables, denoted as `self.*`. It also contains a member function to calculate a Gaussian with the member variables. After the class definition an object, `Peak1`, of the `Gauss` class is created. Then the `Sim` function is defined as in the previous example but with the function call exchanged to `Peak1.Simulate(data.x[0])` in order to simulate the object `Peak1`. The function names that should go into the parameter column in the parameter window will be: `Peak1.setW`, `Peak1.setXc` and `Peak1.setA`. Making the model based on a class makes it easier to extend. For example if two peaks should be fitted the class does not have to be changed. Instead an additional object of the class `Gauss`, for example called `Peak2`, can be created and the two contributions are then added in the `Sim` function. The code would then be modified to (omitting the class definition):

```

1  #Insert the class definition from above
2  # Make Gaussians:
3  Peak1=Gauss(w=2.0,xc=1.5,A=2.0)
4  Peak2=Gauss(w=2.0,xc=1.5,A=2.0)
5
6  def Sim(data):
7      # Calculate the Gaussian
8      I=Peak1.Simulate(data.x[0])+Peak2.Simulate(data.x[0])

```

```

9      # The returned value has to be a list
10     return [I]

```

Thus, for fitting the parameters for the second Gaussian the functions used should be `Peak2.setW`, `Peak2.setXc` and `Peak2.setA`. When the base class is created it can be extended with more problem oriented constraints by using functions as in the first example. For example, in some cases it might be known that the width of the two Gaussians should be the same. This can be solved by defining a new variable:

```

1  #Insert the class definition from above
2  # Make Gaussians:
3  Peak1=Gauss(w=2.0,xc=1.5,A=2.0)
4  Peak2=Gauss(w=2.0,xc=1.5,A=2.0)
5  # Create a class for user variables
6  MyVar=UserVars()
7  # Create your variables + set the initial values
8  MyVar.newVar('BothW',1.0)
9
10 def Sim(data):
11     Peak1.setW(MyVar.BothW)
12     Peak2.setW(MyVar.BothW)
13     # Calculate the Gaussian
14     I=Peak1.Simulate(data.x[0])+Peak2.Simulate(data.x[0])
15     # The returned value has to be a list
16     return [I]

```

Instead of using the `*.setW` functions the `MyVar.setBothW` can be used, which is automatically created by `MyVar` class. In summary it is recommended that the models implemented in libraries are defined as classes and that these are as general as possible with respect to the parameters. The specific parameter couplings can be included as functions in the model file. The methods shown with the examples in this section also apply to the libraries included for x-ray reflectivity. The classes are different but the general use is the same.

5 The figure of merit function

The figure of merit (FOM) function is the function that describes how good the model fits with the data. In the program the error functions can be defined as a function in the model file or by using a predefined FOM function in the file `errorfuncs.py`. All the error functions defined below are implemented in this file.

5.1 Examples of FOM functions

In most cases of fitting, for example a straight line, the method of least squares is usually used. This is a special case of the more general Chi-squared, χ^2 , method.

$$\chi^2 = \text{FOM}_{\text{Chi2Bars}} = \sum_i \left(\frac{M_i - S_i}{\sigma_i} \right)^2, \quad (1)$$

where M_i represents the measured value, S_i the simulated value and σ_i is the error. The index i represents the i 'th point. This should be considered as the starting point for any fitting problem. If error bars are not available for the data or if the data has large systematic errors one can consider the functions below instead. A useful error function for reflectivity and diffraction is the absolute logarithmic error function;

$$\text{FOM}_{\log} = \frac{1}{N-1} \sum_i |\log(M_i) - \log(S_i)|, \quad (2)$$

where N is the number of points. This normalization with the number of points is included in order to compare different data sets. Otherwise the error function would be dependent on the number of points to be fitted. A similar FOM function is the squared logarithmic

$$\text{FOM}_{\log} = \frac{1}{N-1} \sum_i (\log(M_i) - \log(S_i))^2, \quad (3)$$

which is more sensitive to outlying data points than the previous, due to that the error is squared. Thus it is usually not as useful as the absolute logarithmic error function. Another FOM function that has been found useful for high angle diffraction from superlattices is the absolute squareroot

$$\text{FOM}_{\text{sqrt}} = \frac{1}{N-1} \sum_i \left| \sqrt{M_i} - \sqrt{S_i} \right|, \quad (4)$$

