User Manual of FitSuite 2.3.0

Szilárd Sajti

March 9, 2026

Contents

1 Introduction, antecedents **1**

2 Basic concepts of FitSuite **2**
2.1 Experimental scheme and its structure . . . 2
2.2 Transformation matrix technique . . . 3
2.3 Parameter distribution . . . 6
2.4 Statistics . . . 7
2.5 Uncertainty estimation, bootstrap method . . . 11
2.6 Fitting . . . 14
2.6.1 Constraints (Simple bounds) . . . 15
2.6.2 Distributed parameters (using maximum entropy principle) 18
2.6.3 Rescaling parameters . . . 19
2.6.4 Parameter resolution, minimum step width . . . 19
2.6.5 Numerical derivatives . . . 20
2.7 Subspectrum . . . 21

3 Working with FitSuite **22**
3.1 Starting a new project . . . 22
3.2 Building up the model structure . . . 23
3.2.1 Adding objects, models using xml like files . . . 24
3.2.2 Copying and inserting objects . . . 31
3.3 Adding data . . . 31
3.4 Changing parameters, matrices . . . 36
3.5 Parameter filtering . . . 38
3.5.1 Single word arguments, wildcards . . . 39
3.5.2 Complex arguments, logical operators . . . 40
3.5.3 Combination of commands using logical operators . . . 40
3.5.4 List of filter commands . . . 41
3.5.5 List of filter commands with optional arguments . . . 43
3.5.6 Complex examples . . . 45
3.6 Command-line interface . . . 45
3.7 The ‘math’ command . . . 55
3.7.1 Mathematical functions in math command . . . 58
3.8 Report generator . . . 63
3.9 Cloning . . . 63
3.10 Merging projects . . . 65
3.11 Model groups . . . 65
3.12 Simulation, Fit . . . 66
3.12.1 Powell‘s method . . . 66

3.12.2 Nelder – Mead method . . . 68
3.12.3 Polak – Ribiere and Fletcher – Reeves . . . 69
3.12.4 Broyden – Fletcher – Goldfarb – Shanno . . . 69
3.12.5 Levenberg – Marquardt . . . 70
3.12.6 Levenberg – Marquardt method (LMDER) from the MINPACK . . . 71
3.12.7 Genetic algorithms . . . 71
3.12.8 After fit . . . 78
3.13 Uncertainty calculation . . . 79
3.14 Calculating statistics . . . 79
3.15 Plotting . . . 79
3.16 Sounds . . . 80
3.17 Examples . . . 80

4 Sources, documentation **82**

5 'Installation' **82**
5.1 Linux . . . 83
5.2 Windows . . . 84

6 License **84**

References **85**

Glossary **87**

Index **91**

1 Introduction, antecedents

FitSuite is an environment for simultaneous fitting and/or simulation of experimental data of vector-scalar type, such as parametric curves and surfaces typically collected in a physics experiment. *Simultaneous* here means that several sets of the same type of experiment and even of different types of experiments can be simulated/fitted in a self-consistent, statistically correct framework with provisions for cross-correlation of the theoretical and specimen parameters.

Experiments often provide raw data of measurements performed on the same sample by different methods, and/or using different experimental conditions, like temperature, pressure, magnetic field, and the like. Data often partly depend on the same set of experimental and sample parameters, therefore a simultaneous evaluation of all experimental data is a prerequisite. However, data evaluation programs are dominantly organized around a single method, therefore a simultaneous access to the data for a common fitting algorithm is not typical. Lacking suitable programs, some parameters are determined from one measurement, assumed uncertainty-free and kept constant when evaluating other experiments, an obviously incorrect approach. Besides, for different methods different programs are used, which makes it very difficult to tune parameters of such theories and their uncertainties and correlations to each other and to extend or modify the theories to describe different experimental data.

Therefore, starting in 2004 (as part of the Dynasync FP6 project sponsored by the *European Commission*, and since 2006, within the **NAP_VENEUS** project, sponsored by the *Hungarian National Office for Research and Technology*) we developed FitSuite for *Windows* (tested under XP and 7) and *Linux*, a code that consistently handles by now data of over ten spectroscopic methods with over twenty theories together with a large number of sample structures in a common inter-related framework.

FitSuite is an environment, in which, besides the possibility of adding brand new (user written) 'theories', the user can 'build' new theories based on the combination of existing ones, subroutines, which call each other. To our opinion, this feature of FitSuite is really essential since a complex physical system can in general be divided into subsystems and even this subsystem can be divided into further subsystems, groups of which can be described with the same parameters and physical equations. To provide an example, assume we have a thin film built up from layers and the layers may be built up from further objects (depending on to what physical characteristics the method in question is sensitive to) such as domains, atomic groups, lattice sites, molecules, etc. (E.g. in Mössbauer-related problems each layer may contain several sites, with their own hyperfine 'theories' to calculate the corresponding subspectra).

The original idea of cross-correlation and of a hierarchy of theories as well

as a number of subroutines of FitSuite were inherited from EFFI (**E**nvironment **F**or **FI**tting) [1], an originally Mössbauer spectroscopy-related *Fortran* program developed over the years by Prof. Hartmut Spiering from *University of Mainz*. In view of the friendly and fruitful collaboration between the Budapest and Mainz groups over the last decades, Prof. Spiering kindly agreed to build in the theories written by him and tested within EFFI into FitSuite in order to promote both projects. Enlightening discussions with him greatly promoted the FitSuite project. Although in the last three years the two projects developed in different directions, we are very grateful for Prof. Spiering's essential contribution to FitSuite.

In the following, we go through the basic concepts used by FitSuite first. Thereafter, we give a short description of the GUI (**G**raphical **U**ser **I**nterface), how FitSuite should be used from start to fit and present shortly some examples which can be found in the directory *examples*.

2 Basic concepts of FitSuite

In this section, we try to make the reader acquainted with the basic concepts used in the program, which are necessary to know, in order to be able to use it. Here, we summarize the principles. The description of the user interface is given in the next section, there we will see, how these concepts, principles are used in practice.

To be able to simulate (fit), we need to give the program our knowledge of the problems. In FitSuite we should create simultaneous fit projects first, (which have file extension *.sfp), which contain the **fitting problems** consisted of **experimental data** and of **experimental scheme**.

2.1 Experimental scheme and its structure

The experimental scheme contains the information necessary for the description of the system consisted of the experimental apparatus(es), about the experimental method itself and of the system under study (e.g: a measured sample). We know that the meaning of the word *experimental scheme* is a bit different from the present usage, as it usually contains only the apparatuses, but we did not find a better one. In our thoughts, we usually separate the apparatus used for observation and the subject of observation. Here we do not want to do this, as it would lead a more complex structure, and the experimental scheme selects the characteristic properties, features of the studied system (and of the apparatus), which are essential to be able to calculate the theoretical pair of the experimental data set, which is prerequisite for fitting, for checking theory or measurement, etc.

In FitSuite the **experimental scheme** is built up of **objects**, which correspond to physical objects (or concepts) e.g.: source, sample, detector, layer, site, etc. The

experimental scheme is also an object. It has a simple tree structure with one root object, the experimental scheme. To these objects belong **properties** which represent the physical quantities (thickness, roughness, hyperfine field, susceptibility, electric field gradient, effective thickness, ...) and some numbers characterizing the experimental scheme e.g. number of channels, symmetries of the sites, etc.

Besides objects and properties, to each object may belong algorithms for calculation of the characteristic spectra. (In the current built-in problems only the experimental schemes and the sites have algorithms.) These type of objects are called **model** on the program interface, as it is an almost perfect name for them.

This type of structure is needed, because simulating our problem, writing the simulation algorithms, this structure mirrors the real physical system and therefore it is a practical, logical choice.

2.2 Transformation matrix technique

In contrast to the last remark of the previous section, the 'optimization' methods used for fitting require a parameter vector and not an object tree structure with properties. Furthermore, in case of simultaneous fitting we usually have the results of experiments performed in a bit different environment (external field, temperature, etc.) and/or different type of experiments using the same 'sample'. Therefore, there are a lot of common parameters. To eliminate this type of redundancy and as it is also convenient for the user to use as few parameters as possible (as it is more transparent for human and easier to fit in a parameter space with lower dimension at least if we want to get correct results) transformation matrix technique is used [2]. For this, we need also parameter vector (array). Because of these considerations we have to generate the parameter vector and the initial transformation matrix from the object tree structure mirroring the real physical system. The model parameters which still contain all the redundancy can be collected in a vector $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n})$, where $\mathbf{p_i}$ is the vector containing all the parameters belonging to the i-th model in the current simultaneous fit project. Let denote the vector of the fitting (or if you like simulation) parameters with $\mathbf{P}$ and the transformation matrix with $\underline{\underline{\mathsf{T}}}$. The transformation matrix technique uses the expression $\mathbf{p} = \underline{\underline{\mathsf{T}}}\mathbf{P}$, where $\dim \mathbf{P} \leq \dim \mathbf{p}$. Above, it was mentioned that this technique is used in order to eliminate the redundancy arising because of the common parameters, but this is not the unique reason. We can take into account some possible linear relations between the parameters also, which is a redundancy too.

It is advisable to take into account that there are parameters which according to expectations will not have interdependencies and therefore the $\underline{\underline{\mathsf{T}}}$ matrix can be 'block diagonalized'. It is more transparent to handle submatrices with lower dimensions, than one extended sparse matrix. Therefore we have to categorize the parameters according to our expectation whether the subspace stretched by a sub-

set of them may have interdependencies or this is very unlikely. (If the user finds a case, where our expectations are not met, (s)he is able to unite or split the submatrices, thus our choice here is not a constraint.) The initial submatrices generally are identity matrices, but there are exceptions. E.g. the thickness of a multilayer sample will be the sum of the layer thicknesses; in Mössbauer spectroscopy in a doublet site, the line positions and the measure of the splitting and the isomer shift will not be independent, etc.

Before going further, we have to mention a few concepts related to this transformation matrix technique used for simultaneous fitting (and simulation). In order to eliminate redundancy arising because common parameters, we often use the operation

$$\underset{u\in\{j_1,j_2,\ldots,j_s\}}{\mathrm{Corr}} : M_{n\times m} \longrightarrow M_{n\times(m-s+1)} \qquad (1 \leqslant j_i \leqslant m)$$

(where $M_{n\times m}$ denotes n by m matrices) defined by

$$\underset{u\in\{j_1,j_2,\ldots,j_s\}}{\mathrm{Corr}} T = \begin{pmatrix} T_{11} & \ldots & T_{1,j_k-1} & T_{1,j_k+1} & \ldots & T_{1,u-1} & \sum_{r=1}^{s} T_{1j_r} & T_{1,u+1} & \ldots \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\ T_{i1} & \ldots & T_{i,j_k-1} & T_{i,j_k+1} & \ldots & T_{i,u-1} & \sum_{r=1}^{s} T_{ij_r} & T_{i,u+1} & \ldots \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\ T_{n1} & \ldots & T_{n,j_k-1} & T_{n,j_k+1} & \ldots & T_{n,u-1} & \sum_{r=1}^{s} T_{nj_r} & T_{n,u+1} & \ldots \end{pmatrix} \tag{1}$$

which *correlates* the parameters belonging to columns $j_1, \ldots j_s$ except of the u-th column, which becomes the sum of the s correlated columns. The inverse of this operation is the *decorrelation* of parameters. The latter generally is not unequivocal, of course, therefore user interaction may be needed thereafter, in order to set the proper fit/simulation parameter values and transformation matrix elements. We may also split a matrix

$$\underset{\substack{\{i_1,i_2,\ldots,i_k\}\\ \{j_1,j_2,\ldots,j_s\}}}{\mathrm{Split}} : M_{n\times m} \longrightarrow \left\{M_{(n-k)\times(m-s)}, M_{k\times s}\right\} \qquad \begin{pmatrix} 1 \leqslant i_u \leqslant n \\ 1 \leqslant j_v \leqslant m \end{pmatrix}$$

E.g. in case of:

$$\underset{\substack{\{1,4\}\\\{2,3,6\}}}{\text{Split}} \left(\begin{array}{c|cc|cc|c|c} t_{11} & \mathcal{T}_{12} & \mathcal{T}_{13} & t_{14} & t_{15} & \mathcal{T}_{16} & t_{17} \\ \hline T_{21} & t_{22} & t_{23} & T_{24} & T_{25} & t_{26} & T_{27} \\ T_{31} & t_{32} & t_{33} & T_{34} & T_{35} & t_{36} & T_{37} \\ \hline t_{41} & \mathcal{T}_{42} & \mathcal{T}_{43} & t_{44} & t_{45} & \mathcal{T}_{46} & t_{47} \\ \hline T_{51} & t_{52} & t_{53} & T_{54} & T_{55} & t_{56} & T_{57} \end{array}\right) = \left\{ \left(\begin{array}{cccc} T_{21} & T_{24} & T_{25} & T_{27} \\ T_{31} & T_{34} & T_{35} & T_{37} \\ T_{51} & T_{54} & T_{55} & T_{57} \end{array}\right), \left(\begin{array}{ccc} \mathcal{T}_{12} & \mathcal{T}_{13} & \mathcal{T}_{16} \\ \mathcal{T}_{42} & \mathcal{T}_{43} & \mathcal{T}_{46} \end{array}\right) \right\}, \tag{2}$$

where the elements denoted by $\mathcal{T}$ will correspond to the matrix $M_{k\times s}$ and the elements denoted by T to the matrix $M_{(n-k)\times(m-s)}$. The elements denoted by t will be eliminated, therefore information will be lost, if they were not 0. Unification of two matrices can be conceived as the reverse of splitting, but there we set the cross-elements denoted by t to 0.

Sometimes it is useful to insert a new simulation/fit parameter, this corresponds to insertion of a new column in the transformation matrix. E.g. we know the value of the sum of some parameters, but we do not know their value. In such a case we may insert a new parameter, which we keep constant, set it to the value of the known sum, and set the corresponding matrix elements properly, like here:

$$\left(\begin{array}{ccccc} 1 & -1 & -1 & \cdots & -1 \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{array}\right) \left(\begin{array}{c} P_{\text{sum}} \\ P_2 \\ P_3 \\ \vdots \\ P_n \end{array}\right) = \left(\begin{array}{c} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_n \end{array}\right). \tag{3}$$

Thus we add a constraint for the corresponding parameters and we can eliminate a redundant fitting parameter and we do not increase the number of fitting parameters, as we would using Lagrange multipliers.

Some parameters are integer numbers (e.g. channel numbers, switches). These are never fitted, but the transformation matrix technique is useful for them, especially if some of them have the same value. The integer and real number based parameters are separated in the program and on the user interface too, to avoid the possibility of rounding errors, because of the finite precision of the computer representation of the numbers. This does not mean, that in some cases it would not be reasonable to mix the number types, but it is safer, than what we could gain allowing mixing. Everything we said about the transformation matrix for real parameters is valid for integer parameters, except of the fact, that in this case we have integer matrix elements, just because of the above mentioned considerations.

2.3 Parameter distribution

In physics, we often have not a single well defined physical object, but rather an ensemble of them. Even in this case, it may be useful to represent them with a single object in our (computer) model, but we have to know that which parameters are the same for the members of the ensemble and which are different. We can group the objects according to these parameters. The parameter distribution $f_d(\mathbf{p}^d)$ will give the probability that the ensemble has objects, which can be differentiated from other members according to the parameter set $\left\{p_k^d\right\}$ (which is part of the set of all the parameters $\{p_i\}$). As usually this distribution is unknown, we have to determine it by fitting too. There is no sense in defining the distributions on the level of model parameters, as it would be to complex and would require an enormous administration, therefore we will have $\mathbf{P}^d$, from which $\mathbf{p}^d$ can be calculated using the transformation matrix and $f_d(\mathbf{p}^d = \underline{\underline{\mathrm{T}}}(\mathbf{P}, \mathbf{P}^d)) = F_d(\mathbf{P}, \mathbf{P}^d)$. We usually do not know the analytical shape (e.g. Lorentzian, Gaussian) of the distribution either, and even if we know it, in general case, it may not be easy to use it for our calculations. Practically, therefore, we fit histograms with finite resolution and finite range, which is divided up equidistantly around the midrange. E.g. for a single distributed parameter $\mathcal{P}$ with resolution $\mathcal{N}$, range $\mathcal{R}$ and midrange $\mathcal{P}^0$ we will have histogram values h_j for the parameter values:

$$\mathcal{P}_j = \mathcal{P}^0 + \mathcal{R}\left(\frac{j}{\mathcal{N}} - \frac{1}{2}\right), \qquad (j = 0, \ldots, \mathcal{N} - 1). \tag{4}$$

If we have n distributed parameters and they are not independent, we will have a common distribution, which may be handled similarly, but we will have $h_{j_1,\ldots,j_n}$ histogram elements for the parameters $\mathbf{P}^d_{j_1,\ldots,j_n} = (\mathcal{P}_{1,j_1}, \ldots, \mathcal{P}_{n,j_n})$, where the components are given as:

$$\mathcal{P}_{i,j_i} = \mathcal{P}^0_i + \mathcal{R}_i\left(\frac{j_i}{\mathcal{N}_i} - \frac{1}{2}\right), \qquad (j_i = 0, \ldots, \mathcal{N}_i - 1), \quad (i = 1, \ldots, n). \tag{5}$$

In general case having distributed parameters we may fit $\{\mathcal{R}_i\}$, $\{\mathcal{P}^0_i\}$ and the histogram, i.e. the set $\{h_{j_1,\ldots,j_n}\}$. These can be handled as additional fitting parameters. In order to have an appropriate result we have to take into account additional constraints for the histogram. It is obvious, that we may assume that $\mathcal{R}_i \geqslant 0$, $h_{j_1,\ldots,j_n} \geqslant 0$ and $\sum h_{j_1,\ldots,j_n} = 1$. There are several approaches applying further constraints in order to get appropriate results for the parameter distributions [3–7]. One of them, which is also used in FitSuite currently, is the maximum entropy principle [7]. The entropy from information theory is defined as

$$S = \sum_{h_i > 0} -h_i \ln h_i. \tag{6}$$

We try to fit our parameters with the constraint, that S should assume its maximum. For further details we will return to this topic in subsection 2.6.2.

Another nontrivial question is how the objects corresponding to histogram elements contribute to the resulting spectrum, how they should be taken into account. The most simple approach (which currently is also used by FitSuite) is based on the assumption, that the resulting (sub)spectrum $\mathbf{y}_{\text{res}}$ belonging to the lowest rank submodel which still contains the physical object (objects if the distributed parameter is a correlated one) can be obtained as:

$$\mathbf{y}_{\text{res}} = \sum_{j_1,\ldots,j_n} h_{j_1,\ldots,j_n} \mathbf{y}\left(\mathbf{p}^d_{j_1,\ldots,j_n}, \mathbf{p}, \mathbf{x}\right). \tag{7}$$

Of course, other expression can be conceived and put into the program if some physical reason can be attributed to it. $\mathbf{y}_{\text{res}}$ may be and is used as an intermediate result if it belongs to a submodel (a part of a model, which itself is a model too).

The number of histogram elements will be $\prod_i \mathcal{N}_i$, therefore it is not too advisable to increase the number of distributed parameters too much. As fitting too much parameter is always a danger, but fitting too less may also be dangerous in case of a complex distribution, where the spectrum depends on the distributed parameter strongly. If some of them are independent we may gain a lot as e.g. for independently distributed parameters we will have only $\sum_i \mathcal{N}_i$ additional parameters to fit, because of the histogram elements. Before showing, how the maximum entropy principle is used, we will see the minimized functions during fitting in absence of distributed parameters.