This will put more emphasis on the points of high intensity than FOM_{\log} and can be more useful in those cases where the intensity range is not too large or where the features that should be fitted have high intensity. The last function that is implemented is a very empirical error function that has been used to fit multilayers. It consists of two terms, one being the absolute logarithmic FOM function and the second an absolute difference scaled with $\sin(\theta)^4$. The $\sin(\theta)^4$ term removes the overall decay of the reflected intensity as a function of angle. This puts more emphasis on the multilayer Bragg peaks than the features in between.

$$\text{FOM}_{\text{MyError}} = \frac{1}{N-1} \sum_i |M_i - S_i| \sin(\theta_i)^4 |\log M_i - \log S_i| \quad (5)$$

The default error function in the program is the absolute logarithmic error function. To change the error function used, append the following line to the model file, using the FOM function called `MyError`.

```
solver.setFomfunc(MyError)
```

For the names of the other error functions open the file `fomfuncs.py`. It is also possible to add a user defined FOM function in the model file. The syntax for writing an error function is

```
1 def NameFom(data,sim):
2     return 1/sum(data.use)*sum([expression for
3         (y,s,use) in zip(data.y,sim,data.use) if use])
```

The `for` loop is necessary since there can be more than one data set that is used in the fitting procedure. `expression` should be replaced with the expression for the new error function.

6 Error Bars on fitted parameters

The last column in the parameter window represents the error bars. In order to assign a confidence level to the errors the underlying distribution of the errors have to be known. This is for a general FOM not known in advance. However, for the Chi2Fom and Chi2BarsFom it is possible to assign a level of confidence to the errors (or rather calculate the errors given a level of confidence). These two FOM's assume the errors on the data are normally distributed. One other possibility to achieve quantitative error bars is to use Log2Fom which implies that the errors on the data have a lognormal distribution.

To calculate the errors on the refined parameters a level of the FOM which encompasses the level confidence has to be calculated. In the general case this is a 5% increase in the FOM. It should be noted that is value is arbitrary chosen and has no connection to any level of confidence! The procedure for a real confidence level will be explained below. When this level is set all evaluations of the FOM function from the fitting process are used and the min and max values of the parameter of interest that are lower than the set value represents the error. Naturally this interval can be asymmetric and this procedure does not make any assumption of the shape of the minimum of the FOM.

To calculate the errors of the parameters for a given level of confidence the increase in the Chi2 distribution, $\Delta\chi^2$, has to be known with the number of free degrees set to the number of free parameters in the fit. Given this number the level of the FOM function are calculated by

$$\text{FOM}_{\text{level}} = \text{FOM} + \frac{\Delta\chi^2}{N - 1} \quad (6)$$

This procedure relies that the errors on the data are known and available. If this is not the case the level can be calculated by assuming that the errors on the data are constant and that the fit is good i.e.

$$\Delta\chi^2 = N - M = \frac{\text{FOM}(N - 1)}{\sigma^2} \quad (7)$$

where M is the number of free parameters in the fit. In this case the level of the FOM function for a given level of confidence are given by.

$$\text{FOM}_{\text{level}} = \text{FOM} \left(1 + \frac{\Delta\chi^2}{N - M} \right) \quad (8)$$

Finally it should be noted that in order to use the above procedure to calculate the real confidence intervals the user has to first use the Chi2BarsFom for the first alternative and Chi2Fom or Log2Fom for the second alternative. Which procedure that are used are automatically chosen in the program. If there are another FOM function used the errors are represented by a certain increase (percent) in the FOM. Next the level of confidence has to be set in the settings-Conflevel dialog after the model has been loaded, otherwise the errors will be a 5% increase as default. The values in the error column are updated after the fit has ended alternatively stopped by the user.

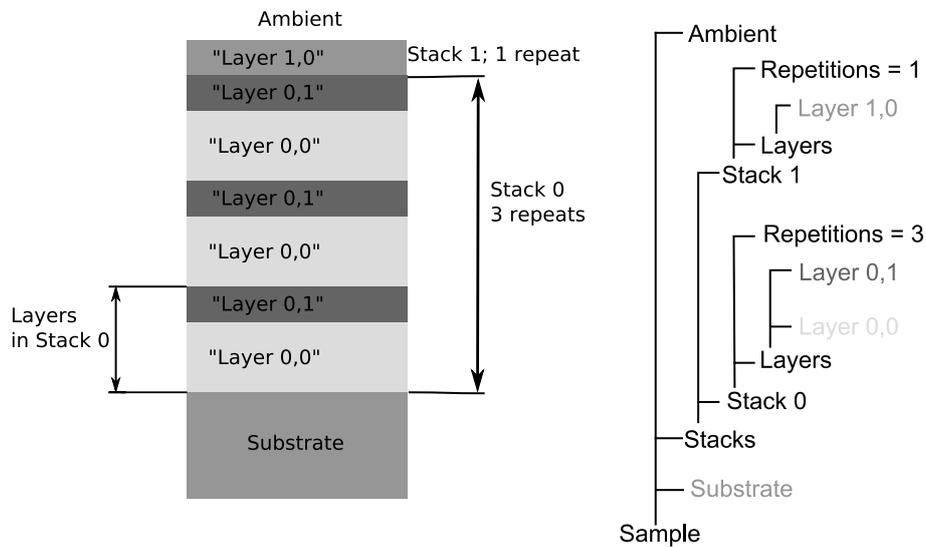


Figure 7: A picture of how the different classes builds a sample. The left figure shows the sample and the right shows the computer representation of the sample.

7 Libraries

This section will give an introduction to the model included in the program. Also some examples on how to write the model will be given. If the first section is unclear, read the section Making a Model again since the same syntax is used in that section. The examples are also less complex.

7.1 X-ray reflectivity

The libraries for x-ray reflectivity are built from the concept of three building blocks for a sample. These are: **Layer**, **Stack** and **Sample**. The **Layer** defines the parameters of a layer (e.g. refractive index, roughness). The **Stack** contains a number of **Layers** which can be repeated a number of times. This corresponds in some way to a multilayer. The **Sample** contains a number of stacks which builds up the sample. Also included is the ambient medium as well as the substrate. A picture of how the different classes make up a sample can be seen in figure 7. A class named **Instrument** is also included for various instrument parameters (e.g. wavelength).

7.1.1 Usage

In order to use a model library it should be imported into the model file by the `import` statement. The current model for x-ray reflectivity is called `ModelInterdiff`. As an example consider the sample consisting of a Fe/Pt multilayer with 25 repeats on top of a buffer layer that consists of a Fe layer closest to the substrate and then a thick Pt layer. The sample has been grown on MgO. In order to model this sample two different stacks are needed: one for the buffer layers and the second for the multilayer. The model file would look something like the following.

```

1  # Import the model classes
2  from ModelInterdiff import *
3
4  # get the refractive indexes for the materials
5  nFe=getn('Fe1',2/2.866**3,1.54)
6  nPt=getn('Pt1',4/3.924**3,1.54)
7  nMgO=getn('Mg101',4*2/4.2**3,1.54)
8
9  # Create all the layers needed
10 MLFe=Layer(n=nFe,d=13.5,sigmar=1.0,reldens=1.0)
11 MLPt=Layer(n=nPt,d=13.5,sigmar=1.0,reldens=1.0)
12 bufPt=Layer(n=nPt,d=40,sigmar=1.0,reldens=1.0)
13 bufFe=Layer(n=nFe,d=5,sigmar=1.0,reldens=1.0)
14 sub=Layer(n=nMgO,sigmar=5,reldens=1.0)
15
16 # Create the Stacks - consists of layers
17 ML=Stack(Layers=[MLFe,MLPt],Repetitions=25)
18 buf=Stack(Layers=[bufFe,bufPt],Repetitions=1)
19
20 # Create the sample
21 sample=Sample(Stacks=[buf,ML],Ambient=Layer(n=1.0),Substrate=sub)
22 # Create the instrument -
23 # Coordinates=1=> two theta Coordinetes =0 => Q_Z
24 inst=Instrument(Wavelength=1.54,Coordinates=1,I0=1.0e6)
25
26 # Simulate function - MANDATORY
27 def Sim(data):
28     I=sample.SimSpecular(data.x[0],inst)
29     return [I]