2.4 Statistics

We usually mean by *fitting parameters* finding the parameters, for which the classical χ^2 statistic is minimal. This function is given as

$$\chi^2(\mathbf{p}) = \sum_i \frac{(y_i^{\text{exp}} - y^{\text{theo}}(\mathbf{x}_i, \mathbf{p}))^2}{\sigma_i^2}, \tag{8}$$

where y_i^{exp} and y_i^{theo} are the experimental and theoretical values for the i-th data point, and σ_i^2 is the standard deviation (uncertainty) of measurement of i-th data point, $\mathbf{x}_i$ is the independent variable (it may be a vector). Fitting simultaneously we minimize the (weighted) sum of the χ^2-s of the different fitting problems. The χ^2 is not the single statistic, which may be used for this purpose (e.g. see absolute deviation on wikipedia[1]). Furthermore its use is justified only for normally

[1] https://en.wikipedia.org/wiki/Least absolute deviation

(Gauss) distributed experimental data. In the problems handled by FitSuite currently, we have data obtained by particle counters, which have usually Poisson distribution. In case of large count rates, there is no problem, the Poisson distribution can be approximated well with Gaussian distributions, but in other cases we have to use other statistics in order to fit parameters. Several approaches exist to tackle this problem [8]. There are approximations based on (8) and on the fact, that the variance and the expectation value of the Poisson distribution is the same. These are of the form of classical χ^2, namely Pearson's χ^2

$$\chi^2_{\text{Pearson's}}(\mathbf{p}) = \sum_i \frac{(y_i^{\text{exp}} - y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p}))^2}{y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}, \tag{9}$$

modified Neyman's χ^2

$$\chi^2_{\text{Neyman's}}(\mathbf{p}) = \sum_i \frac{(y_i^{\text{exp}} - y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p}))^2}{\max(y_i^{\text{exp}}, 1)}. \tag{10}$$

(In 'unmodified' Neymann's χ^2 statistic, y_i^{exp} appears instead of $\max(y_i^{\text{exp}}, 1)$.) Furthermore there are other statistics based on ***M**aximum **L**ikelihood **E**stimation*, namely Poisson MLE

$$\chi^2_{\text{PMLE}} = 2\left[\sum_i \left(y^{\text{theo}}(\mathbf{x}_i, \mathbf{p}) - y_i^{\text{exp}}\right) - \sum_{y_i^{\text{exp}} \neq 0} y_i^{\text{exp}} \ln \frac{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{y_i^{\text{exp}}}\right], \tag{11}$$

obtained by using MLE for Poisson distributed data. And Gaussian MLE

$$\chi^2_{\text{GMLE}} = \sum_i \left[\frac{\left(y_i^{\text{exp}} - y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})\right)^2}{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})} + \ln \frac{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{c_i} - \frac{\left(y_i^{\text{exp}} - c_i\right)^2}{c_i}\right], \tag{12}$$

$$\text{where} \quad c_i = \sqrt{\left(y_i^{\text{exp}}\right)^2 + \tfrac{1}{4}} - \tfrac{1}{2},$$

obtained by using MLE for normally (Gauss) distributed data and used for Poisson distributed data applying the substitution $\sigma_i^2 = y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})$ also used for (9).

The detailed considerations leading to these statistics and their usage can be found in [8]. Here we restrict ourselves to mention a few additional facts about them. These statistics all follow asymptotically (as the number of data points, more accurately the degree of freedom (DOF) goes to ∞) a χ^2 distribution. For Poisson data χ^2_{PMLE} should be used during fitting. But according to the tests available in [8] this is not the appropriate *Goodness Of Fit statistic* (hypothesis test), for that $\chi^2_{\text{Pearson's}}$ should be used.

In case of simultaneous fitting, we can have experimental data with different distributions, therefore the statistics used to fit for each fitting problem may be

different. We fit a resulting statistic, their (weighted) sum. This is not a problem, as if we start from the MLE, from which all of them are derived, we would obtain also such a resulting statistic. In the case of the resulting statistic, the above mentioned names generally will not have any meaning, therefore in the program it is referred to just as the 'Fitted Statistic'.

There are experiments, where the data usually are preprocessed. E.g. in case of neutron reflectometry, the experimentalists normalize the results, as they prefer to plot reflection and not count rate. The problem with this approach is that calculating the (9-12) statistics, their value will not be the correct one. In case of the classical χ^2 statistic, defined by (8), this is not a problem, as it is enough just to normalize σ_i as well. We are able even to fit, as for that it is enough to know the location of the minimum. The value of the statistic gives some information about the quality of the fit. If we just fit the value of the statistic, this question is not as interesting, as in case of parameter uncertainty estimation, or hypothesis test. Without knowing the correct value the uncertainty estimations, hypothesis test will be incorrect. Therefore we need the raw, unpreprocessed data set, or at least the parameters used during preprocessing in order to be able to calculate the correct statistic, e.g. in the above mentioned example we need the normalization factor.

With the above mentioned *Goodness Of Fit statistic* $\mathcal{S}_{\text{GOF}}$ it may be checked whether the used model is correct or not. For χ^2 statistic this can be done by calculating the probability $Q(\chi^2_{\text{min}}, \nu)$ that the observed χ^2 exceeds the fitted value χ^2_{min} even for a correct model. Q is the incomplete gamma function:

$$Q(\chi^2_{\text{min}}, \nu) = \frac{1}{\Gamma(\frac{\nu}{2})} \int_{\frac{1}{2}\chi^2_{\text{min}}}^{\infty} \mathrm{e}^{-t} t^{\frac{\nu}{2}-1} \mathrm{d}t, \tag{13}$$

where Γ is the gamma function. The number ν is the degrees of freedom, which is the number of data points minus the number of fitted parameters. Q is usually called the 'goodness-of-fit.' As a rule of thumb in textbooks is stated, that models with $Q < 0.001$ are likely wrong. (To be honest, we never saw such a good fit for spectra fitted by the program. X-ray reflectometry is very far from that. The Mössbauer spectra are nearer but still far below this value, so for them it should be taken more seriously. The probable problem with X-ray reflectometry is that our theoretical models still neglect some properties of the physical system, e.g. the material inhomogeneities.) For data with Poisson distributions this value can be much lower. Models with small Q values may be accepted only if we know the reason. (The X-ray reflectometry is a good example for this.) Very good fits $Q \approx 1$ are also suspicious, as they usually arise if the experimenter overestimated her(is) uncertainty, or made something we would never assume of anyone. As

another measure of 'goodness-of-fit', often the 'reduced' (or relative) χ^2 statistic is used

$$\chi^2_{\text{reduced}} = \frac{\chi^2}{\langle\chi^2\rangle} = \frac{\chi^2}{\nu}. \tag{14}$$

As a rule of thumb χ^2_{reduced} should be close to 1 for good models (for Mössbauer spectra this is usually true, for X-ray reflectograms not). This depends strongly on the degrees of freedom. This rule is based on the fact that the χ^2 statistic has a mean ν and standard deviation $\sqrt{2\nu}$ and for large ν becomes normally distributed. As the above mentioned statistics also have asymptotically χ^2 distribution, these rules of thumb may be useful for them and their weighted sum, but we should be cautious.

If we start from the maximum likelihood estimation in case of a simultaneous fitting problem, we will get that the weights of the statistics with which they are summed up in order to get the common resulting statistic(s) should be 1. But sometimes it may be useful to have the possibility to change these weights. This may be useful especially when we are still far from the minimum. Usually, it is not a good idea to start fitting all the problems at once, if we are far from the true minimum. Initially, it is worth to fit the most simple (and most error free) problem which can be simulated fast and 'promises' to get good preliminary results for the common parameters easier. And only if we reached an acceptable result, using this simpler fitting problem, should we continue the fitting, adding the other fitting problems. This way we can progress faster still having the simultaneous fitting, which is the single correct way for evaluation of spectra, where we have measured the same sample with different methods and/or under different conditions and so forth.

Some fitting methods do not require these statistics or their sums directly. Instead they require a vector $\boldsymbol{\kappa}(\mathrm{p})$ and maybe its derivatives according to the parameters. The square of i-th component of this vector should give the contribution of the i-th data point to the corresponding statistic, i.e.

$$\chi^2_{\cdots} = \sum_i \kappa_i^{\cdots}(\mathbf{x}_i, \mathbf{p})\kappa_i^{\cdots}(\mathbf{x}_i, \mathbf{p}) = \boldsymbol{\kappa}^{\cdots}(\mathrm{x}, \mathbf{p})^T \boldsymbol{\kappa}^{\cdots}(\mathrm{x}, \mathbf{p}) \tag{15}$$

E.g.: in case of classical χ^2 statistic defined by (8) we will have

$$\kappa_i(\mathbf{x}_i, \mathbf{p}) = \frac{y_i^{\text{exp}} - y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{\sigma_i}, \tag{16}$$

in case of Pearson's χ^2 statistic defined by (9)

$$\kappa_i^{\text{Pearson's}}(\mathbf{x}_i, \mathbf{p}) = \frac{y_i^{\text{exp}} - y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p}))}{\sqrt{y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}}, \tag{17}$$

in case of modified Neyman's χ^2 statistic defined by (10)

$$\kappa_i^{\text{Neyman's}}(\mathbf{x}_i, \mathbf{p}) = \frac{y_i^{\text{exp}} - y_i^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{\sqrt{\max(y_i^{\text{exp}}, 1)}}, \tag{18}$$

in case of Poisson MLE statistic defined by (11)

$$\kappa_i^{\text{PMLE}}(\mathbf{x}_i, \mathbf{p}) = \sqrt{2} \begin{cases} \sqrt{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p}) - y_i^{\text{exp}}} & \text{if} \quad y_i^{\text{exp}} = 0 \\ \sqrt{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p}) - y_i^{\text{exp}} - y_i^{\text{exp}} \ln \dfrac{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{y_i^{\text{exp}}}} & \text{if} \quad y_i^{\text{exp}} \neq 0 \end{cases} \tag{19}$$

and in case of Gaussian MLE statistic defined by (12)

$$\kappa_i^{\text{GMLE}}(\mathbf{x}_i, \mathbf{p}) = \sqrt{\frac{\left(y_i^{\text{exp}} - y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})\right)^2}{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})} + \ln \frac{y^{\text{theo}}(\mathbf{x}_i, \mathbf{p})}{c_i} - \frac{\left(y_i^{\text{exp}} - c_i\right)^2}{c_i}}, \tag{20}$$

$$\text{where} \quad c_i = \sqrt{\left(y_i^{\text{exp}}\right)^2 + \frac{1}{4}} - \frac{1}{2}.$$

2.5 Uncertainty estimation, bootstrap method

Uncertainty (error) estimation of the fitted parameters $\mathbf{p}_{\text{fit}}$ is also a complex issue. The usual procedure to obtain the uncertainties is based on the fact, that for χ^2 statistics the uncertainty of parameters with confidence level c and degree of freedom M can be obtained looking for the minimal hyperrectangle (with edges parallel to the unit vectors belonging to parameter components) containing the hypervolume $V = \{\mathbf{p} | \chi^2(\mathbf{p}) \leqslant \chi^2(\mathbf{p}_{\text{fit}}) + \gamma\}$, where γ is the c quantile belonging to chi-square distribution with degree of freedom M. (A c quantile[2] $(0 < c < 1)$ of a (cumulative) distribution (function) $F(x)$ is γ if $F(\gamma) = c$.) As the fitted parameters $\mathbf{p}_{\text{fit}}$ are obtained with minimization of χ^2, therefore their vector will be part of V. For further details, see chapter 15 in [9]. This approach is also valid for statistics (9-12), for reasons see section 2.4 in [8].

To simplify the procedure further, it is often assumed that the fitted statistic as a function of fitted parameters has a parabolic profile in V and therefore knowing the second derivatives of the fitted statistic, the parameter uncertainties δp_i can be obtained by solving a second order equation $\sum_{i,j} A_{ij} \delta p_i \delta p_j = \gamma$. Sorrily in case of nonlinear problems, the assumption about parabolic profile is usually correct only in a small part of V (in neighborhood of $\mathbf{p}_{\text{fit}}$) and not in the whole V (see Fig. 1). Therefore, we may estimate the uncertainties quite inaccurately using

[2] https://en.wikipedia.org/wiki/Quantile function

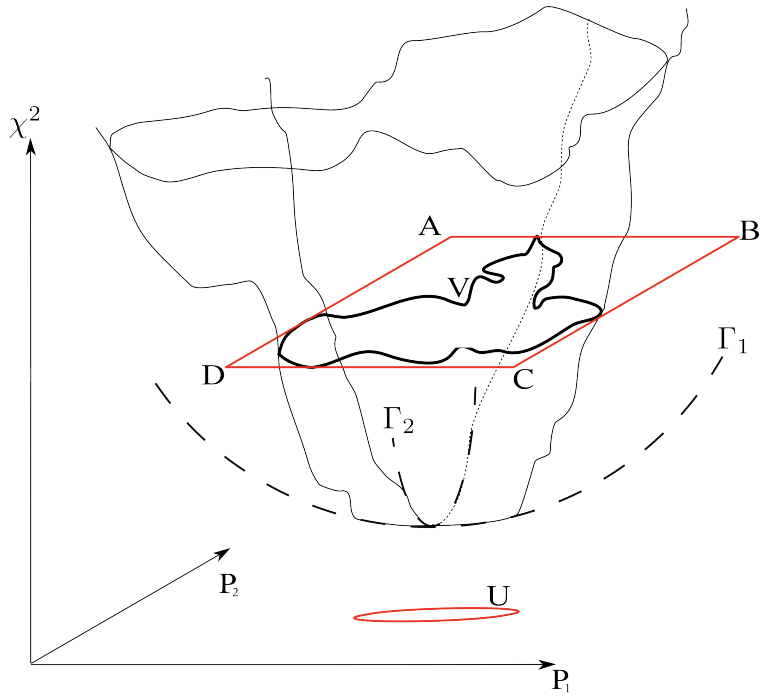

Figure 1: Uncertainty estimation for two free parameters. As it can be seen, the parabolic approximation is valid only in domain U, where the parabolas Γ_1, Γ_2 are good approximations of the sections of the $\chi^2(P_1, P_2)$ with planes parallel to P_1 and P_2 respectively. The true uncertainty for the given confidence level is given by the bounding rectangle $ABCD$ of V which is the corresponding 'elevation' line. And $[P_1^A, P_1^B]$ and $[P_2^D, P_2^A]$ are the confidence intervals.

this method. If it is applicable, we may also use the fact, that A_{ij} is with good approximation the inverse of the covariance matrix [see page 683 in chapter 15 of ref. 9]. This may have advantages in case of some methods, where this matrix is calculated in each iteration step.

There is another possible source of error using this approach, if we do not have (as is the case mostly) the analytical derivatives according to the fitted parameters of the 'theoretical function' used to model the problem, which is needed to calculate the second order derivatives of the fitted statistic. In this case the derivatives should be calculated numerically, which may have quite unacceptable errors in some regions of the parameter space (see subsection 2.6.5).

Another approach is the bootstrap method [10–14] using synthetic data sets [see pages 689-699 in section 15.6 of ref. 9] generated by Monte Carlo methods. We may generate synthetic data sets from the measured data sets:

- randomly erasing some data points, or
- randomly replacing some data points (this can be used only, if we have several measurements for the same independent variable) with another measurement value, or
- replacing the measured values y_i^{exp} with $y_i^{*\text{exp}}$ generated as it would have been drawn randomly from a 'bootstrap sample' whose mean is y_i^{exp} and its elements are distributed according to the distribution assumed for the experimental data. Therefore, we may have to know besides y_i^{exp} all the other parameters on which the distribution depends. E.g. in case of normally distributed data we have to now the deviation (i.e. the uncertainties of the data), but y_i^{exp} is enough in case of Poisson distributed data, as that already determines unequivocally the distribution.

The synthetic data sets, which in principle could also be the results of the experiments performed (with the known uncertainties), are fitted. The results of the fit of a synthetic data set is stored only if it was convergent and the corresponding fitted statistic differs from the fitted statistic of the experimental data set less than a user defined small positive number. We filter out this way the synthetic data sets which give a very different fitted statistic, because they miss already too much information compared to the real experimental data, or because the data points were not changed really randomly, e.g. $\left|\sum\left(y_i^{\text{exp}} - y_i^{\text{synth}}\right)\right|$ becomes unacceptably large because of a 'synthetic systematic error.' It may also happen, that the fit has gone wrong unexpectedly, and we do not want to throw out everything just because a few such bad fits. (If the fitted statistic of a synthetic data is less, than the value used in the 'filtering criterion', that value is updated with it. Therefore, the bootstrap method may also be used as some sort of fitting method.)

Applying this algorithm, we survey the 'basin' in the parameter space which contains the parameter vectors providing fits with the same quality for some possible outputs of our experiment(s) according to the experimental results and our knowledge about the precision of the measurement. Therefore, using the set of fitted parameter vectors of synthetic data sets, we may get a probability density (histogram) $f(\mathbf{p})$ corresponding to possible experimental uncertainties. Knowing $f(\mathbf{p})$ we can calculate the expectation value and the standard deviation of fitted parameters. This standard deviation can be used as an uncertainty estimate. The bootstrap method needs a lot of ($\gtrapprox 2000$) synthetic data sets,[3] therefore it may be very expensive in computation time. It is advisable to calculate uncertainties only

[3]The number 2000 is not graved in stone, in literature we may find lower values. This depends on the problem, on the paper.

when we have a quite good fit. For usability conditions and further details of the bootstrap method see the cited works.

It is inappropriate to give just the uncertainties for a given confidence level, as we may never know, when somebody will need our data with higher our lower confidence levels. In that case it is very useful to have the uncertainties for quantile=1, which is the sample standard deviation, as thereafter the uncertainties knowing the degree of freedom can be calculated for arbitrary confidence level easily.

The covariance matrix has other uses than the uncertainty estimation. We can 'discover' interdependencies between the parameters looking for off-diagonal elements with large (>0) absolute values compared to the corresponding diagonal elements. In case of such interdependency the transformation matrix technique may be useful.

2.6 Fitting

Without going into the details here, we try to summarize the approaches used to fit experimental data sets using optimization methods, enhancing the facts, which may be important even for the users, who sorrily do not know (and maybe would not like to know) the mathematical background scrupulously. For further details we refer to the rich literature about this topic. E.g.: chapter 10 of *Numerical Recipes in C* [9] available at http://www.nrbook.com, or as a good starting point see the optimization page on Wikipedia[4] and the references available there.

We use optimization methods, as fitting parameters we want to find the parameters for which the fitted statistic assumes its minimum. As the fitted statistics are positive definite such minimum should exist, but there maybe several one, and a lot of local minima. Therefore, finding the global minimum(ma) of a general function depending on a lot of parameters (variables) is not easy. There is no perfect solution, user interaction, intuition is needed. In case of fitting, we usually have some preliminary knowledge about most of the fitted parameter values, and we want just to have more accurate values, and to determine only a few totally unknown parameters, and even in that case we may have some conjecture about the range in which we should look after them. (Preliminary simulations may be very helpful at this stage.) We start the fitting from a point of the parameter space, which according to expectation is not too far from the solution we are looking for. If we have luck the method will find it, or will get nearer to the solution. The method may stuck in a local minimum, or in more unlucky cases the method may become divergent, or it may need further iteration, to get closer to the minimum.