```

First the script imports the library `ModelInterdiff`. This loads the underlying model. Next the different refractive indices for the materials are extracted from the scattering length tables[3]. The function `getn` takes the atomic composition of the elements as a string as first argument, the second argument is the atomic density in atoms/Å³ and the the last argument is the x-ray wavelength in Å. In the next code block all the layers needed are created. The constructor for the `Layer` class takes the following parameters as input:

1. `n`, the refractive index of the material.
2. `d`, the thickness of the layer in Å.
3. `sigmar`, the roughness of the *upper* interface in Å.
4. `reldens`, the relative density of the material, where 1.0 corresponds to the (bulk)density that `n` refers to.

After the layers have been created the different stacks are created, one for the multilayer and one for the buffer layers. The constructor for a `Stack` takes the layers that are contained in that stack as a list and the number of repeats of that stack. Note that the first layer in the list is the layer closest to the substrate. When the stacks have been created the sample can be constructed

Parameter	Description
n	A complex number for the refractive index of the layer. It should be on the form $n = 1 - \delta + \beta j$
d	The thickness of the layer given in Å.
sigmar	The root mean square (rms) value of the roughness given in Å.
sigmai	The root mean square (rms) value of the interdiffusion given in Å. Note that the roughness used for specular calculations is $\sigma = \sqrt{\sigma_i^2 + \sigma_r^2}$.
reldens	The relative density of the material. Used as a scaling factor for δ and β in the refractive index.

Table 1: The different parameters for the class Layer.

Parameter	Description
Layers	A <i>list</i> of the different Layers in the Stack. Thus the input should be on the form Layers=[layer1,layer2]
Repetitions	The number of times the stack repeats itself. Should be a integer.

Table 2: The different parameters for the class Stack.

from these stacks. The constructor for a **Sample** takes three input parameters: a list of the Stacks, the Ambient layer (what is on top of the multilayer, usually air, $n = 1$) and the substrate layer. For the ambient layer the calculations ignore the roughness and the thickness. For the substrate the calculations ignore the thickness. The first element in the **Stacks** list is the one closest to the substrate.

Next an **Instrument** object is created. This contains the wavelength used and in which coordinates the data was recorded; 1 corresponds to 2θ and 0 corresponds to Q_z . In addition it contains the incident intensity **I0**. Finally the **Sim** function is created. The function **sample.SimSpecular** calculates the specular reflectivity from the sample given the x-data and the instrument object.

In order to use the model file for fitting all the functions for setting the parameters have to be known. These functions are named according to the following rule: **Objectname.set[parameter name first letter uppercase]**. For example for fitting all the parameters for the object **MLFe** the functions would become: **MLFe.setD**, **MLFe.setN**, **MLFe.setSigmar** and **MLFe.setReldens**. Note that it not recommended to fit **n** since this is a complex number and the fitting program only works with real numbers.

The model **ModelInterdiff** also contains a function to calculate the diffuse reflectivity from a multilayer stack, **[sample].SimOffSpecular(TwoThetaQz,ThetaQx,inst)**. To simulate diffuse reflectivity some more parameters has to defined for the sample, **eta_x** the in-plane correlation length in Å, **eta_z** the out-of-plane correlation length between the interfaces and **h** the jaggedness parameter. Note that these parameters are assumed to be constant throughout the sample. In addition, the amount of interdiffusion of each layer can be defined as **sigmai**.

Parameter	Description
Stacks	A <i>list</i> of the different Stacks in the Sample. Thus the input should be on the form Stacks=[stack1,stack2] .
Ambient	The Ambient layer, i.e. material that is on top of the sample. Should be a member of the class Layer .
Substrate	The Substrate layer, i.e. material that is below the sample structure. Should be a member of the class Layer .
h	The jaggedness parameter. Only used for diffuse calculations. Should be in the interval [0.15, 1.0].
eta_z	The out-of-plane correlation length given in Å.
eta_x	The in-plane correlation length given in Å.