To understand these features, we have to tell more about these methods. It is

[4] https://en.wikipedia.org/wiki/Mathematical optimization

common in all of them, that they are some sort of iteration algorithms. And that in each iteration step, the value (and/or derivatives) of the *objective function*, whose minimum is to be found, are calculated in discrete points of the parameter space determined by the method and it is tried to determine, whether is there a minimum, or where we have to take up the new points, in which the objective function should be examined next. If we look at the path of the iteration steps getting nearer and nearer to the current minimum, we will get a curve similar (except of some jitter) to a meandering river flowing always to lower levels. It may be imagined, that this path may be quite complex in higher dimensions. Reaching the minimum can take a lot of time and we may stuck into local minima much easier. As we can examine the function only at finite number of points, if the resolution of the method (specified by proper options) is not high enough we may even step over some minima, if it is too high it may take much longer time to get there. The methods were devised to make the best possible compromise, but they are not perfect. It is the task of the user to influence the fitting by setting the proper options of the method appropriately.

2.6.1 Constraints (Simple bounds)

As we mentioned in the beginning knowing approximately the domain in which the parameters may be found, can be very helpful. We can add to our optimization problem constraints of the form $g_j(\mathbf{P}) \leqslant 0$. To solve such optimization problems with constraints, several methods were devised. Here we do not dive into the *nonlinear programming*, which tackles the most general problems. We will show only three simple methods, two of which are used currently in our program. We have currently only simple bounds, in which case $g_j(\mathbf{P}) = P_{i_j} - c_j \leqslant 0$, or $g_j(\mathbf{P}) = -P_{i_j} - c_j \leqslant 0$, where c_j-s are constants, simplifying the problem further.

The first two methods, the *penalty* and *barrier* methods have common features, as in both cases the objective function is modified. In both cases we replace the problem with a series of unconstrained optimization problems which should converge to the original problem. We will have a sequence of objective functions of form

$$H_k(\mathbf{P}) = f(\mathbf{P}) + q_k Z(\mathbf{P}), \quad q_k > q_{k-1} > 0, \quad q_k \to \infty \tag{21}$$

where $f(\mathbf{P})$ is the original objective function, $Z(\mathbf{P})$ is the penalty function and q_k is the monotonically increasing divergent sequence of penalty coefficients. If M is the *feasible region* in the parameter space given by the constraints the penalty function should be of the form

$$Z(\mathbf{P}) = \begin{cases} 0 & \mathbf{P} \in M \\ > 0 & \mathbf{P} \notin M \end{cases} \tag{22}$$

Two often used examples for such penalty functions with m constraints:

$$Z(\mathbf{P}) = (\max\{0, g_1(\mathbf{P}), \dots, g_m(\mathbf{P})\} + 1)^r - 1 \quad (r = 1, 2, 3, \dots) \tag{23}$$

and

$$Z(\mathbf{P}) = \sum_{i=1}^{m} (\max\{0, g_i(\mathbf{P})\} + 1)^r - m \quad (r = 1, 2, 3, \dots). \tag{24}$$

Solving the modified optimization problems consecutively, starting from the solution of the previous modified problem each time, we may get close to the real solution of the original constrained problem.

In the program FitSuite we refer: to the function (23) and (24) as *1.* and *2. Penalty function,* respectively; to r as *Penalty exponent*. The penalty coefficients are of the form $q_k = q^k$ of a power sequence and q is referred to as the *Penalty multiplicator*. Convergence is reached if for all the fitted free parameters $\left|p_i - p_i^{\text{previous}}\right| \leqslant \tau_{\text{tolerance}} \cdot \frac{1}{2}\left(|p_i| + \left|p_i^{\text{previous}}\right|\right)$, i.e. the relative change in all the parameter values is less, than the tolerance factor $\tau_{\text{tolerance}}$ which in the program is referred to as *Penalty tolerance.*

With barrier method we have a sequence of objective functions of form

$$H_k(\mathbf{P}) = f(\mathbf{P}) + w_k B(\mathbf{P}), \quad 0 < w_k < w_{k-1}, \quad w_k \to 0 \tag{25}$$

where $f(\mathbf{P})$ is the original objective function, w_k is the sequence of monotonically decreasing barrier coefficients converging to 0 and $B(\mathbf{P})$ is the barrier function growing to ∞ on the boundary of M. Because of this property of the barrier function it is useful in case of $g_j(\mathbf{P}) < 0$ constraints, but numerically (as we have always a finite precision using computers) there is not much difference between $g_j(\mathbf{P}) < 0$ and $g_j(\mathbf{P}) \leqslant 0$. Two often used examples for such barrier functions with m constraints:

$$B(\mathbf{P}) = -\sum_{i=1}^{m} -\ln\left(g_i(\mathbf{P})\right) \tag{26}$$

and

$$B(\mathbf{P}) = \sum_{i=1}^{m} \frac{1}{(g_i(\mathbf{P}))^r} \quad (r = 1, 2, 3, \dots). \tag{27}$$

In contrast with the penalty method the fitting program using the barrier method may not get out of the domain defined by constraints. (This is not quite the case working numerically.) This feature may be especially useful, if the calculated

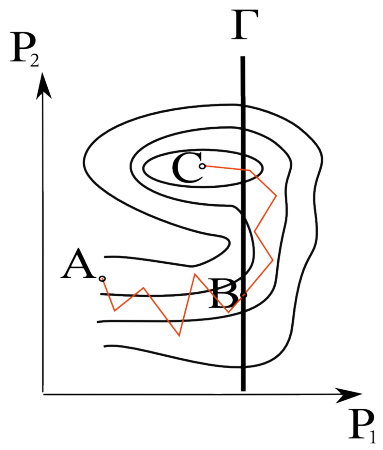

Figure 2: Demonstration of artificial local minimum arising because of constraint on a contour plot, where the contours denote decreasing levels approaching the real minimum C. Without the bound represented by line Γ for parameter P_1, the fitting started from A would be finished in C, but because of bounds it will be finished in B.

spectrum as a mathematical function of parameters has a finite domain, out of which we may get into unpredictable problems.

The third method handling the constraints is quite different. We use our fitting method for problems without constraints, but we check, in each iteration step whether we are out from the feasible region. If we are out, we continue with the nearest parameter vector on the boundary on the feasible set and if the method uses also gradient we continue with the component of the gradient projected on the boundary. This method is almost the *projected gradient method* [e.g. see ref. 15], but may also be used in case of optimization methods without derivatives. This *projection method* is much faster, than the barrier or penalty method, but it may have its own problems. If some constraint cuts through a bend belonging to the 'path' obtained connecting the steps of the optimization method, we may get an artificial local minimum on the boundary, where the method may stuck (see Fig. 2). In case of a very complex, meandering path, lot of constraints, parameters to fit, the number of such artificial local minima is multiplied. Therefore, we should be cautious using constraints. Although the usage of the penalty and barrier methods is a bit safer in this regard, but this problem may also arise there.

To use the penalty or barrier function method in case of 'vector statistics' defined by (16 - 20) should be modified, during fitting. E.g. in case of penalty

function, we should replace in these equations κ_i by

$$\kappa_i' = \sqrt{\kappa_i^2 + \frac{q_k}{n} Z(\mathbf{P})}, \quad i = 1, \ldots, n. \tag{28}$$

2.6.2 Distributed parameters (using maximum entropy principle)

To fit problems with distributed parameters, we minimize the fitted statistic χ_*^2 and maximize the entropy S of the distribution (see subsection 2.3), therefore we minimize $\chi_*^2 - S$. As we prefer positive definite objective function f^{obj}, remember on vector statistics defined in (16-20), instead of this we minimize

$$f^{\text{obj}} = \chi_*^2 + S_{\max} - S = \chi_*^2 + \ln N + \sum_{\substack{i=1 \\ h_i > 0}}^{N} h_i \ln h_i, \tag{29}$$

as S is maximal if $h_i = \frac{1}{N}$. If we have distributed parameters, we sum the entropies of the m independent distributions

$$f^{\text{obj}} = \chi_*^2 + \sum_{j}^{m} \left(\ln N_j + \sum_{\substack{i=1 \\ h_{j,i} > 0}}^{N_j} h_{j,i} \ln h_{j,i} \right). \tag{30}$$

The 'vector statistics' defined by (16-20) should be modified similarly to (28), i.e. we should replace in these equations κ_i by

$$\kappa_i' = \sqrt{\kappa_i^2 + \frac{1}{n} (S_{\max} - S)}, \quad i = 1, \ldots, n. \tag{31}$$

The penalty (or barrier) function can be added the same way too.

In order to fulfill automatically the constraints $\mathcal{R}_i \geqslant 0$, $h_{j_1,\ldots,j_n} \geqslant 0$ and $\sum h_{j_1,\ldots,j_n} = 1$ (see subsection 2.3), we fit instead of these parameters π_i^R, $\pi_{j_1,\ldots,j_n}^h$ and calculate from these $\mathcal{R}$ and h using the formulae:

$$\mathcal{R}_i = \left(\pi_i^R\right)^2 \tag{32}$$

$$h_{j_1,\ldots,j_n} = \frac{\left(\pi_{j_1,\ldots,j_n}^h\right)^2}{\sum \left(\pi_{j_1,\ldots,j_n}^h\right)^2}, \tag{33}$$

which give the definition of 'π' parameters as well. This is working correctly, but sometimes, we may experience a bit different behavior fitting these distribution parameters compared to the normal parameters.

2.6.3 Rescaling parameters

The expected order of magnitude may also be a helpful information during optimization (fitting), as there are methods (most of them) which do not work properly (they may become even divergent) for parameters of different order of magnitude (see fit using rescaling demo). In these cases we can rescale the parameters $\mathbf{P}$ by dividing each P_i by the corresponding order of magnitude m_i and optimizing the modified objective function $f^{\mathbf{m}}(\mathbf{P^m}) = f(\mathbf{P})$, according to the rescaled parameters $P_i^{\mathbf{m}} = P_i/m_i$.

The parameter bounds give the order of magnitude, only if the signs of the upper and lower bounds are identical, that is the main reason, why the magnitude should be provided by the user separately.

In the program FitSuite, there are some 'corrections' which take into account the bounds as well. If $P_i^{\max} P_i^{\min} > 0$, i.e. the bounds have the same sign, then the modified scaling factor will be

$$m_i' = \min\{\max[m_i, \min(\left|P_i^{\min}\right|, |P_i^{\max}|), \lambda^{\min}], \max(\left|P_i^{\min}\right|, |P_i^{\max}|), \lambda^{\max}\}, \tag{34}$$

and if $P_i^{\max} P_i^{\min} < 0$, then

$$m_i' = \min\{\max[m_i, \lambda^{\min}], \max(\left|P_i^{\min}\right|, |P_i^{\max}|), \lambda^{\max}\}, \tag{35}$$

where $\lambda^{\min}$ and $\lambda^{\max}$ are the *Minimum scale* and *Maximum scale*, respectively, which can be changed by the user in FitSuite.

In FitSuite it is used only if the option *Scale parameters* is checked.

2.6.4 Parameter resolution, minimum step width

Sometimes, it is useful to have a minimum step width for the fitted parameters, as then it can be avoided to get in a long iteration cycle, where we make a lot very short steps. Therefore, each real parameter has such a value which is called *Resolution* in the program interface.

In FitSuite it is used only if the option *Use minimum step width* is checked (it has no effect in case of the genetic algorithm). Setting this we may get close to the minimum faster. And thereafter we can get the more accurate value by switching of the option *Use minimum step width*. In practice we check each step Δp_i of the algorithm (corresponding to the i-th free parameter) and we replace it with a step of minimum step width $\Delta p_i^{\min}$ according to:

$$\Delta p_i^* = \begin{cases} \operatorname{sgn}(\Delta p_i)\Delta p_i^{\min} & \text{if} \quad \theta\Delta p_i^{\min} < |\Delta p_i| < \Delta p_i^{\min} \\ \Delta p_i & \text{otherwise} \end{cases}, \tag{36}$$

where $0 < \theta < 1$ is the *Minimum step width threshold factor* used to avoid steps with minimum step widths in case of 0 or very small steps. If the set parameter resolution values are proved to be inappropriate, too small or too large, but their ratios to each other are acceptable, then the user can change them by using a multiplication factor, or a geometric sequence of multiplication factors and choose the best fit.

Besides this, the parameter resolution has another use related to the plotting of the fitted statistics according to a parameter(s), as the points in which we calculate are determined by the parameter bounds and resolution.

2.6.5 Numerical derivatives

Usage of methods using or not using derivatives is another question we should address a bit. Mathematicians usually assume, that we have objective functions, whose derivatives are known, therefore most of the optimization method uses them as well. The problem is, that in practice we usually do not have the derivatives in case of complex problems, as we do not have an analytical function, but we have algorithms, which may use a lot of numerical (iterative) algorithms already (e.g. determining eigenvalues in quantum mechanical problems, as Mössbauer line positions and line strengths calculated from Hamiltonian), and we have a lot of parameters. Therefore, calculating the derivatives requires tremendous additional work for which we usually do not have time and it may not be worth either. Still, we should provide the derivatives for the methods requiring it, therefore we should determine them numerically. The problem is, that because of the finite computer representation of the numbers and because it needs dividing of a number with a small number, numerical differentiation is dangerous, therefore it is avoided in numerical computations whenever possible.

The method implemented by us uses the same trick, as the Romberg's method for integration (see [page 140 in section 4.3 of ref. 9 available at http://www.nrbook.com], or wikipedia[5]), to get high precision derivatives. We know, that $f(x_0 + h) = f(x_0) + \sum_{i=1}^{\infty} h^i \frac{\mathrm{d}^i}{\mathrm{d}x^i} f(x_0)$, therefore using the notation

$$\begin{aligned} \mathrm{D}_h^0 f(x_0) &= \frac{f(x_0+h) - f(x_0-h)}{2h} = \frac{\mathrm{d}}{\mathrm{d}x} f(x_0) + \sum_{i=1}^{\infty} h^{2i} \frac{\mathrm{d}^{2i+1}}{\mathrm{d}x^{2i+1}} f(x_0) \\ &= \frac{\mathrm{d}}{\mathrm{d}x} f(x_0) + \mathcal{O}(h^2), \end{aligned} \tag{37}$$

where we use the *big O notation*[6], $\mathcal{O}(h^2)$ should be read as *order of* h^2. We can

[5] https://en.wikipedia.org/wiki/Romberg's method
[6] https://en.wikipedia.org/wiki/Big O notation

eliminate the higher order terms, as in Romberg integration. E.g.:

$$\begin{aligned} \mathrm{D}_h^1 f(x_0) &= \frac{4\frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{f(x_0+2h)-f(x_0-2h)}{4h}}{3} \\ &= \frac{4\mathrm{D}_h^0 f(x_0) - \mathrm{D}_{2h}^0 f(x_0)}{4-1} = \frac{\mathrm{d}}{\mathrm{d}x} f(x_0) + \mathcal{O}(h^4). \end{aligned} \tag{38}$$

Generally we may use

$$\mathrm{D}_h^{k>0} f(x_0) = \frac{4^k \mathrm{D}_h^{k-1} f(x_0) - \mathrm{D}_{2h}^{k-1} f(x_0)}{4^k - 1} = \frac{\mathrm{d}}{\mathrm{d}x} f(x_0) + \mathcal{O}(h^{2k}). \tag{39}$$

In order to calculate with accuracy $\mathcal{O}(h^{2k})$ we need $2k$ function evaluations

$$\left(f(x_0 - 2^k h), f(x_0 - 2^{k-1} h), \ldots, f(x_0 + 2^{k-1} h), f(x_0 + 2^k h)\right),$$

usually meaning simulations for the derivatives and one to get $f(x_0)$, which may be slow the fitting very much. If we fit several parameters and therefore we have to calculate several partial derivatives, the program will slow even further. And even then in general case we cannot guarantee, that the function has not a very great high order derivative, which deteriorates everything. We may check the convergence comparing different order of approximations. E.g. we may accept and stop calculating approximations of higher order if

$$\left|\mathrm{D}_h^k f(x_0) - \mathrm{D}_h^{k-1} f(x_0)\right| < C\left(\left|\mathrm{D}_h^k f(x_0)\right| + \left|\mathrm{D}_h^{k-1} f(x_0)\right|\right), \tag{40}$$

where $0 < C < 1$ is a small number giving the user required *precision.* It is useful to have a maximum for the order of approximations (In the program this appears as *Maximum difference order*), as numerically we cannot take h arbitrarily small (at least not without extra work and computation time which is the cost of using a library using numbers with arbitrary precision). The most plausible choice for h would be $|x_0|\varepsilon$ (for $x_0 \neq 0$) which is defined as $1 + \varepsilon$ being the smallest number which may be differentiated from 1 for a given machine precision, but the user may have other choice by changing *Initial step factor*, which corresponds to h.

2.7 Subspectrum

Sometimes it is useful to see what is the contribution of the parts of the studied system to the whole spectrum. Therefore, Mössbauer spectroscopists devised the concept of subspectrum. In Mössbauer spectroscopy the contributions of different sites (atomic environments of the Mössbauer isotope) of the sample are added up weighted by their concentration calculating the 'absorption coefficient' used to

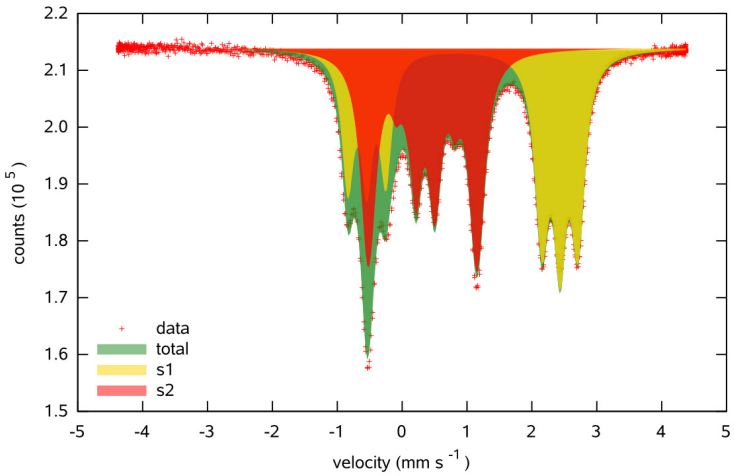

Figure 3: Two subspectra *s1, s2*, the *total* calculated Mössbauer spectrum and the corresponding data set.

determine the spectrum of the system. Therefore, plotting spectrum and subspectra Fig. 3, i.e. the spectra calculated taking into account only one (or a few, but not all) of the sites, we may see which site is corresponding to a specific peak, etc.

The subspectrum in FitSuite is a spectrum, where some of the physical objects of the studied system are not taken into account. This concept may also be helpful (in better understanding of the studied physical system) in cases different from Mössbauer spectroscopy, even if the whole spectrum cannot be obtained as weighted sum of single contributions, subspectra. (see subspectrum demo)

3 Working with FitSuite

In the following we try to show the features, the usage of the program in an order, as a new user should go step by step through the different interfaces of the program.