Table 3: The different parameters for the class Stack.

Parameter	Description
Wavelength	The wavelength of the radiation used given in Å.
Coordinates	The coordinates which the data are in. Have to be an integer. 1 corresponds to reciprocal coordinates in Å ⁻¹ and 0 to angular coordinates, i.e. the scattering angle, 2θ in degrees.
I0	The incident intensity. A scaling factor for the reflectivity.
Restype	An integer determining which type of resolution convolution to do. 0 no resolution convolution. 1 fast convolution assuming equally spaced data points with a spacing much smaller than the resolution and gaussian distribution. 2 "Full convolution" calculates Respoints data extra and convolutes these with a gaussian.
Res	The resolution of the instrument. Given as an rms width of the data coordinates.
Respoints	The number of points used for the resolution convolution. Only valid for Restype=2
Resinrange	How many standard deviations given by Res is included in the convolution.
Footype	an integer determining which footprint correction to use. 0: no footprint correction. 1: footprint correction for a gaussian beamprofile. 2: footprint correction for a square beamprofile.
Beaw	the width of the beam at the sample position given in mm. For Footype= 1 it is the rms width of the beam. For Footype= 2 it is the FWHM of the beam. Used for footprint corrections.
Samlen	The sample length given in mm. Used for footprint corrections.

Table 4: The different parameters for the class Instrument.

7.1.2 Description

This section will give an introduction to the different files and functions for the classes used for simulating the reflectivity. This could become a bit confusing for the reader not familiar with programming. It is mainly intended for people wanting to extend the models while keeping the general interface the same.

The underlying calculation routine is implemented in the file `Paratt.py` where the function `Ref1` is implemented. This function takes the refractive indices, thicknesses and roughnesses of the layers as 1-D arrays. Thus it does not make use of the classes defined in the previous section. In order to make the implementation of a new model as easy as possible the classes are created dynamically with the `MakeClasses` function in the file `Ref1.py`. This function takes a dictionary that contains the parameter names and their default values as input parameters. The function then returns the classes. This function allows easy creation of the interface of any model based on the Layers-Stacks-Sample concept. As an example how to tie these two libraries together see the file `ModelInterdiff.py`. This file defines the model and extracts the parameters from the classes and sends them to the `Ref1` function for calculation. The definition of the classes in `MakeClasses` utilizes metaclasses to create all the parameter functions and the parameters dynamically. The process of creating the classes provides a very flexible framework for creating new models.

7.2 Neutron reflectivity

The included model for neutron reflectivity is very similar to the one for x-ray reflectivity. Neutrons interact not only with the nucleus but also with the spin of the unpaired electrons. Thus, the calculations has to also include magnetic scattering. The model for Neutron reflectivity is called `ModelSpecNX`. The models are very similar to the x-ray models discussed in the previous section. The differences are: different parameters in the `Layer` class, see table 5 and one new parameter in the `Instrument` class, `Sim` which defines the type of simulation according to the following scheme

Sim=0 Calculates the reflectivity for x-rays. The `SimSpecular` function gives one output.

Sim=1 Calculates the reflectivity for neutrons without magnetic contrast. The `SimSpecular` function gives one output.

Sim=2 Calculates the non spin flip neutron reflectivity. `SimSpecular` gives two outputs, as a tuple, corresponding to the up-up and down-down reflectivity.

Sim=3 Calculates the spin-flip reflectivity. `SimSpecular` gives three outputs, as a tuple, corresponding to the up-up, down-down and the spin flip channel.

8 The fitting algorithm

A general introduction to differential evolution can be found at [6]. For the specific case of the application of the algorithm to x-ray reflectivity and diffraction

Parameter	Description
fb	The scattering length per atom of the material. Given as per Å for neutrons and per electron (Thomson scattering length) for x-rays.
d	The thickness of the layer given in Å.
sigma	The root mean square (rms) value of the roughness/interdiffusion given in Å.
dens	The atomic density given in atoms/Å ³
magn	The magnetic moment given in μ_B /atom.
magn_ang	The angle of the magnetic moment relative to the applied field (spin direction of the neutron).

Table 5: The different parameters for the class Layer for the model for Neutrons.

the reader is referred to the paper of Wormington *et. al.*[7].