3.1 Starting a new project

Starting the program, the user can start a new project (File > New Project) or load (File > Open) a previously saved one. The extension of the project files is '.sfp' (**s**imultaneous **f**it **p**roject). We can save our project anytime clicking File >

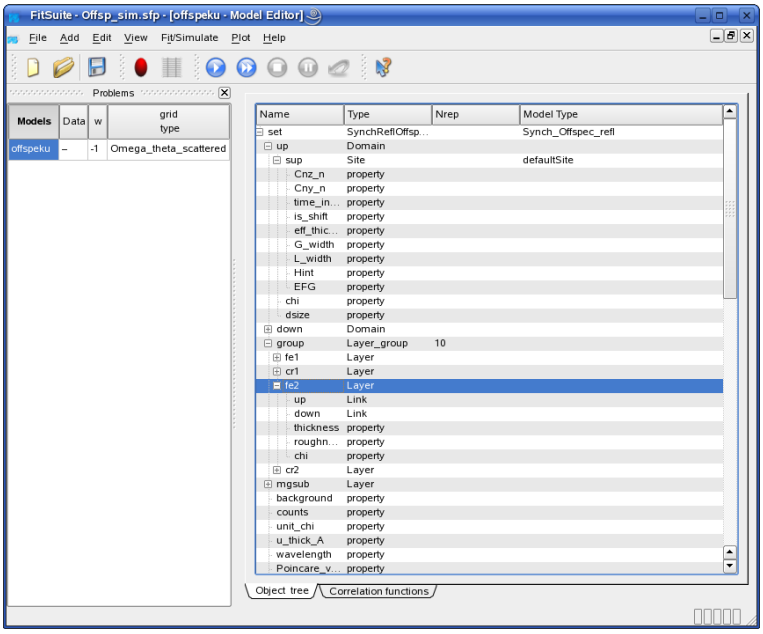

Figure 4: Model Editor: the part used to build up the structure represented by a tree.

Save...

If we have a new (empty) project, we have to add models (theories in EFFI terminology). This can be done in two ways. The user can load it from a '*.mod' file (if there is such available, e.g.: such files may be created by exporting), or you can click Add > New Model and then can choose from a list. The model appears with an initial name 'Model$i$' ($i = 0, 1, \ldots$) in the window **Problems** on the left of the main window (see Fig. 4). You can change the name clicking on the text 'Model$i$' and typing the new name in this window. Please do not use white spaces (space, tabulator etc.) in names in FitSuite, as it may have very queer consequences. Currently only ASCII characters may be used in names. Using other type of characters may have undesired side effects. (see model definition demo)

3.2 Building up the model structure

On the right side of the main window (Fig. 4) should be seen a **Model Editor** now. In this we can build up the hierarchical structure of the model.

On the top is always one object the experimental scheme. We can add other objects to this (and to every object) with right clicking on it (them) and choos-

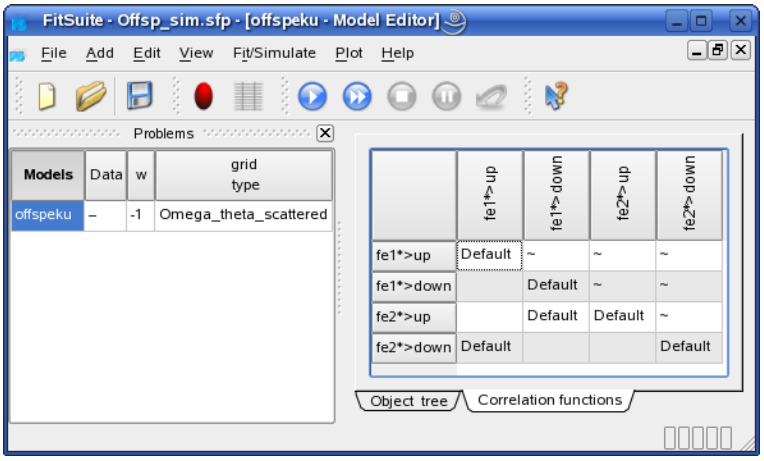

Figure 5: Model Editor: the part used to choose the correlation functions

ing Add or Insert or Read and add from file from the arising pop-up menu (the usage of *Read and add from file* is described in subsubsection 3.2.1). If there is more than one possibility, you may choose by moving the mouse on Add ... . In the current built in model types only the layers may be grouped. Selecting them with shift+cursors or shift+mouse and right clicking on selection you can choose Group from the pop-up menu. In case of the objects representing the groups the column labelled with *Nrep* contains the repetition number telling us, how many times the elements of the group are repeated in the real physical system. This could be changed with right clicking on it in previous versions. Now you can change it setting the corresponding integer parameter (see later).

In case of off-specular (synchrotron Mössbauer reflection) problems we have to give the correlation functions between the domains belonging to different layers. The domains available at the level of scheme can be linked to the layers. The correlation functions can be chosen with the help of graphical user interface, which appears after clicking on tab with label "Correlation function" on the bottom of the model editor (see Fig. 5).

3.2.1 Adding objects, models using xml like files

Sometimes it can be useful to add objects, or a whole model structure reading from a file generated by some other program, macro, etc. It is obvious, that we need some sort of file format convention to be able to do this. In this subsubsection, we will see by examples, how this can be done in FitSuite and what file convention is used.

In the directory *examples/ReadingObjectsFromFile*, we may find several ex-

amples, some of which will be described here as well. The order of the examples, as it is explained in *examples/ReadingObjectsFromFile/README*, is not random, it has didactic reasons. Some of the explanations will be available in the example files (which are also included in this manual) as comments.

1.) *TwoSites.txt* is the most simple example. To use this create a 'Mossbauer-Transmission' model with a sample and at least a layer (or use a model available of this type, e.g. *examples/MossbauerSpectraFemtz/Fit.sfp*). To add to the layers the two sites defined in *TwoSites.txt* open the Model Editor right click on the name of the chosen *Layer* and select from the pop-up menu Read and add from file, and open the file TwoSites.txt. Thereafter, we should see two new sites.

Example 'xml' file 1: TwoSites.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1  //Comments as in C++ start with "//"
2  //Two sites with model type defaultSite. The names are s5 and s6. Hint::r will be 25 kG (for s5)
     ↪ and 10 kG (for s6), Hint::theta 1 radian (for s5) and 2 radian (for s6), Hint::phi 5 radian (
     ↪ for s5) and 6 radian (for s6). The units are kG and radian, because of historical reasons (
     ↪ EFFI) these are the internal (default) units used in calculations.
3  //The other properties not listed here will have their default values.
4
5  <Site = defaultSite | List | Name, Hint::r, Hint::theta, Hint::
     ↪ phi> //"Site" is the type name of the objects, as sites may belong to several model types,
     ↪ that should be specified here too, and that is why we have the "= defaultSite", "=" is used as
     ↪  separator.
6  //"List" shows that this is a list of sites and "|" symbols are separators, in another example we
     ↪ will see cases in which here we have something else.
7  //Remark "| List |" and "| |" are equivalent as we will see in other examples.
8  //After the second "|" symbol the separator will be ",". The "Name" specifies, that in the first
     ↪ column we will provide the names of the sites, as we can see below s5 and s6. If we would
     ↪ delete "Name," and s5 and s6 below, then the program would generate names automatically
     ↪ . Names are also generated automatically in cases, when there is already an object with a
     ↪ given name.
9  //Hint::r is expected in the second column, Hint::theta in the third and Hint::phi in the fourth.
10
11 //Each line will correspond to an object (here Site), therefore (ATTENTION) do not break lines
     ↪ corresponding to one object. The columns are separated by white spaces, i.e. space(s) and/
     ↪ or tabulator(s).
12 s5 25 1 5 //The first site should be named s5. If there is already an object named s5 in the
     ↪ model, then s5 it will be complemented with a suffix, i.e. an automatically generated number
     ↪ , to avoid name collision. Hint::r will be 25 kG, Hint::theta 1 radian, Hint::phi 5 radian.
13 s6 10 2 6 //The second site should be named s6, and Hint::r will be 10 kG, Hint::theta 2
     ↪ radian, Hint::phi 6 radian.
14 </Site> //Like in html/xml <Site...> is ended here.
```

2.) *ThreeSites.txt* is similar to *TwoSites.txt*, but here we have sites with different model types, i.e. the calculation of the corresponding subspectra and used parameters are different.

Example 'xml' file 2: ThreeSites.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1 //It is similar to TwoSites.txt, but here we have different type of sites.
2 <Site=doubletSiteWPI | List | Name, G_width::[1], G_width::[2],
  ↪ rel_intensity::[1], rel_intensity::[2], splitting> //Here we have
  ↪ two doublet sites
3 s7 1 1 1 2 2
4 s8 2 2 2 1 3
5 </Site>
6
7 <Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness,
  ↪ G_width, L_width, Hint::r, Hint::theta, Hint::phi, EFG::zz,
  ↪ EFG::eta, EFG::phi, EFG::theta, EFG::psi> //site using the solution of
  ↪ the hamiltonian
8 s9 2 -0.01 2 2 5 10 1 0 .22 .3 -0.12 1.2 .05
9 </Site>
```

3.) In *ThreeSitesUnit.txt*, the units are also given for some of the properties + components (parameters). Otherwise, the program assumes that the parameters read in are in the units used by FitSuite in internal calculations. Reading parameters for which the unit is given does not mean, that in the parameter list it will appear in this unit, as it is converted by the program to the internal units.

Example 'xml' file 3: ThreeSitesUnit.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1 //It is similar to ThreeSites.txt, but here we have units for some of the parameters.
2
3 <Site=doubletSiteWPI | List | Name, G_width::[1], G_width::[2],
  ↪ rel_intensity::[1], rel_intensity::[2], splitting>
4 s7 1 1 1 2 2
5 s8 2 2 2 1 3
6 </Site>
7
8 //The units are given in parentheses "()" following the name of the corresponding property::
  ↪ componentname.
9 //The names of the units can be found by using the "Command-line" interface of FitSuite (in a
  ↪ tab on the bottom part). We need the command "listUnitsOf", the description of which can
  ↪ be found in the UserManual.pdf available in doc/pdf directory. (The "Parameter filtering"
```

```
  ↪ and parts of the "Command−line interface" sections in the UserManual.pdf may be needed
  ↪ for understanding, especially the introductory parts.)
<Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness,
  ↪ G_width, L_width, Hint::r (mT), Hint::theta (deg), Hint::phi
  ↪ (deg), EFG::zz, EFG::eta, EFG::phi(rad), EFG::theta(deg), EFG
  ↪ ::psi (deg)> //here Hint:r is given in mT, therefore it will be 10mT, Hint::theta in
  ↪ degree, and therefore it will be 45 deg, similarly EFG::psi 30 deg, but EFG::phi −1 radian.
s9 2 -0.01 2 2 5 10 45 90 .22 .3 -1 60 30
</Site>
```

4.) *LayerThreeSites.txt* Here we have a layer containing sites. This should be added in the Model Editor by right clicking on the name of the Sample and selecting from the pop-up menu Read and add from file,

Example 'xml' file 4: LayerThreeSites.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1  //Here we have a layer containing sites
2  <Layer |List| thickness> //The layer has automatic name and the thickness is set
3  <> //Without this (and </> below) see LayerThreeSitesBad.txt 3 layers would be generated. The
     ↪ first one would contain the sites s5 and s6 and it would have the default layer thickness 0.
     ↪ The second layer would contain the site s7 and would have 0 thickness. And the third one
     ↪ would not contain any site, but its thickness would be 6.73 ångström.
4  //The <> </> 'brackets' show, that the lines between them belong to a single layer.
5  <Site=defaultSite | List | Name, Hint::r, Hint::theta, Hint::phi>
6  s5 25 1 5
7  s6 10 2 6
8  </Site>
9  <Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness,
     ↪ G_width, L_width, Hint::r, Hint::theta, Hint::phi, EFG::zz,
     ↪ EFG::eta, EFG::phi, EFG::theta, EFG::psi>
10 s7 2 -0.01 2 2 5 10 1 0 .22 .3 -0.12 1.2 .05
11 </Site>
12 6.73 //The layer thickness will be 6.73 ångström.
13  </>
14 </Layer>
```

5.) *LayerThreeSitesBad.txt* It is a bad version of *LayerThreeSites.txt* created to explain, why we need empty "<>" "</>" strings. If we read in both and compare the results, it is obvious.

Example 'xml' file 5: LayerThreeSitesBad.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1  //For explanation see LayerThreeSites.txt
2  <Layer || thickness>
3  <Site=defaultSite | List | Name, Hint::r, Hint::theta, Hint::phi>
4  s5 25 1 5
5  s6 10 2 6
6  </Site>
7  <Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness,
     ↪ G_width, L_width, Hint::r, Hint::theta, Hint::phi, EFG::zz,
     ↪ EFG::eta, EFG::phi, EFG::theta, EFG::psi>
8  s7 2 -0.01 2 2 5 10 1 0 .22 .3 -0.12 1.2 .05
9  </Site>
10  6.73
11 </Layer>
```

6.) In *SourceSample.txt*, we have a source and a sample. This should be added in the Model Editor by right clicking on the name of the "root object" (i.e. which contains everything) and selecting from the pop-up menu Read and add from file. In this case, it is **important**, that we need a model which does not contain anything except of the root object. As a Mössbauer transmission model may have only one source and one sample.

Example 'xml' file 6: SourceSample.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1  //Here we have a source containing a singlet site, and a sample with one layer with 3 sites.
2  //IMPORTANT: In this case, we need a model which does not contain anything except of the root
     ↪ object. As a Mössbauer transmission model may have only one source and one sample.
3  <Source || Name>
4  <>
5  source //The name of the Source
6  <Site = singletSiteWPI || Name, eff_thickness, L_width>
7  srcSite 1 1 //The source is a singlet
8  </Site>
9  </>
10 </Source>
11
12 <Sample || Name>
13 <>
14 <Layer |List| thickness> //The layer has automatic name and the thickness is set.
15 <> //Without this (and </> below), see LayerThreeSites.txt and LayerThreeSitesBad.txt, 3 layers
     ↪ would be generated. The first one would contain the sites s5 and s6 and it would have the
     ↪ default layer thickness 0. The second layer would contain the site s7 and would have 0
     ↪ thickness. And the third one would not contain any site, but its thickness would be 6.73 å
     ↪ ngström.
16 //The <> </> 'brackets' show, that the lines between them belong to a single layer.
17 <Site=defaultSite | List | Name, Hint::r, Hint::theta, Hint::phi>
```

```
s5 25 1 5
s6 10 2 6
</Site>
<Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness, G_width, L_width, Hint::r, Hint::theta, Hint::phi, EFG::zz, EFG::eta, EFG::phi, EFG::theta, EFG::psi>
s7 2 -0.01 2 2 5 10 1 0 .22 .3 -0.12 1.2 .05
</Site>
6.73 //The layer thickness will be 6.73 ångström.
 </>
</Layer>
absorber//The name of the Sample
</>
</Sample>
```

7.) In *MossbauerModel.txt*, we create a "MossbauerTransmission" model. To use this select the menu item Add ⟩ Model from XML file, and choose this file.

Example 'xml' file 7: MossbauerModel.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
//Here we have a whole model
//To use this select the menu item: "Add | Model from XML file", and choose this file.
<Model=MossbauerTransmission || Name, base> //A model should start like this instead of the object type name likes "Site" we have always "Model" and after = the model type name of the model, here "MossbauerTransmission". Then we can specify the columns containing the name of the root object, and the properties like "base" here
<>
TreeRoot 70000 //As there is only one root object there is no use to have here more than a single line.

<Source || Name>
<>
source //The name of the Source
<Site = singletSiteWPI || Name, eff_thickness, L_width>
srcSite 1 1 //The source is a singlet
</Site>
</>
</Source>

<Sample || Name>
<>
<Layer |List| thickness> //The layer has automatic name and the thickness is set
<> //Without this (and </> below) see LayerThreeSitesBad.txt 3 layers would be generated. The first one would contain the sites s5 and s6 and it would have the default layer thickness 0. The second layer would contain the site s7 and would have 0 thickness. And the third one would not contain any site, but its thickness would be 6.73 ångström.
```

```
//The <> </> 'brackets' show, that the lines between them belong to a single layer.
<Site=defaultSite | List | Name, Hint::r, Hint::theta, Hint::phi>
s5 25 1 5
s6 10 2 6
</Site>
<Site=defaultSite | List | Name, Cnz_n, is_shift, eff_thickness,
 ↪ G_width, L_width, Hint::r, Hint::theta, Hint::phi, EFG::zz,
 ↪ EFG::eta, EFG::phi, EFG::theta, EFG::psi>
s7 2 -0.01 2 2 5 10 1 0 .22 .3 -0.12 1.2 .05
</Site>
6.73 //The layer thickness will be 6.73 ångström.
 </>
</Layer>
absorber//The name of the Sample
</>
</Sample>

 </>
</Model>//<=MossbauerTransmission>
```

8.) *SuperMirrorBase.txt* explains the usage of TypeList and repetition groups. It contains layers of a neutron super mirror. To use it, create a neutron reflection model with an incoherent part, in the Model Editor right click on the name of the incoherent part and select from the pop-up menu Read and add from file. **TypeList is not working properly in case of complete models (like in MossbauerModel.txt). It was tested only as in SuperMirrorBase.txt for neutron reflection, it may not work for other model types.**

Example 'xml' file 8: SuperMirrorBase.txt (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the file.)

```
1  //Number of layers: 66
2  //Number of layer groups: 5
3
4  <Layer |TypeList| Name, scatteringLengthDensity::.re,
    ↪ scatteringLengthDensity::.im > //This is a type list, as "TypeList" shows
    ↪ between the two "|" symbols.
5  //Here we define two layer types Ni and Ti, by their scattering length densities. In neutron
    ↪ reflectometry, this is the parameter which is material specific.
6  Ni 9.40572 -0.00115824
7  Ti -1.949 -0.000975259
8  </Layer>
9
10 <Layer |List| , TypeName, thickness > //The first column is not used as there is
    ↪ only space before the first ",". The second column contains the TypeName by which we refer
```

```
    ↪ to their definitions above in the TypeList. The scattering length densities of the layers and
    ↪ their names will be set accordingly.
0 Ni 2601.62
1 Ti 63.1249
2 Ni 451.291
3 Ti 126.25
<Repeat |group| 5 > //This is a repetition group with repetition number 5. The name of
    ↪ the group will be "group" complemented with a suffix (an automatically generated number)
    ↪ if it is necessary, i.e. there is already an object named group.
4 Ni 130.27
5 Ti 92.5466
</Repeat>
<Repeat |group| 7 > //This is a repetition group with repetition number 7
6 Ni 106.728
7 Ti 82.8688
</Repeat>
<Repeat || 9 > //The name of the group will be "noname" complemented with a suffix (an
    ↪ automatically generated number) if it is necessary, i.e. there is already an object named
    ↪ noname.
8 Ni 93.4042
9 Ti 76.1495
</Repeat>
<Repeat |Gruppe| 11 > //The name of the group will be "Gruppe" complemented with a
    ↪ suffix (an automatically generated number) if it is necessary, i.e. there is already an object
    ↪ named Gruppe.
10 Ni 84.3988
11 Ti 71.0282
</Repeat>
12 Ni 2601.62
</Layer>
```

3.2.2 Copying and inserting objects

There is another useful method to add new objects. If we select an object (or objects) in the Model Editor and right click on this selection, then in the pop-up menu will appear an item Copy. Choosing this menu item, the selected objects are copied, and thereafter their copies can be inserted before a selected object of the same type by right clicking on the object name and selecting the menu items Insert a copy or Insert copies in the pop-up menu.

3.3 Adding data

We can import from files the experimental data selecting the menu item Add > Data, after this we can choose the format of the file e.g.: 'one column (y)', 'two columns (x,y)', 'three columns (x,y,yerr)', 'three columns (x1,x2,y)', 'four

columns (x1,x2,y,yerr)', 'MCA file', 'Data series (x,{y})', 'compact' (I name so, as I could not find better name, the format:

```
  0   68348   68699   68315   68375   68253
  5   68198   68508   67983   68114   68041
 10   67436   67776   68123   68143   68480.
```

E.g. see the files *4k0t.dat, 4k3t.dat, 4k5t.dat* in directory *adatok*), etc.

In this dialog, we may give:

- The line with which the reading begins (labelled with **First line to read in**). The numbering starts with 0.
- The number of lines we want to read in (labelled with **Number of lines to read**). If it is 0 then it will read all.
- A string (labelled with **Until text**), this of course could be a number, until whose first occurrence (after the line which was given in *First line to read in*) we want to read in the lines.
- A string with which the comments start (labelled with **Comments start with**).
- The decimal delimiter, as sometimes we have data files which have decimal commas instead of decimal points, or vice-versa. (labelled with **Decimal delimiter is**).
- In case of compact format we can add:
  - the **Number of data columns**, in the above mentioned example this is 5, as the first (0-th) column (0, 5, 10, ...) gives the number of data in former lines. (see reading ASCII data file demo)
  - and the **First data column**, which in the example is 1.
- We may **Choose columns**. This option can be useful, if we have a different order of columns, or we have for example the results of several experiments in the same file, in different columns. In such a case, we can specify by column number, that e.g. the 4th column contains the independent *(x)*, the 2nd dependent variable *(y)*, the 15th the uncertainty (*yerr*) and so on.

  In case of 'Data series (x,{y})', we may create several data sets specifying several dependent variable *(y)* columns (in the editbox labelled with **Read as {y} columns**) by giving column ranges. E.g. **-3, 5-7, 9, 15-** will correspond to the columns **0, 1, 2, 3, 5, 6, 7, 9, 15, 16, ..., last column**. As

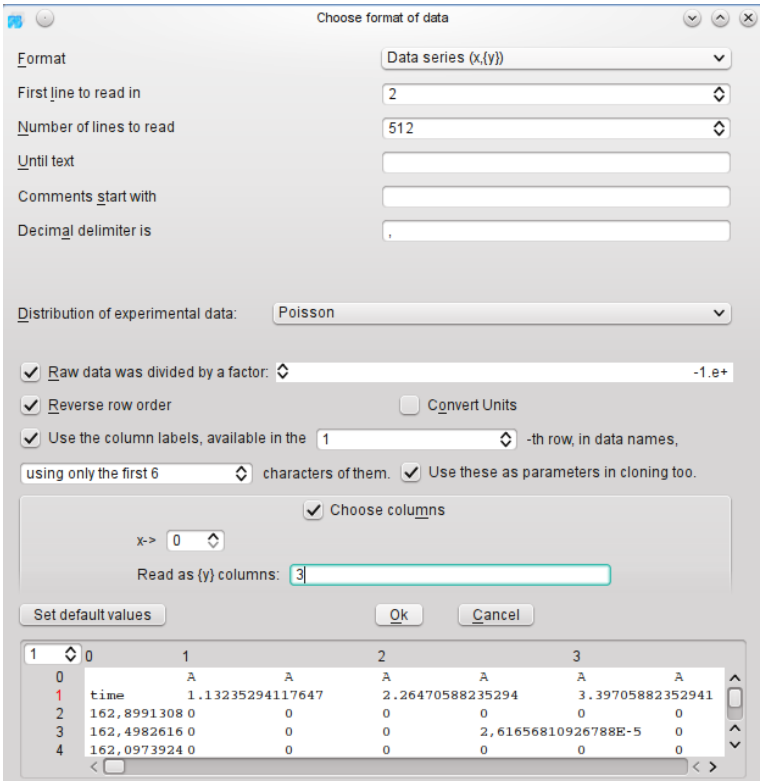

Figure 6: The dialog used to read in data sets in case of data series.

it can be seen from this example we can have several ranges separated by commas. And a range can be a single number or two numbers separated by a minus sign, if one of the numbers is missing that corresponds to first (-3 $\Leftrightarrow$ 0-3) or last column (15- $\Leftrightarrow$ 15-Last). The first column is indexed by 0.

- We may specify a row in which we have dependent variable column labels. These labels may be used to generate automatic names for the data sets. E.g. we have the number 1.357412 in row 2 column 3 (corresponding to the dependent variable of our new data set), then we will have a data set named *Data1.357412*. These feature can be especially useful in case of data series and/or if we have a lot of data sets to fit simultaneously, and we have a row containing labels or data which could be used as a label. If we would like to use only a part of this label, to have a shorter name, we may e.g. say that we would use only the first 4 characters of the label, and then in our example the name of the data set will be *Data1.35*.

  As it can be seen in Fig. 6, on the bottom of the dialog there is a box used to preview the data file, here we can check whether our settings are correct

or not. In top left corner there is a spin box, where we can chose the row number according to which the column numbers appear on the top of the previewing part. This is useful if we have a lot of columns, and the alignment of the columns in the different rows is far from perfect, as in Fig. 6 apparently is the case, and we have a column with a specific label in mind, or we would like check which row in our column a 'nasty' value like a NaN appears.

In case of cloning, the label can be used naming cloned models and/or parameter, and if the label is a number, it can also be used as a parameter, e.g. setting simulation/fit parameter values. For further details, see the part concerning cloning starting on page 63.

- We may reverse the row order, if the row order available in the data file is not what FitSuite expects, e.g. decreasing independent variables values.
- We may convert units of the columns, as in FitSuite the data columns are expected in predefined units, which may not agree with the units available in the data files.
- In case of 'MCA file' format (MCA stands for **M**ulti **C**hannel **A**nalyser), as it can be seen in Fig. 7, we may have several parameters (Number of channels, first channel, last channel, and other calibration parameters), which are read from the comments available in the header of the file. These parameters can be overwritten by the user, in case the program interpreted something incorrectly.

Presently, except of some cases mentioned above, the program does not read parameters, constants from data file.

Scans from Certified Scientific Software's spec™ (X-Ray Diffraction and Data Acquisition software) files can be obtained in another way from version 1.4.1. on. Open **spec** file clicking (File ⟩ Open spec File), wereafter a window should appear, where you may (filter) select the scans and extract the chosen data sets. If the required spec file has already been opened, it is enough to choose the menu item (View ⟩ Show Open spec Files), to have this window. Extraction types may be defined, changed. For further details, see this spec file demo.

The experimental data have some distribution. E.g. data obtained using particle counters usually have Poisson distribution. 'Ordinary' experimental data are expected to have Gaussian distribution. Fitting the experimental data, we have to know, which distribution should be used as the fitted statistics should be chosen accordingly. The type of the distribution can be chosen here or later. If the imported data contains uncertainties too we may set its type (root mean square or mean square) as well. Sometimes the raw data is already preprocessed. E.g.

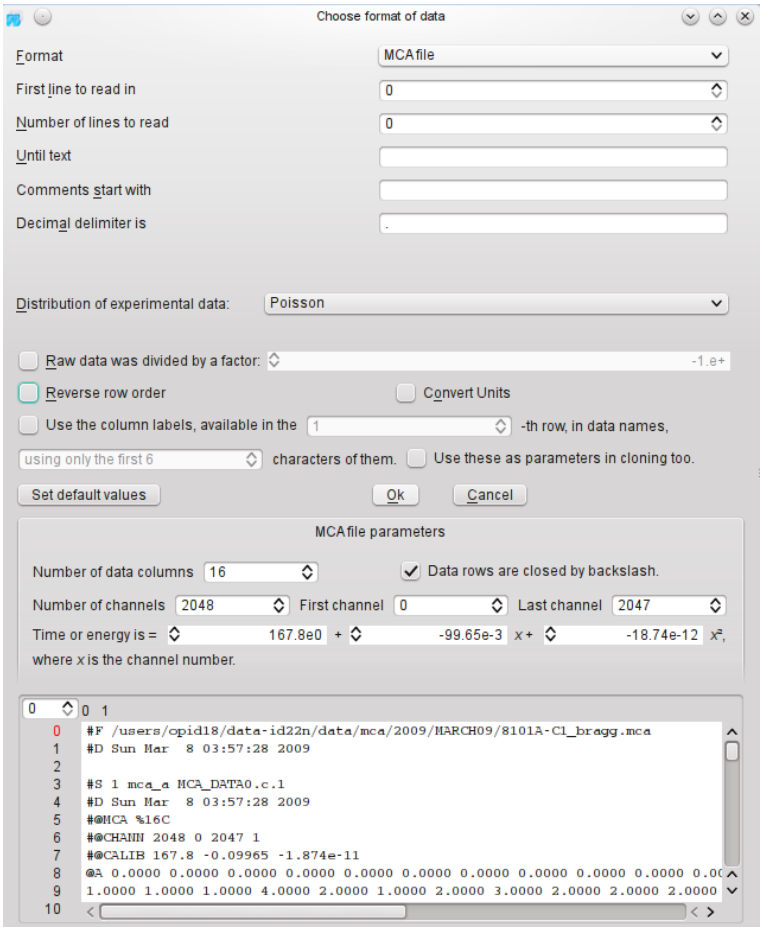

Figure 7: The dialog used to read in data sets in case of MCA file format.

neutron reflectometrists usually normalize their data, as they measure reflection, which has a maximal value of 1. The problem with this preprocessing is that in case of Poisson distribution we throw out this way information (see subsubsection 2.4). Therefore here you can tackle this problem by providing the normalization factor if there was such one. This piece of information may be provided later as well.

Right clicking in the window *Problems* on the name of the new dataset, we can add the data to the chosen model. At present, the user should know which format is required, accepted by a given type of model. If you choose a wrong one it will warn you only with malfunctioning or with segmentation fault error message. Data set may be replaced right clicking on the name of data set and clicking on Replace Data in the arising menu. This is useful if there was a mistake made by the user choosing, reading the data set, or after cloning (see later).

Some data points may be ex(in)cluded from the fit selecting with shift + cursor and right clicking. This can be used if you are sure that some values are badly measured. The exclusion is not working correctly for all the models at the moment, do not use it in case of Mössbauer and stroboscopy spectra. From version 1.0.3 on you may exclude data points according to their values as well.

The user may also add his(er) notes to the data after clicking on button **Notes** at the bottom of the data window.

The above mentioned statistical properties of the data set can be changed clicking on button **Statistical Properties** at the bottom of the data window. Here you can choose the distribution of the data set. Set the normalization factor, if there is such one (preprocessed data). Besides the user may choose the statistic, whose minimum has to be found by fitting the parameters, and the GOF (goodness of fit) statistic. For further details, see 2.4.

3.4 Changing parameters, matrices

From version 1.0.3 the parameters and transformation matrices are generated automatically. With Edit > Regenerate Matrices you may generate them, if there is some problem, or you want to set the transformation matrices starting from the initial ones. The parameters and the matrices may be changed with Edit > ... . For the program generated parameter (variable) names the following name convention is used:

- For the parameters of simple physical objects:
  ModelName=>*ObjectName*:>*PropertyName*::*ComponentName*.
  E.g.: FirstProblem=>SecondIncoherentFraction:>ExternalMagneticField::x.

- In case a linked objects (e.g.: domains in off-specular problem):
  Model=>*ParentObject*-~>*LinkedObject*:>*Property*::*Component*.
  E.g.: FirstProblem=>ithLayer-~>jthDomain:>size.

- In case of correlation function parameters of two linked objects:
  Model=>*FirstParent*-~>*FirstLinkedObject*,*SecondParent*-~>*SecondLinkedObject*>>
  FunctionTypeName:>*Property*::*Component*.
  E.g.:
  FirstProblem=>ithLayer-~>jthDomain,lthLayer-~>mthDomain>>Gauss:>sigma.

In **Parameter Editor** (Edit > Fitting Parameters), everything is included what is not integer independently on being constant or not. The parameters with check boxes (change them by double click) filled with 'x' or '√' (depending on system settings) denote the free parameters. The constants do not have check boxes, thus they cannot be freed. There are calibration constants, user defined constants

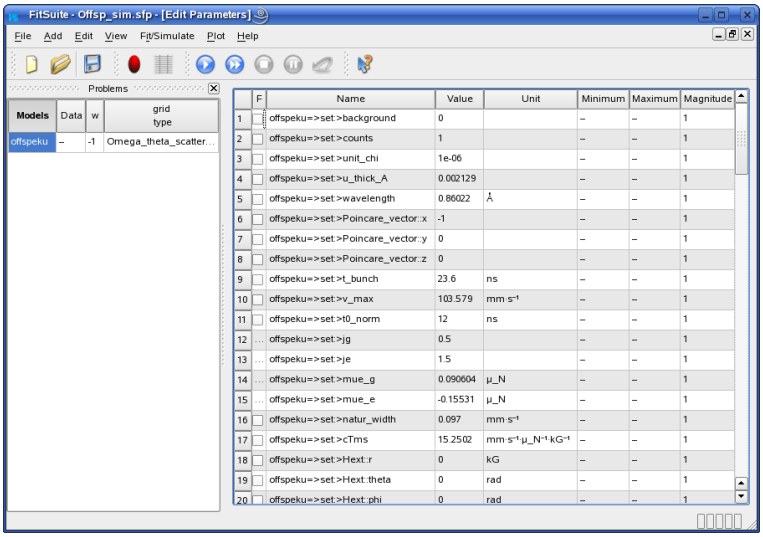

Figure 8: Parameter Editor

and model defined constants. Calibration constants have a label ‘ca’ instead of check box. They may be fitted right clicking on the check box or selecting the appropriate calibration constants (shift + cursor) + right clicking and selecting > Let Not Be Constant from the arising menu. The user may set arbitrary parameters to be constant, by choosing in this menu > Let Be Constant. If the parameter was not a calibration constant, the check box will replaced by ‘c’, these are the user defined constants. There may be constants, which are never fitted, these are denoted by label ‘cn’, these are the model defined constants. (see free/fix, make constant parameter demo) To increase the transparency of the parameter list the user may hide parameters, which (s)he thinks have the correct value and will not be fitted. This can be done similarly to previous operations, just a different menu item should be chosen. For obvious reasons free parameters may not be hidden and hidden parameters may not freed. The hidden parameters may be seen pressing the proper button (or menu item) appearing after parameters were hidden. The hidden parameters will appear with a different background color. This color may be changed in the Editor Settings (Settings > Editor Settings) (see hidding parameters demo).

From version 1.0.3 the parameters may have units. In older project files they will appear only after the command Edit > Regenerate Matrices was given for the program. (Sorrily with this the transformation matrices changed by the user will be lost and should be made again.) The units may be changed several ways. Just double clicking on the unit in the parameter editor, the unit may be changed. If the button with an arrow is pressed down, not only the unit is changed, but the

parameter value is also converted from the previous unit to the new one. Editing a parameter value with units, the unit may also be changed. In this case pressing down the button with arrow, the new unit and value is set, there is no conversion. If the button is not pressed down the unit remains the original one, but the value will correspond to the selected value and unit converted to the original one. (see parameter values and units demo) You may change the units of the minimum, maximum and magnitude (the latter is used to rescaling) values as well. If these units are identical with the unit of the parameter value, they are represented by shortcut ~. The user is able to specify a bit the behaviour of unit editor in Settings > Editor Settings (see editor settings demo).

You can get some information about the parameters by first clicking Help > What is this? or pressing Shift + F1, (on this the cursor icon should change to a question mark), clicking thereafter on the parameter name in the editor a short help should appear. Presently, this type of help is not complete, for some problems, e.g.: stroboscopic mode problems there is nothing available.

The displayed numbers in **Parameter Editor** and in **Transformation Matrix Editor** (Edit > T Matrices) also are rounded to a few digits. If a number is longer than that, the rounded number is displayed in blue (or other user set color) and we can see the real (not rounded) value by pulling the mouse over that cell in the editor and waiting until it appears in a tooltip. The user is able to specify the number of the displayed digits, choose the precision, what he needs and a lot of other options in Settings > Editor Settings.

In current version, the matrices can be united, split, and the parameters can be correlated, decorrelated and user defined parameters may be inserted (see subsection 2.2 and the demos of parameter correlation, decorrelation1, decorrelation2 and matrix split-unite).

The handling of integer parameters and the related integer transformation matrices may be handled quite similarly. The main difference is, that there we have to use the **Integer Parameter Editor** (Edit > Integer Parameters) and similarly the **Integer Transformation Matrix Editor** (Edit > Integer T Matrices).

3.5 Parameter filtering

From version 1.0.5 in the edit box at the top of the **Parameter Editor** (in **Integer Parameter Editor** too), the user may type in filter commands in order to view only the parameters (s)he is interested in momentarily instead of scrolling to and fro, looking hard for the parameters in a long list.

There are several simple filter commands available (see below) each is started by character @ and is of the form @$\mathcal{C}$ $\mathcal{A}$, where @$\mathcal{C}$ is the command word (which may not contain white spaces) e.g. @, @fr, @m, @mt (see below). The command word is followed by a space as separator and the commands argument, here

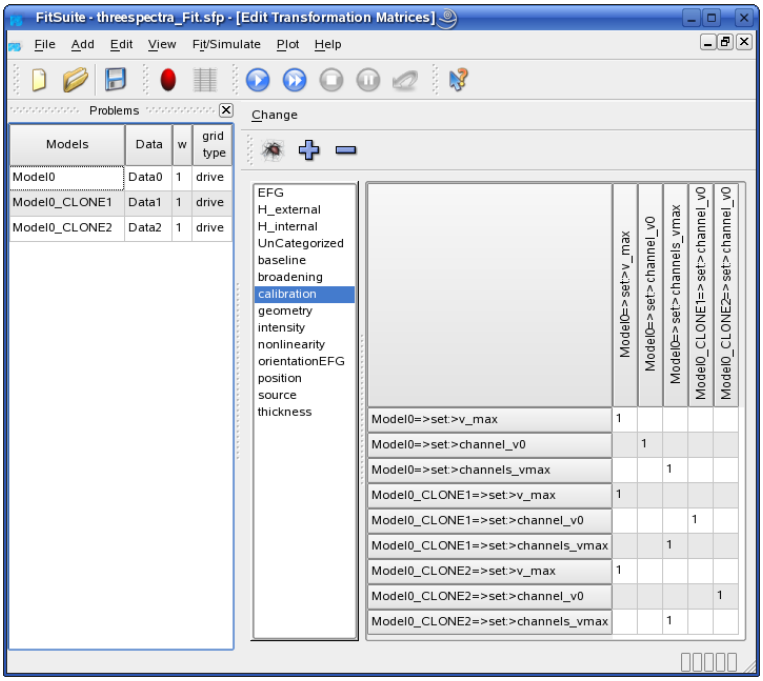

Figure 9: Transformation Matrix Editor

denoted by $\mathcal{A}$. The argument may be a single word or (as we will see) several words combined by logical operations.

3.5.1 Single word arguments, wildcards

Single word means, that it does not contain white spaces, but it may contain wildcards, which according to Unix convention are the following: ***** matches arbitray number ($\geqslant 0$) of characters, **?** matches a single character, $[\ldots]$ sets of characters. E.g.:

- **@p thickness** will filter all the parameters belonging to the properties named *thickness*,
- **@p thick*** will filter all the parameters belonging to the properties whose name begins with *thick*,
- **@p thick?ess** will filter all the parameters belonging to the properties whose name begins with *thick*, which is followed by a single arbitrary character and ends with *ess*,
- **@p thick[nml]ess** will filter all the parameters belonging to the properties whose name is *thickness* or *thickmess* or *thickless*.

Besides the filter editbox there are two checkboxes, one of them is labelled **'Strict**

pattern'. If this is not checked, which is the default case each of the single words typed in by the user are automatically complemented by wildcard * at the beginning and at the end, therefore the filtering condition is fulfilled even if the corresponding word of the argument is contained by the proper name. Therefore, in this case, we may type just e.g. **@p hick** instead of **@p *hick***. Just to avoid ambiguity we will assume strict pattern in this text. We may also set and unset this inside the command-line using the commands **setStrictPattern** (or just **setStriP**) and **unsetStrictPattern** (or just **unsetStriP**), respectively.

If the other checkbox labelled **Case sensitive** is checked, the **command argument** is filtered case sensitively. **The command word is always case sensitive.** We may also set and unset this inside the command-line using the commands **setCaseSensitive** (or just **setCaseS**) and **unsetCaseSensitive** (or just **unsetCaseS**), respectively.

3.5.2 Complex arguments, logical operators

Such 'single words' may be combined by logical operators: '**!**' denotes *negation*, ' ' (space) is interpreted as *or*, and '**&**' as *and* operator. '(', ')' parentheses may be used. In the explanation part of the following examples, squared parentheses are used to highlight the precedence of the logical operators (i.e. evaluate first the expressions inside parentheses, then execute *negations*, then *and* operations and at last the *or* operations), which otherwise may be not quite clear. Besides, we restrict the meaning of words 'and', 'or' to the sense of corresponding operations used in the mathematical logic, especially, if these words are highlighted using bold cases. E.g.:

- **@p *thickness* & !eff_thickness** will filter all the parameters belonging to the properties whose name contains *thickness*, **but not** the one called *eff_thickness*,
- **@p *thickness* *Hint*** will filter all the parameters belonging to the properties whose name contains *thickness*, **or** the one containing *Hint*. (If **'Strict pattern'** is checked we may also write instead of this **@p thickness Hint**, as each single word is complemented by two * wildcards.)

3.5.3 Combination of commands using logical operators

The commands may also be combined by these operations, e.g.

- **@fr *thick* & !*eff_thick* *Hi* & @m *Split*** means filter [the free parameters whose name [[contains the string *thick* **and** [does **not** contain *eff_thick*]] **or** contains *Hi*]] **and** [on which the models whose names contain *Split* depend].

- **@fr *thick* !*eff_thick* *Hi* @m *Split*** means filter the [free parameters whose name contains the string *thick* **or** [does **not** contain *eff_thick*] **or** contains *Hi*] **or** [all the parameters on which the models whose names contain *Split* depend].

A command ends when a new command @$\mathcal{C}_2$ $\mathcal{A}_2$ is started or when, in case of a command inside parentheses, the closing parenthesis is reached. Therefore and because the command @ is the parameter name filter command (see below), if the command word is missing automatically the parameter name filter (i.e. @) is inserted at the proper places, this is made internally, thus the user will see only in its effect. E.g.:

- **(abcd efgh) & ijkl** is interpreted as @ **(abcd efgh) & ijkl**,
- **(abcd efgh) & @m ijkl** is interpreted as @ **abcd efgh & @m ijkl**,
- **(@m abcd & efgh) ijkl** is interpreted as **(@m abcd & efgh) @ ijkl**,
- **(@m abcd (@fr) & efgh) ijkl** is interpreted as **(@m abcd @fr & @ efgh) @ ijkl**.

3.5.4 List of filter commands

Now, we will list commands words, which are filtering according different features of the parameters:

- @ is the parameter name filter. **Only in case of this command the @ character is optional if the text of the command starts with this.** E.g.: @ $\mathcal{A}_1$ @**p** $\mathcal{A}_2$ and $\mathcal{A}_1$ @**p** $\mathcal{A}_2$ and @**p** $\mathcal{A}_2$ @ $\mathcal{A}_1$ all have the same meaning, but in case of @**p** $\mathcal{A}_1$ $\mathcal{A}_2$ the argument $\mathcal{A}_2$ belongs to the filter command @**p** too it is just combined with $\mathcal{A}_1$ by the logical operation **or**. (see also the former subsubsection)
- @**m** is the model name filter. @*m* $\mathcal{A}$ filters the parameters on which the models whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- @**mg** is the model group name filter. @*mg* $\mathcal{A}$ filters the parameters on which the models of the modelgroups whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- @**mt** is the model type name filter. @*mt* $\mathcal{A}$ filters the parameters on which the models whose type name matches the condition given by the argument part $\mathcal{A}$ depend. E.g. if we have a project containing models of X-ray and

neutron reflectometry, with this filter we may choose the parameters on which the neutron problems depend using the command **@mt NeutronReflection**.

- **@smt** is the submodel type name filter. *@smt* $\mathcal{A}$ filters the parameters on which the submodels whose type name matches the condition given by the argument part $\mathcal{A}$ depend. We should note that if the argument of *@smt* matches the typename of a 'main model' the parameters of that will also appear in the list.
- **@o** is the object name filter. *@o* $\mathcal{A}$ filters the parameters on which the physical objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- **@ot** is the object type name filter. *@ot* $\mathcal{A}$ filters the parameters on which the physical object types whose name matches the condition given by the argument part $\mathcal{A}$ depend. E.g.: **@ot Layer** will filter the layer parameters.
- **@oc** is the object children filter. *@oc* $\mathcal{A}$ filters the parameters on which the physical objects contained by objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- **@op** is the object parent filter. *@op* $\mathcal{A}$ filters the parameters on which the physical objects containing the objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- **@od** is the object descendants filter. *@od* $\mathcal{A}$ filters the parameters on which the physical objects (and their descendant objects in the tree structure) contained by the objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- **@oa** is the object ancestors filter. *@oa* $\mathcal{A}$ filters the parameters on which the physical objects (an their ancestor objects in the tree structure) containing the objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.
- **@p** is the property name filter. *@p* $\mathcal{A}$ filters the simulation/fit parameters depending on model parameters belonging to a property whose name matches the condition given by the argument part $\mathcal{A}$. E.g.: **@p thickness** will filter the parameters on which the properties named *thickness* depend.
- **@pc** is property component name filter. *@p* $\mathcal{A}$ filters the simulation/fit parameters depending on model parameters belonging to a property component whose name matches the condition given by the argument part $\mathcal{A}$ depend. E.g.:

  - **@pc theta** will filter the parameters on which the *theta* components belonging to all the properties, which have such components, depend.
  - **@p Hint & @pc r** will filter the parameters on which the *r* components of the property Hint depend.
  - **@p chi & @pc .im** will filter the parameters on which the *.im* components (in this case the imaginary part) of the property *chi* depend.

- **@s** is the linked (symbolic) object name filter. *@s* $\mathcal{A}$ filters the parameters on which the linked physical objects whose name matches the condition given by the argument part $\mathcal{A}$ depend.

- **@st** is the linked (symbolic) object type name filter. *@st* $\mathcal{A}$ filters the parameters on which the linked physical object types whose name matches the condition given by the argument part $\mathcal{A}$ depend.
  E.g.: **@st Layer-~>Domain** or **@st Layer*Domain** will filter the parameters on which *layer-domain* linked objects depend.

- **@Cf** is the correlation function name filter. *@Cf* $\mathcal{A}$ filters the parameters on which the correlation functions whose name matches the condition given by the argument part $\mathcal{A}$ depend.

- **@T** is the transformation matrix name filter. *@T* $\mathcal{A}$ filters the parameters belonging to the transformation matrix whose name matches he condition given by the argument part $\mathcal{A}$. E.g.: **@T roughness** will filter al the simulation/fit parameters available in the matrix called *roughness*.

3.5.5 List of filter commands with optional arguments

The former commands without arguments give the whole parameter list, i.e have no effect, but there are commands which are meaningful without arguments too (e.g.: **@fr** will filter all the free parameters):

- **@og** is the object group filter. *@og* $\mathcal{A}$ filters the parameters on which the physical object groups whose name matches the condition given by the argument part $\mathcal{A}$ depend. Without arguments it matches all the parameters on which the object groups depend.

- **@fr** filters the free parameters. (This has no sense in case of integer parameters.)

- **@fi** filters the fix parameters. (This has no sense in case of integer parameters.)

- **@c** filters the constant parameters (calibration, model and user defined constants). (This has no sense in case of integer parameters.)
- **@ca** filters the calibration constants. (This has no sense in case of integer parameters.)
- **@cn** filters the model defined constants. (This has no sense in case of integer parameters.)
- **@cu** filters the user defined constants. (This has no sense in case of integer parameters.)
- **@u** filters the unit parameters (the parameters defining the units of other parameters, e.g.: , in case of X-ray reflectometries *unit_chi*, in case of neutron reflectometries *unit_Beff, unit_scatt_length_dens*, in case of Mössbauer effect related problems *u_thick_W, u_thick_A, natur_width*).