This section will briefly deal with the parameters the user can change to tune the algorithm. However, this should not be necessary for most problems, the algorithm usually works as it is. The class that implements the differential evolution algorithm is called `Solver` and can be found in the file `DEWorm.py`. In the program an instance of this class is created and called `solver`. The parameters that could be useful to change is:

km The mutation constant. The default value is 0.7. A reasonable value is usually between 0.5 and 1. A lower value yields a faster convergence but also a higher probability of misconvergence. Example: `solver.setKm(0.7)`

kr The crossover constant. Default value is 0.5. Determines the probability that a parameter in the present vector is exchanged. Should be between 0 and 1. Example: `solver.setKr(0.5)`

Popmult A multiplicative factor that determines the size of the population. The population size is determined by `Popmult`×The number of free parameters. The default value is 3. Example: `solver.setPopmult(3)`

maxGenMult A multiplicative factor that determines the maximum number of generations/iterations. The maximum number of generations is determined by the number of free parameters×population size×`maxGenMult`. The default value is 1. Example: `solver.setMaxgenmult(1)`

FOMFunc The function that calculates the error. See the section 5 for more details. Example: `solver.setFomfunc(Chi2Fom)`

Note that the initial parameter settings for fitting algorithm are optimized for many free parameters. Fitting only one parameter will not work properly. The minimum number of free parameters for these settings are about 3-4 free parameters in order to get reasonable results.

9 A short HowTo for x-ray reflectivity

1. Get the x-ray data, do not use too fine steps since this will increase the computation time (and not too large). Check if the sample is bent or

“wavy”. If that is the case try to decrease the resolution. In case of reflectivity data check so that the diffuse scattering does not contribute to the specular. If so subtract it.

2. Write a model file, using your knowledge of the system. Make the `Sim` function. Couple parameters to make them more independent. For example, use the bilayer repetition length as a free parameter for multilayers. Do not forget to reload the model file when it has changed.
3. When the model file works include the instrument resolution (convolution with a Gaussian is often good)[1, 4, 5, 2]. If needed, include the footprint correction[2].
4. Remember to choose an error function that makes sense for your problem. This is an important point.
5. Cut out the region that does not contain any information, mainly for speeding up the calculation.
6. Define *all* free variables in the grid/spreadsheet
7. Type in reasonable minimum and maximum values, be generous with the bounds.
8. Set a start guess. A rough guess is enough.
9. Start fitting. Keep an eye on the parameter distribution; if the population spread (blue bars) are very small there will be very little improvement.
10. If the best value converges to the min or max boundaries increase the min and max values. Restart the fit.
11. If the fit is not good enough there are three possibilities;
 - (a) Check the data. Make sure that the sample is not for example bent (varying width of the specular component). This is best checked at a couple of angles close to the total reflection and also far away.
 - (b) The model is wrong - change it to a better one.
 - (c) The error function does not give a good representation of the features you want to fit
12. Fix the problem mentioned previously and restart the fitting process.
13. Continue until the fit is good (Repeat 9-12)

References

- [1] W. H. de Jeu, J. D. Shindler, and E. A. Mol. *J. Appl. Cryst.*, 29:511–515, 1996.
- [2] A. Gibaud, G. Vignaud, and S. K. Shina. *Acta Cryst.*, A49:642–648, 1993.
- [3] B.L. Henke, E.M. Gullikson, and J.C. Davis. *Atomic Data and Nuclear Data Tables*, 54:181–342, 1993. The data was obtained from <http://www-cxro.lbl.gov/optical.constants/>.

- [4] D. Sentenac, A. N. Shalaginov, A. Fera, and W. H. de Jeu. *J. Appl. Cryst.*, 33:130–136, 2000.
- [5] J. D. Shindler and R. M. Suter. *Rev. Sci. Instrum.*, 63:5343–5347, 1992.
- [6] Rainer Storn. <http://www.icsi.berkeley.edu/~storn/code.html>.
- [7] M. Wormington, C. Panaccione, K. M. Matney, and D. K. Bowen. *Phil. Trans. R. Soc. Lond. A*, 357:2827–2848, 1999.