- **@d** filters the distributed parameters (the parameters which have distribution, this filters both the range and the midrange of the distribution). (This has no sense in case of integer parameters.)
- **@dm** filters the midranges of parameter distributions. (This has no sense in case of integer parameters.)
- **@dr** filters the ranges of parameter distributions. (This has no sense in case of integer parameters.)
- **@de** filters the decorrelated parameters (parameters which were recently decorrelated and which are still highlighted in the parameter list).
- **@Co** filters the correlated parameters (simulation/fit parameters on which several model parameters depend).
- **@in** filters the inserted parameters (parameters which were recently inserted by the user and which are still highlighted in the parameter list).
- **@gd** filters the grid parameters (parameters determining the grid, i.e. the set of the independent variable points for which the simulation is performed).
- **@cugd** filters the currently used grid parameters. **@gd** filters parameters for all the possible grid types. **@cugd** filters only the parameters belonging to the currently used grid.
- **@hi** filters the hidden parameters, which may be hidden by the user; or disabled parameters hidden by the program if the corresponding option is chosen in Editor Settings (Settings » Editor Settings).

- **@hu** filters the parameters hidden by the user.
- **@di** filters the disabled parameters. The disabled parameters do not have effect on the simulation result. Disabled parameters may arise for several reason in the program. E.g.:
  - We may have switches in our model which influences whether some parameters have a role or not in calculations. This sort of disabled parameters may become enabled changing the corresponding switch.
  - Having a type of physical object as a child which renders some properties redundant, as in simulations we use the properties of this (these) type of child(ren). This sort of disabled parameters may become enabled removing all instances of the corresponding object type.
- **@rc** filters the parameters changed by the user recently (since the last calculation).

3.5.6 Complex examples

- ***Hint* @fr *thick* & !*eff_thick*** filters [the parameters whose name contains *Hint*] **or** [the free parameters whose name contains *thick*, **and not** *eff_thick*].

3.6 Command-line interface

At the bottom of the main window there is a command-line interface. This may be useful in case of large projects, where the graphical user interface sometimes needed too much clicking. This command-line interface uses extensively the parameter filters shown in the previous subsection. The arguments of the commands are separated by the # character.

The commands may have options. The options are given by single characters following the command and a colon, e.g.: *list:IR*, where *list* is a command and *I* and *R* are two options, or by keywords separated from each other by colons and from the command by two colons. E.g. *list:IR*, *list:::I:R list:::Integer:Real*, *list::I:Real*, *list:I:Real*, *list:R:Integer* are all equivalent, but for the incorrect commands *list::IR* and *list::IntegerReal* we will have a message of unknown option *IR* and *IntegerReal*, and similarly in case of *list:Real* we will have a messages of unknown options *e* and *l*. A command may have default options, which are used if the user gives the optionless command.

The default options will be denoted by placing them inside [] brackets in the following command descriptions, and the optional options in () brackets. In case

of some commands we cannot use some of the options together, such exclusive options will be separated by $\oplus$ in the description.

We have the following options, where the character for the option is highlighted with boldface:

- **R***eal* $\Leftrightarrow$ execute the commands for the real parameters only,
- **I***nteger* $\Leftrightarrow$ execute the commands for the integer parameters only, which are not group repetition numbers,
- **G***roup* $\Leftrightarrow$ execute the commands for the group repetition number parameters only,
- **A***ll* $\Leftrightarrow$ execute the commands for the real and integer based and group repetition number parameters, this is equivalent to the options *:IGR* (or *::Integer:Group:Real*),
- **f***orce* $\Leftrightarrow$ force execution of the command. This is useful, in cases where the command may ask, warn the user whether (s)he wants really execute the command, e.g. because a file would be overwritten,
- **a***ppend* $\Leftrightarrow$ append to the end of the file the result of the command (e.g. the commands *export* and *exportMP*),
- *re***v***erse* $\Leftrightarrow$ reverses the order in which the results are written in a file (e.g. the commands *export* and *exportMP*).

The interface is still not complete, we have a lot of possible commands still in mind. Currently we have the following commands:

- **list(:[R]IGA)** (or just **li**) $\mathcal{F}$ lists the parameters which match the parameter filter $\mathcal{F}$ given as an argument. E.g. *li @fr* will list the free parameters, *li* or *li:R* will list all the real parameters, *li:I xx* will list the integer parameters containing *xx*.
- **correlate(:[R]$\oplus$I$\oplus$G)** (or just **corr**) $\mathcal{F}$ # (optional)$\mathcal{S}$ correlates the parameters which match the parameter filter $\mathcal{F}$ given as first argument, and optionally if the second argument is available sets the name $\mathcal{S}$ of the new parameter. If '# $\mathcal{S}$' is missing, the name of the new parameter will be the name one of the correlated parameters. E.g. *corr Fe & thickness & @mg X* will correlate the parameters, whose name contains the string 'Fe' and 'thickness' and belong to model groups whose name contains X. *corr Fe & thickness & @mg X # ABCD* will do the same, but the new parameter will have the name 'ABCD'. *corr:I C_nzn & @mg X # F* will correlate integer number based parameters.
- **decorrelate(:[R]$\oplus$I$\oplus$G)** (or just **decorr**) $\mathcal{F}_1$ # (optional)$\mathcal{F}_2$ decorrelates the simulation/fit parameters which match the parameter filter $\mathcal{F}_1$ given as first argument. Optionally we may specify further by adding a second argument, a

model parameter filter $\mathcal{F}_2$ according to which we may choose the model parameters (which depend on the fit parameters given by $\mathcal{F}_1$). E.g. *decorr Fe & thickness & @mg X* will decorrelate the fit parameters, whose name contains the string 'Fe' and 'thickness' and belong to model groups whose name contains X. By ordering *decorr Fe & thickness & @mg X # Fe1* we will have probably less new simulation/fit parameters, as we choose only those model parameters whose name contains the string 'Fe1' to become fit parameters.

• **hide(:[R]IGA)** $\mathcal{F}$ hides the parameters matching the filtering condition $\mathcal{F}$ (parameter distribution range parameters and free parameters may not be hidden).

• **unhide(:[R]IGA)** (or **!hide**) $\mathcal{F}$ unhides the parameters matching the filtering condition $\mathcal{F}$.

• **free(:[R])** $\mathcal{F}$ frees the parameters matching the filtering condition $\mathcal{F}$. Only real parameters may be fixed or freed.

• **fix(:[R])** $\mathcal{F}$ fixes the parameters matching the filtering condition $\mathcal{F}$.

• **setConstant(:[R])** (or just **setCon**) $\mathcal{F}$ sets constant the parameters matching the filtering condition $\mathcal{F}$.

• **setVariable(:[R])** (or just **setVar**) $\mathcal{F}$ sets variable (the inverse of *setConstant*) the parameters matching the filtering condition $\mathcal{F}$.

• **export(:[R]IGAfav)** (optional)$\mathcal{N}$ # (optional)$\mathcal{F}$ {(optional)# $\mathcal{S}_i$ # $\mathcal{S}_{i+1}$} exports the simulation/fit parameters specified by the optional parameter filter $\mathcal{F}$ in the file given by its path and name in the optional argument $\mathcal{N}$. The argument $\mathcal{F}$ may be followed by arbitrary number of argument pairs $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$. In exported file the regular expressions using wildcards according unix usage[7] $\mathcal{S}_i$ will be replaced by the string $\mathcal{S}_{i+1}$. E.g.:

– *export results/trial.txt # (thickness Hint rough) & !Ni # X*g[hj]?X # y # Fe # iron* will export the real parameters whose names contain *thickness* or *Hint* or *rough*, but not *Ni* in the file *results/trial.txt*, and in the file the regular expressions *X*g[hj]?X* will be replaced by *y* and *Fe* by *iron*,

– *export:I # (thickness Hint rough) & !Ni # X*g[hj]?X # y # Fe # iron* will do the same, but for integer parameters and the file name is chosen by the program, it will be like *ExportedSimFitParameters_0.txt*.

[7] In these regular expressions:
* An asterisk matches any number of characters in a name, including none.
? The question mark matches any single character.
[] Brackets enclose a set of characters, any one of which may match a single character at that position.
- A hyphen used within [] denotes a range of characters.
\ The character backslash escapes the wildcard.

– *export # # X*g[hj]?X # y # Fe # iron* will export all the real parameters in a program chosen file, and it will make the same text replacements.

– *export:aAv result/trial.txt* will export all (option *A*) the (real and integer and group repetition number) parameters in *results/trial.txt* without text replacements. If the file exists it will append (option *a*) to the end of the file and not overwrite its previous content. And the parameters will be exported in reversed order (option *v*).

• **exportModelParameters(:[R]IGAfav)** (or just **exportMP**) (optional)$\mathcal{N}$ # (optional)$\mathcal{M}$ will export the model parameters specified by the optional model parameter filter $\mathcal{M}$ into the file given by its path and name in the optional argument $\mathcal{N}$.

• **exportModelParameterTable** (or just **exportMPT**) (optional)$\mathcal{N}$ # (optional) $\mathcal{M}$ # (optional)$\mathcal{P}_2$ # (optional)$\mathcal{U}_2$ # ... (optional)$\mathcal{P}_n$ # (optional)$\mathcal{U}_n$ will export parameters into the file given by its path and name in the optional argument $\mathcal{N}$ in a tabular form in which each line belongs to a different model, and the first column contains the name of that model. These models are specified by the 'model filter' $\mathcal{M}$ (If $\mathcal{M}$ is an empty string then all the models. A model filter may contain only the combinations of filter commands @m, @mt, @mg. @ is replaced by internally by @m). The argument pairs $\mathcal{P}_i$ # $\mathcal{U}_i$ specify the contents of the i-th column of the table. $\mathcal{P}_i$ is a parameter filter which should correspond to a single parameter in each model (or a statistic, or mathematical expression of parameter filters, about these we will write below). It is assumed that it corresponds to a real model parameter, and only if there is no such parameter, then it is checked whether there is such a simulation parameter satisfying the filtering condition. In $\mathcal{U}_i$, we can specify the unit in which the parameter value is to be exported, and some other options which we will elaborate below. E.g. *exportMPT myplace/text1.txt # abc & !G # Fe*rough # nm # Si*rough # A* will result a file *text1.txt* in the directory *myplace*. This file will contain a table like

Model	#Fe*rough (nm)	#Si*rough (A)
abc1	5	2
Xabc	6	1
abcDD	0.7	3

As we can see each model name listed in the first column satisfies the filtering conditon *abc & !G*, which in this case as it is a model filter is equivalent to *@m abc & !G*.

In ($\mathcal{U}_i$ after the unit and a colon (:), we may have additional options specifying whether we want to export simulation (S) or model parameters (M), real (R) or integer number based parameters (I), or repetition group numbers (G). E.g. in case of *exportMPT myplace/text1.txt # abc & !G # Fe*rough # nm : S # Si*rough*

*# A: M # s1*Cny_n # :I # Fe*thick # nm : SMR* the first parameter is requested as a simulation parameter the second as a model parameter and the third one is an integer parameter. The option *:SMR* in the last parameter means first try as a real model parameter if there is no such one, then try to find such a real simulation parameter. This is the default option, therefore *# Fe*thick # nm : SMR* and *# Fe*thick # nm* is equivalent. A parameter cannot be integer and real number or group repetition number simultaneously, therefore only one of *R, I* or *G* may appear in such an expression, and *R* is the default setting. Therefore *:S* is equivalent to *:SR*, and similarly *:M* is equivalent to *:MR* and *:MS* (or *:SM*) is equivalent to *:SMR*. For similar reason *:R* is equivalent to *:SMR* and *:I* to *:SMI*.

As it was mentioned the arguments $\mathcal{P}_i$ may not only parameter filters, but they may be statistics of the corresponding models. E.g. *exportMPT myplace/-text1.txt # abc & !G # Fe*rough # nm # $stat FI # # $stat DOF* will write out in the third and fourth columns the *Fitted statistic* and the *degrees of freedom* for each model. (To have correct values, we have to calculate the statistics choosing the proper item from the menu Fit/Simulate, before trying to export the values.) Similarly we can write out the *goodness of fit statistic* using *$stat GOF*, and the corresponding reduced statistics using *$stat RFI* and/or *$stat RGOF*.

Mathematical expressions are also possible, having a *$math* prefix followed by a possible argument of a math command (see subsection 3.7) resulting a scalar value. E.g. the command *exportMPT text1.txt # abc & !G # $math sum($(Si*:>thickness) [nm]) # nm* can be used write out for each model the sums of the thicknesses of the layers, the name of which contain *Si*, in nm. Currently, it is a restriction that in such expressions the arguments of the parameter filter functions $(\mathcal{A})$ are replaced internally for each filtered model, i.e. for each row of the exported file by $(\mathcal{A}$ & @m $\mathcal{M})$, where $\mathcal{M}$ is the name of the model. Therefore, we may not have expressions depending on parameters of different models.

• **plotModelParameterTable** (or just **plotMPT**) (optional)$\mathcal{N}$ # (optional) $\mathcal{M}$ # (optional)$\mathcal{P}_2$ # (optional)$\mathcal{U}_2$ # . . . (optional)$\mathcal{P}_n$ # (optional)$\mathcal{U}_n$ is similar to exportModelParameterTable, but here we do not export, but plot the filtered parameters. The argumentation is very similar, therefore here we describe only the differences. $\mathcal{N}$ is not a file name, but a label which will appear in the title bar of the plot window. The first parameter will be the independent variable of the plot, and the others will be the dependent variables. To be able to plot some of the dependent variables with a common vertical axis we may specify with an additional option an integer number in $\mathcal{U}_i$. Furthermore we can specify the axis title of the dependent variables, and the curve label as well. E.g. *plotMPT myplot # abc & !G # Fe*thick # nm : S::Thickness (nm) # Si*rough # nm: M : 2 : Roughness : r_Si # Fe*rough # nm: M : 2 :: r_Fe # s1*Cny_n # :I:5:Symmetry*

*# s2*Cnz_n # :I:5* will have two dependent variable axes, one titled *Roughness* and one *Symmetry*, and the independent variable axis will be titled *Thickness (nm)*, furthermore the two curves belonging to *Si*rough* and *Fe*rough* will be labelled by *r_Si* and *r_Fe*, respectively.

• **setValue(:[R]IG)** (or just **setVal**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ sets the value of the parameters matching the filtering condition $\mathcal{F}$ to $\mathcal{V}$ in units $\mathcal{U}$. E.g. *setVal Fe*thickness & @fr # 5 nm* sets the free real parameters of which name contains *Fe*thickness* to 5 nm, *setVal Hint & (theta phi) # 65 deg* sets the real parameters of which name contains *Hint* and (*theta* or *phi*) to 65 ° *setValue graz_beg # 1 mrad* sets the parameter of which name contains *graz_beg* to 1 mrad, *setValue Diff_coeff # 1 nm^2s^-1* sets the parameter of which name contains *Diff_coeff* to 1 $\mathrm{nm}^2 \cdot \mathrm{s}^{-1}$, *setVal:I flag # 3* sets the integer parameter of which name contains flag to 3.

Units are not needed in case of dimensionless parameters, therefore in case of integer parameters, and if the filtered parameters may have only a single unit (SI prefix included), i.e. when the unit is fully determined. The filtered parameters should belong to the same *unit group* (A unit group is similar to a physical dimension. Only units of the same group can be converted into each other by the program). If that is not the case the program will not execute command, and will return with an error message.

In case of complex units (like $\mathrm{nm}^2 \cdot \mathrm{s}^{-1}$) it may not be straightforward for the user what (s)he (nm^2s^-1) should type in or whether the filtered parameters belong to the same unit group, therefore there is a command listUnitsOf which provides help in such cases.

• **listUnitsOf(:[R])** $\mathcal{F}$ lists for the parameters matching the filtering condition $\mathcal{F}$ all the unit groups and the corresponding units providing their ASCII and UNICODE names (e.g. meter) and symbols (e.g. m), and whether SI prefix is or is not allowed to use. On the output the user can see something like the following:
The unit group Angle (used by the parameters H0T=>src:>Hext::theta, H0T=>src:>Hext::phi) may have the following units:

- *ASCII: deg ♦ degree*
- *ASCII/UNICODE: rad ♦ radian*
- *UNICODE: ° ♦ degree*
- *SI prefix is not allowed*
- *SI prefix is allowed*

It is not too practical, but the user may also use the unit names in the input e.g. *setVal thickness # 5 nanometer* (but not nmeter). The user may use the unicode symbols (and names, some of which nowadays can be accessed quite easily on keyboards) e.g. *setVal theta # 5 °*, but it may happen, that there are several quite similar unicode characters, in which case the user may choose the wrong one

and may not understand what the problem is. In such cases the use of ASCII symbol or name may be the safe solution.

• **setMinimum(:[R])** (or just **setMin**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ sets the minimum value of the parameters matching the filtering condition $\mathcal{F}$ to $\mathcal{V}$ in units $\mathcal{U}$. It is similar to **setValue**, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **setMaximum(:[R])** (or just **setMax**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ sets the maximum value of the parameters matching the filtering condition $\mathcal{F}$ to $\mathcal{V}$ in units $\mathcal{U}$. It is similar to **setValue**, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **setMagnitude(:[R])** (or just **setMag**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ sets the magnitude value of the parameters matching the filtering condition $\mathcal{F}$ to $\mathcal{V}$ in units $\mathcal{U}$. It is similar to **setValue**, a difference is that this cannot be used in case of integer parameters.

• **setResolution(:[R])** (or just **setRes**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ sets the resolution value of the parameters matching the filtering condition $\mathcal{F}$ to $\mathcal{V}$ in units $\mathcal{U}$. It is similar to **setValue**, a difference is that this cannot be used in case of integer parameters.

• **addToValue(:[R]IG)** (or just **addToVal**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ changes the value of the parameters matching the filtering condition $\mathcal{F}$ by adding $\mathcal{V}$ in units $\mathcal{U}$. E.g. *addToVal Fe*thickness & @fr # 5 nm* adds 5 nm to the free real parameters of which name contains *Fe*thickness*, *addToVal Hint & (theta phi) # 65 deg* adds 65 ° to the real parameters of which name contains *Hint* and (*theta* or *phi*)), *addToVal:I flag # 3* adds 3 to the integer parameters of which name contains *flag*.

Units are not needed in case of dimensionless parameters, therefore in case of integer parameters, and if the filtered parameters may have only a single unit (SI prefix included), i.e. when the unit is fully determined. The filtered parameters should belong to the same *unit group* (A unit group is similar to a physical dimension. Only units of the same group can be converted into each other by the program). If that is not the case the program will not execute command, and will return with an error message.

In case of complete units (like $\mathrm{nm}^2 \cdot \mathrm{s}^{-1}$) it may not be straightforward for the user what (s)he (nm^2s^-1) should type in or whether the filtered parameters belong to the same unit group, therefore there is a command listUnitsOf which provides help in such cases.

• **addToMinimum(:[R])** (or just **addToMin**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ adds $\mathcal{V}$ in units $\mathcal{U}$ to the minimum value of the parameters matching the filtering condition $\mathcal{F}$. It is similar to addToValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **addToMaximum(:[R])** (or just **addToMax**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ adds $\mathcal{V}$ in units $\mathcal{U}$ to the maximum value of the parameters matching the filtering condition $\mathcal{F}$. It is similar to addToValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **addToMagnitude(:[R])** (or just **addToMag**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ adds $\mathcal{V}$ in units $\mathcal{U}$ to the magnitude value of the parameters matching the filtering condition $\mathcal{F}$. It is similar to addToValue, a difference is that this cannot be used in case of integer parameters.

• **addToResolution(:[R])** (or just **addToRes**) $\mathcal{F}$ # $\mathcal{V}$ $\mathcal{U}$ adds $\mathcal{V}$ in units $\mathcal{U}$ to the resolution value of the parameters matching the filtering condition $\mathcal{F}$. It is similar to addToValue, a difference is that this cannot be used in case of integer parameters.

• **multiplyValue(:[R]IG)** (or just **mulVal**) $\mathcal{F}$ # $\mathcal{V}$ changes the value of the parameters matching the filtering condition $\mathcal{F}$ by multiplying by the dimensionless scalar $\mathcal{V}$. E.g. *multiplyValue Fe*thickness & @fr # 5* multiplies the free real parameters of which name contains *Fe*thickness* by 5, *mulVal:I flag # 3* multiplies the integer parameters of which name contains *flag* by 3.

• **multiplyMinimum(:[R])** or just **mulMin**) $\mathcal{F}$ # $\mathcal{V}$ multiplies the minimum value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to multiplyValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **multiplyMaximum(:[R])** (or just **mulMax**) $\mathcal{F}$ # $\mathcal{V}$ multiplies the maximum value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to multiplyValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **multiplyMagnitude(:[R])** (or just **mulMag**) $\mathcal{F}$ # $\mathcal{V}$ multiplies the magnitude value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to multiplyValue, a difference is that this cannot be used in case of integer parameters.

• **multiplyResolution(:[R])** (or just **mulRes**) $\mathcal{F}$ # $\mathcal{V}$ multiplies the resolution value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to multiplyValue, a difference is that this cannot be used in case of integer parameters.

• **divideValue(:[R]IG)** (or just **divVal**) $\mathcal{F}$ # $\mathcal{V}$ changes the value of the parameters matching the filtering condition $\mathcal{F}$ by dividing by the dimensionless scalar

$\mathcal{V}$. E.g. *divideValue Fe*thickness & @fr # 5* divides the free real parameters of which name contains *Fe*thickness* by 5, *divVal:I flag # 3* divides the integer parameters of which name contains *flag* by 3.

• **divideMinimum(:[R])** or just **divMin**) $\mathcal{F}$ # $\mathcal{V}$ divides the minimum value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to divideValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **divideMaximum(:[R])** (or just **divMax**) $\mathcal{F}$ # $\mathcal{V}$ divides the maximum value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to divideValue, a difference is that this cannot be used in case of integer parameters as the integer minimum and maximum values cannot be changed by the user.

• **divideMagnitude(:[R])** (or just **divMag**) $\mathcal{F}$ # $\mathcal{V}$ divides the magnitude value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to divideValue, a difference is that this cannot be used in case of integer parameters.

• **divideResolution(:[R])** (or just **divRes**) $\mathcal{F}$ # $\mathcal{V}$ divides the resolution value of the parameters matching the filtering condition $\mathcal{F}$ by $\mathcal{V}$. It is similar to divideValue, a difference is that this cannot be used in case of integer parameters.

• **setToAbsoluteValue(:[R]IG)** (or just **setAbsVal**) $\mathcal{F}$ replaces the value of the parameters matching the filtering condition $\mathcal{F}$ by their absolute values.

• **swapParameterForMatrixDiagonal(:[R])** (or just **swapParMatrDiag**) $\mathcal{F}$ swaps the value of the parameters matching the filtering condition $\mathcal{F}$ for the corresponding diagonal transformation matrix elements. This may be useful, if the ratio of some parameters is known and we would like to fit their common multiplication factor.

• **setMatrixElement(:[R]IG)** (or just **setME**) $\mathcal{X}$ # $\mathcal{M}$ # $\mathcal{S}$ # $\mathcal{V}$ sets the element of the matrix the name of which is filtered by $\mathcal{X}$ and belongs to the row of model parameter filter $\mathcal{M}$ and to the column of the simulation/fit parameter $\mathcal{S}$ to $\mathcal{V}$. The filters should have a single match, i.e. it is not possible to set several matrix elements simultaneously. We allow filters instead of the matrix, row and column names only to spare users (including ourselves) of unnecessary typing. E.g. *setMatrixElement thick # Fe*thickness # Ag*thick # -1* will work only if only one model parameter corresponds to *Fe*thickness* and only one simulation/fit parameter to *Ag*thick*, and there is only a single real transformation matrix the name of which contains the string *thick*. The $\mathcal{V}$ may be a mathematical expression, for further details see the corresponding part in the text explaining the *setMatrixPartialColumn* command on page 54. The only difference is, that here only a single element is set, therefore *$x* will be always 0, thus it has no use.

• **setMatrixPartialColumn(:[R]IG)** (or just **setMPC**) $\mathcal{X}$ # $\mathcal{M}$ # $\mathcal{S}$ # $\mathcal{V}$ sets some of the elements of the matrix the name of which is filtered by $\mathcal{X}$ and belong to the rows of model parameter filter $\mathcal{M}$ and to the column of a single simulation/fit parameter $\mathcal{S}$ to $\mathcal{V}$. The filters should match elements of a single column of a single matrix. We allow filters instead of the matrix, and column names only to spare users (including ourselves) of unnecessary typing. E.g. *setMatrixPartialColumn thick # Fe*thickness # Ag*thick # -1* will work only if only one simulation/fit parameter corresponds to *Ag*thick*, and there is only a single real transformation matrix the name of which contains the string *thick*. The $\mathcal{V}$ may be a mathematical expression having a *$math* prefix followed by a possible argument of a math command (see subsection 3.7) resulting a scalar value, the index of the filtered matrix element starting with 0 should be refered to as *$x*. E.g. *setMPC rough # del0*Ni*rough # Ni*nov # $math 40*exp(-$x)* will set the filtered matrix elements to $40 * (\mathrm{e}^0, e^{-1}, e^{-2}, \ldots, e^{-(D-1)})$, where D is the number of the matrix elements to be set.

• **insertSP(:[R]IG)** (or just **ins**) $\mathcal{S}$ # $\mathcal{X}$ # (Optional) $\mathcal{B}$ # (Optional) $\mathcal{F}$ # (Optional) $\mathcal{M}$ will try to insert a new simulation/fit parameter named $\mathcal{S}$ in the transformation matrix the name of which is filtered by $\mathcal{X}$. Optionally we can specify by filter $\mathcal{B}$ the parameter before which the new parameter will be inserted, otherwise it will append after the last parameter belonging to the matrix. And we can specify optionally a simulation/fit parameter $\mathcal{F}$ or model parameter $\mathcal{M}$ according to which the value and unit of the new parameter will be set. If both are given, then only $\mathcal{F}$ is used.

• **renameModel** (or just **renMod**) $\mathcal{O}$ # $\mathcal{N}$ will try to rename the model named $\mathcal{O}$ to $\mathcal{N}$, here we use names and not filters.

• **connectModelData** (or just **conMod**) $\mathcal{O}$ # $\mathcal{D}$ will try to connect the model named $\mathcal{O}$ to the data named $\mathcal{D}$, here we use names and not filters.

• **addModelToGroup** (or just **addModGr**) $\mathcal{G}$ # $\mathcal{M}$ will try to add the models filtered by $\mathcal{M}$ to the model group named $\mathcal{G}$. If there is no such group, then it tries to create it as well.

• **excludeDataValues** (or just **excDaV**) $\mathcal{D}$ # $\mathcal{C}_1\mathcal{I}_1 \ldots \mathcal{C}_n\mathcal{I}_n$ excludes the points of the data set named $\mathcal{D}$ specified the other arguments. $\mathcal{C}_i$ can be *x* or *x1* or *x2* or *y* specifying the data set column to which interval $\mathcal{I}_i$ belongs to. The intervals can be started by '(' (or '[') in case of left-open (left-closed) and ended by ')' (or ']') in case of right-open (right-closed) intervals. Inside we may have two numbers separated by a colon. If the first (second) number is missing, then the interval is left(right)-unbounded. The program excludes the points inside the hyperrectangles stretched by all the possible combination of these intervals. E.g. *excDaV Data1 # x[1.3:52.6] x(73:75] y[:0] y[2.e6:3.12e7]* will exclude

the points inside the hyperrectangles: $[1.3 : 52.6] \times [: 0], [1.3 : 52.6] \times [2.e6 : 3.12e7], (73 : 75] \times [: 0], (73 : 75] \times [2.e6 : 3.12e7]$.

- **includeDataValues** (or just **incDaV**) It is the reverse of excludeDataValues, it has the same argumentation.
- **help** (or just **he**) $\mathcal{T}$ shows help about the commands of which name contains the text $\mathcal{T}$, and opens the User Manual at the first such command.
- **math** is used for several operations containing mathematical expressions. For further details, see subsection 3.7
- **simulate** (or just **sim**) $\mathcal{M}$ will try to simulate the model(s) filtered by $\mathcal{M}$.
- **calculateStatistics** (or just **calStat**) $\mathcal{M}$ will try to calculate statistics for the model(s) filtered by $\mathcal{M}$.

3.7 The 'math' command

The *math* command is one of the commands of the *Command-line interface* introduced in subsection 3.6, which is described in details here. It is used for several operations containing mathematical expressions, which may contain operators +, -, *, / and ^ (exponentiation); parentheses *(,)*; functions like *abs, sgn, floor, ceil, round, sin, cos, tan, cot, asin, acos, atan, sinh, cosh, tanh, coth, asinh, acosh, atanh, ln, log, log10, exp, pow, erf, cyl_bessel_j, cyl_neumann, cyl_bessel_i, cyl_bessel_k, etc.;* parameter filters inside 'functions' *$()*; and (real, integer) numbers.

The math command can be used for:

- **Evaluation of mathematical expressions,** e.g.:
  - *math 3*2+2^3* will result 14.
  - *math 2*$(Cr*thick)[nm]* will return the parameters which are filtered by the parameter filter *Cr*thick* in nanometers multiplied by 2;
  - *math 2 * $pi / $(wavelength)[A] * $(Cr*thick)[A]* will return 2π divided by the parameter *wavelength* in ångström and multiplied by the parameters which are filtered by the parameter filter *Cr*thick* in ångströms.
  - *math sum($(Cr*thick))* will return the sum of the parameters which are filtered by the parameter filter *Cr*thick* each in its current unit, for example if we have two parameters filtered *Cr1thickness* = 2 nm and *Cr2thickness* = 5 Å the result will be 7. To have the correct result we have to specify the unit too, like *math sum($(Cr*thick)[nm])* which will result the correctly 2.5 (nanometer).

Currently we have four mathematical constants *$pi = $π = π*, *$2pi = $tau = $τ = 2π = τ*, the Euler number *$e* = e ≈ 2.718281828459 and the Euler–Mascheroni constant *$EulerMascheroni = $γ = γ* ≈ 0.5772156649015. These constants are represented by double precision floating point numbers, therefore do not be surprised if *math sin($pi)* will not return 0, (as some sophisticated mathematical software would do) but something like 1.22465e-16.

As we can see in the above mentioned examples the parameter values can be referred to by the parameter filters written inside the parentheses of the expression *$()*, similarly we can refer to the minimum of the parameters by *$<()*, to the maximum by *$>()*, to the magnitude by *$|()* (| is the vertical bar character) and to the resolution by *$.()*.

- **Definition of mathematical functions,** e.g.:

  – *math sinc(x) := if(lt(abs($x),1e-100), 1, sin($x)/$x)* defines the *sinus cardinalis* $\mathrm{sinc}(x) = \frac{\sin x}{x}$ function. (**Currently we cannot save a function, the program will remember the definition, until it is not closed.**)

  – *math aver(xx, y) := ($xx + $y)/2.* will define a function with two variables calculating the arithmetical mean of the two variables.

The functions may have several variables. The names of the variables should be separated by comma on the left side of the definition (:=), and prefixed by $ on the right side. As $pi, $e, $tau are predefined constants, pi, e, tau (and π, τ) may not be used as variable names.

- **Assignment of parameter values,** e.g.:

  – *math $(Cr*roug)[nm] =: $(Cr*thick) [nm]/10* will set the roughnesses of the layers corresponding to the parameter filter to the tenth part of the corresponding thicknesses.

  The parameter assignment in most general case has the form

  $$\textit{math } \$(\mathcal{F})[w][u] =: \mathcal{M},$$

where $\mathcal{F}$ is the filter specifying the parameters the values of which we would like to set according to the results of the mathemathical expression $\mathcal{M}$ on the right side of the assignment operator (=:); *w* and *u* are optional, both specify unit names, the same way as in the case of the command setValue, therefore for further details about the unit names please see the help of setValue and listUnitsOf. *u* is the name of the unit in which the result of the expression $\mathcal{M}$ is expected. *w* is the name of the unit to which the parameters filtered by $\mathcal{F}$ are to be set after the assignment. The expression **$($\mathcal{F}$)[u][u]** has an equivalent shorter form **$($\mathcal{F}$)[*u]**. **$($\mathcal{F}$)[u]** is equivalent to **$($\mathcal{F}$)[][u]** and not **$($\mathcal{F}$)[u][]**. If there

are no units specified, then the program assumes, that the result of $\mathcal{M}$ will be in the default units of the parameters to be set, and those values should be converted to the current units. We will see, how this works, by examples below. If we would like to set the minimum, maximum, magnitude, resolution values of the parameters then instead of $(\mathcal{F})$ we should use $<(\mathcal{F}), $>(\mathcal{F}), $|(\mathcal{F}), $.(\mathcal{F}) respectively.

As the unit conversion may appear a bit confuse at first look, let explain it by some examples. Let have three parameters filtered by *Cr*rough*

Name	...	Magnitude	Unit	Resolution	Unit
Cr1roughness		5	Å	0	Å
Cr2roughness	...	6	Å	0	nm
Cr3roughness		0.7	nm	0	Å

and let see what results with different commands, parameter assignments using this filter:

- The command *math $|(Cr*rough)* will result (5, 6, 0.7), as if we do not specify any units it takes the magnitude parameter values in their current units.
- The command *math $|(Cr*rough) [nm]* will result (0.5, 0.6, 0.7), as the magnitude values are asked in nanometers.
- The command *math $|(Cr*rough) [A]* will result (5, 6, 7), as the magnitude values are asked in ångströms.
- The assignment *math $.(Cr*rough) =: $|(Cr*rough)* will set the resolution parameters in order to (5 Å, 0.6 nm, 0.7 Å), as no unit was specified on either side, the result of the expression on the right side is expected in the internal unit of the parameters on the left side, which in this case is Å, i.e. (5 Å, 6 Å, 0.7 Å), and as the second parameter unit is in nm, therefore, the second component is converted to that.
- The assignment *math $.(Cr*rough)[nm] =: $|(Cr*rough)* will set the resolution parameters in order to (50 Å, 6 nm, 7 Å), as the result of the right side is expected in nm and there was no unit specified there, therefore the program assumes that it is (5 nm, 6 nm, 0.7 nm), and this are converted to the current units of the resolution parameters.
- The assignment *math $.(Cr*rough)[A] =: $|(Cr*rough)* will set the resolution parameters in order to (5 Å, 0.6 nm, 0.7 Å), as the result of the right side is expected in Å and there was no unit specified there.
- The assignment *math $.(Cr*rough)[A] =: $|(Cr*rough)*[A] will set the resolution parameters in order to (5 Å, 0.6 nm, 7 Å), as the result of the right side is expected in Å, and is calculated in Å.
- The assignment *math $.(Cr*rough)[A] =: $|(Cr*rough)*[nm] will set the resolution parameters in order to (0.5 Å, 0.06 nm, 0.7 Å), as the result of the

right side is expected in Å, but is calculated in nm.

• The assignment *math $.(Cr*rough)[nm] =: $|(Cr*rough)*[nm] will set the resolution parameters in order to (5 Å, 0.6 nm, 7 Å), as the result of the right side is expected in nm, and is calculated in nm.

• The assignment *math $.(Cr*rough) =: $|(Cr*rough)*[nm] will set the resolution parameters in order to (0.5 Å, 0.06 nm, 0.7 Å), as no unit was specified on the left side, therefore its expect the results in the internal units which in this case is Å, and the right side is calculated in nm.

• The assignment *math $.(Cr*rough)[*nm] =: $|(Cr*rough)*[nm] (which is equivalent to *math $.(Cr*rough)[nm][nm] =: $|(Cr*rough)*[nm]) will set the resolution parameters in order to (0.5 nm, 0.6 nm, 0.7 nm), as here the unit to which the resolution parameter is to be set was chosen to nm too.

• The assignment *math $.(Cr*rough)[A][nm] =: $|(Cr*rough)*[nm] will set the resolution parameters in order to (5 Å, 6 Å, 7 Å), as here the unit to which the resolution parameter is to be set was chosen to Å.

On the right side of a parameter assignment *$$* corresponds to the expression on the left side. E.g. *math $(Cr*roug)[nm] =: sin(2.0*$$)* is equivalent with *math $(Cr*roug)[nm] =: sin(2.0*$(Cr*roug)[nm])*. As we may see from this example the command $$ includes the unit specified on the left side as the unit in which the result of $\mathcal{M}$ is expected, at least if it was given there. But we may also specify the units independently like *$$[A],* e.g. *math $(Cr*roug)[nm] =: sin(2.0*$$[A])* is equivalent to the command *math $(Cr*roug)[nm] =: sin(2.0 * $(Cr*roug) [A])*. We have similar shortcuts for the minimum ($$<), maximum ($$>), magnitude ($$|) and resolution ($$.) values too.

3.7.1 Mathematical functions in math command

• **$**($\mathcal{F}$) parameter filter function. A parameter filter $\mathcal{F}$ can be given in a mathematical expression in the form *$($\mathcal{F}$)[u]*, where the *[u]* is optional and it is the unit in which the values of the filtered parameters are requested, or on the left side of a parameter assignment the unit in which the result of the expression on the right side is expected.

In case of parameter assignment, the form *$($\mathcal{F}$)[w][u]* is also meaningful, then *[w]* (which is optional too) gives the unit to which the parameter should be set. If the unit is not given, the program will use the parameters in their current units, i.e. the units which appear in the Parameter Editor. The expression *$($\mathcal{F}$)[u][u]* has an equivalent shorter form *$($\mathcal{F}$)[*u]*.

• **$<**($\mathcal{F}$) parameter filter function, it filters the minimum values. For further details, see the help of **$**($\mathcal{F}$) parameter filter function.

• **$>**($\mathcal{F}$) parameter filter function, it filters the maximum values. For further details, see the help of **$**($\mathcal{F}$) parameter filter function.

• **$l**($\mathcal{F}$) parameter filter function, it filters the magnitude values. For further details, see the help of **$**($\mathcal{F}$) parameter filter function.

• **$.**($\mathcal{F}$) parameter filter function, it filters the resolution values. For further details, see the help of **$**($\mathcal{F}$) parameter filter function.

• **sum**(v) calculates the sum of the components of an array v. E.g.: *math sum($(thick & !Substrate))* will return the sum of the layer thicknesses except the one of which the parameter name contains the string **Substrate**.

• **seqparsum**(v) or **sps**(v) calculates the **seq**uence of **par**tial **sum**s of the components of an array v, i.e. form the array $(v_1, v_2, \ldots, v_n)$ we get sps(v) = $(v_1, v_1 + v_2, v_1 + v_2 + v_3, \ldots, \sum_{i=1}^{n-1} v_i, \sum_{i=1}^{n} v_i)$. E.g.: *math sps($(thick & !Substrate))* will return the depth of the layer interfaces, the last component of this array will be the depth, of the substrate (**Substrate**), i.e. the thickness of the multilayer system without the substrate. Of course, this is meaningful only if there are no repetition groups.

• **reverse**(v) or **rev**(v) reverses the order of components of an array v.

• **dim**(v) returns the dimension of v.

• **component**(v, i) or **com**(v, i) returns the i-th component of v if i is greater than -1 and less than the dimension of v, otherwise it returns 0.

• **array**($a, b, c, \ldots, n$) or **arr**($a, b, c, \ldots, n$), where the arguments $a, b, c, \ldots, n$ should be scalars, returns an array $(a, b, c, \ldots, n)$. E.g.: *arr(1, 2, 3)* will be a 3 dimensional array, *arr(1, $(wavelength), 3, sin($pi/2))* a 4 dimensional array (**If $(wavelength) filters several parameter values, we will have a problem, as array(...) expects a scalar and not an array**).

• **abs**(x) returns the absolute value of x. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(|x_1|, |x_2|, \ldots, |x_n|)$.

• **sgn**(x) calculates the signum function $\operatorname{sgn} x$. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(\operatorname{sgn} x_1, \operatorname{sgn} x_2, \ldots, \operatorname{sgn} x_n)$.

• **floor**(x) calculates the floor function $\lfloor x \rfloor$. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $\lfloor x_1 \rfloor, \lfloor x_2 \rfloor, \ldots, \lfloor x_n \rfloor)$. E.g.: floor(2.3) = $\lfloor 2.3 \rfloor = 2, \lfloor 2.8 \rfloor = 2, \lfloor 3.5 \rfloor = 3, \lfloor 3 \rfloor = 3$.

• **ceil**(x) calculates the ceiling function $\lceil x \rceil$. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $\lceil x_1 \rceil, \lceil x_2 \rceil, \ldots, \lceil x_n \rceil)$. E.g.: ceil(2.3) = $\lceil 2.3 \rceil = 3, \lceil 2.8 \rceil = 3, \lceil 3.5 \rceil = 4, \lceil 3 \rceil = 3$.

• **round**(x) rounds x to the nearest integer. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(\mathrm{round}(x_1), \mathrm{round}(x_2), \ldots, \mathrm{round}(x_n))$. E.g.: $\mathrm{round}(2.3) = 2, \mathrm{round}(2.8) = 3, \mathrm{round}(3.5) = 4, \mathrm{round}(3) = 3$.

• **lt**(x, y) returns 1 if $x < y$ and 0 otherwise. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(\mathrm{lt}(x_1, y), \mathrm{lt}(x_2, y), \ldots, \mathrm{lt}(x_n, y))$. Similarly if x is a scalar and y an array, then it will return $(\mathrm{lt}(x, y_1), \mathrm{lt}(x, y_2), \ldots, \mathrm{lt}(x, y_n))$. If both are arrays of the same dimension $(\mathrm{lt}(x, y) = (\mathrm{lt}(x_1, y_1), \mathrm{lt}(x_2, y_2), \ldots, \mathrm{lt}(x_n, y_n))$. If they are different dimensions (and not scalars), then the result is undefined.

• **le**(x, y) returns 1 if $x \leqslant y$ and 0 otherwise. If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *le*.

• **gt**(x, y) returns 1 if $x > y$ and 0 otherwise. If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *gt*.

• **ge**(x, y) returns 1 if $x \geqslant y$ and 0 otherwise. If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *ge*.

• **eq**(x, y) returns 1 if $x = y$ and 0 otherwise. If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *eq*.

• **neq**(x, y) returns 1 if $x \neq y$ and 0 otherwise. If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *neq*.

• **not**(x) returns 1 (0) if x is (not) equal to 0. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(\mathrm{not}(x_1), \mathrm{not}(x_2), \ldots, \mathrm{not}(x_n))$.

• **and**(x, y) returns x and y, where x and y are boolean ($x(y)$ are false if x is equal to 0 (y is equal to 0) and true otherwise). If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *and*.

• **or**(x, y) returns x or y, where x and y are boolean ($x(y)$ are false if x is equal to 0 (y is equal to 0) and true otherwise). If x and/or y are arrays, then the results are determined as for the function *lt*, but here we use *or*.

• **if**$(x, y, z, \ldots)$ is a function used to define conditional expressions. It should have $3 + 2 * N$ arguments, where $N = 0, 1, 2, \ldots$ E.g.:

– $\mathrm{if}(x, y, z)$ is y if the expression x is not 0, and z otherwise;

– $\mathrm{if}(x, y, z, w, u)$ is y if the expression x is not 0, and otherwise w if z is not 0, if z is 0 than u;

– $\mathrm{if}(a_1, a_2, \ldots, a_{2k+1})$ corresponds to the following C(++) code snippet:
if(a_1) $r = a_2$;
else if(a_3) $r = a_4$;
$\vdots$
else if(a_{2k-1}) $r = a_{2k}$;

else $r = a_{2k+1}$;,
where r represents the result of the function.

For example, the *sinus cardinalis* $\mathrm{sinc}(x) = \frac{\sin x}{x}$ function can be defined with the command *math sinc(x) := if(lt(abs($x),1e-100), 1, sin($x)/$x).*

• **pow**(x, y) calculates the power function x^y. If x is an array $(x_1, x_2, \ldots, x_n)$, then it will return $(x_1^y, x_2^y, \ldots, x_n^y)$. Similarly if x is a scalar and y an array, then it will return $(x^{y_1}, x^{y_2}, \ldots, x^{y_n})$. And if both are arrays of the same dimension the result will be $(x_1^{y_1}, x_2^{y_2}, \ldots, x_n^{y_n})$. If they are different dimensions (and not scalars), then the result is undefined.

• **exp**(x) calculates the natural exponential function of x. If x is an array of size n, then it will return $(\exp(x_1), \exp(x_2), \ldots, \exp(x_n))$.

• **ln**(x) or **log**(x) calculates the natural logarithm of x. If x is an array of size n, then it will return $(\ln(x_1), \ln(x_2), \ldots, \ln(x_n))$.

(**log10**(x) is the logarithm of x to base 10.)

• **log10**(x) calculates the logarithm of x to base 10. If x is an array of size n, then it will return $(\log_{10}(x_1), \log_{10}(x_2), \ldots, \log_{10}(x_n))$.

(**log**(x) and **ln**(x) are the used notations for natural logarithm of x.)

• **sin**(x) calculates the sine of x in radian. If x is an array of size n, then it will calculate $(\sin x_1, \sin x_2, \ldots, \sin x_n)$.

• **cos**(x) calculates the cosine of x in radian. If x is an array of size n, then it will calculate $(\cos x_1, \cos x_2, \ldots, \cos x_n)$.

• **tan**(x) or **tg**(x) calculates the tangent of x in radian. If x is an array of size n, then it will calculate $(\tan x_1, \tan x_2, \ldots, \tan x_n)$.

• **cot**(x) or **ctg**(x) calculates the cotangent of x in radian. If x is an array of size n, then it will calculate $(\cot x_1, \cot x_2, \ldots, \cot x_n)$.

• **asin**(x) or **arcsin**(x) calculates the arcus sine of x the result is in radian. If x is an array of size n, then it will calculate $(\mathrm{asin}\, x_1, \mathrm{asin}\, x_2, \ldots, \mathrm{asin}\, x_n)$.

• **acos**(x) or **arccos**(x) calculates the arcus cosine of x the result is in radian. If x is an array of size n, then it will calculate $(\mathrm{acos}\, x_1, \mathrm{acos}\, x_2, \ldots, \mathrm{acos}\, x_n)$.

• **atan**(x) or **arctg**(x) calculates the arcus tangent of x the result is in radian. If x is an array of size n, then it will calculate $(\mathrm{atan}\, x_1, \mathrm{atan}\, x_2, \ldots, \mathrm{atan}\, x_n)$.

• **sinh**(x) or **sh**(x) calculates the hyperbolic sine of x. If x is an array of size n, then it will calculate $(\sinh x_1, \sinh x_2, \ldots, \sinh x_n)$.

• **cosh**(x) or **ch**(x) calculates the hyperbolic cosine of x. If x is an array of size n, then it will calculate $(\cosh x_1, \cosh x_2, \ldots, \cosh x_n)$.

• **tanh**(x) or **th**(x) calculates the hyperbolic tangent of x. If x is an array of size n, then it will calculate $(\tanh x_1, \tanh x_2, \ldots, \tanh x_n)$.

• **coth**(x) or **cth**(x) calculates the hyperbolic cotangent of x. If x is an array of size n, then it will calculate $(\coth x_1, \coth x_2, \ldots, \coth x_n)$.

• **asinh**(x) or **arsh**(x) calculates the area hyperbolic sine of x. If x is an array of size n, then it will calculate $(\operatorname{asinh} x_1, \operatorname{asinh} x_2, \ldots, \operatorname{asinh} x_n)$.

• **acosh**(x) or **arch**(x) calculates the area hyperbolic cosine of x. If x is an array of size n, then it will calculate $(\operatorname{acosh} x_1, \operatorname{acosh} x_2, \ldots, \operatorname{acosh} x_n)$.

• **atanh**(x) or **arth**(x) calculates the area hyperbolic tangent of x. If x is an array of size n, then it will calculate $(\operatorname{atanh} x_1, \operatorname{atanh} x_2, \ldots, \operatorname{atanh} x_n)$.

• **erf**(x) calculates the error function of x. If x is an array of size n, then it will calculate $(\operatorname{erf}(x_1), \operatorname{erf}(x_2), \ldots, \operatorname{erf}(x_n))$.

• **invErf**(x) calculates the inverse of the error function. If x is an array of size n, then it will calculate $(\operatorname{invErf}(x_1), \operatorname{invErf}(x_2), \ldots, \operatorname{invErf}(x_n))$.

• **Ei**(x) calculates the exponential integral $\operatorname{Ei}(x) = -\operatorname{E}_1(-x) = \int_{-x}^{\infty} \frac{\mathrm{e}^{-t}}{t}\,\mathrm{d}t$ of x. If x is an array of size n, then it will calculate $(\operatorname{Ei}(x_1), \operatorname{Ei}(x_2), \ldots, \operatorname{Ei}(x_n))$.

• **expInt**(m, x) calculates the exponential integral $\operatorname{E}_m(x) = \int_{1}^{\infty} \frac{\mathrm{e}^{-xt}}{t^m}\,\mathrm{d}t$ of x. If x is an array of size n, then it will calculate $(\operatorname{E}_m(x_1), \operatorname{E}_m(x_2), \ldots, \operatorname{E}_m(x_n))$. There exist extensions for real m and negative x values, but the implementation used by the program assumes $m = 0, 1, 2, \ldots$ and $x \geqslant 0$. If m is not an integer, it will calculate with floor(m).

• **gamma**(x) calculates the gamma function $\Gamma(x)$. It returns nan for negative integer values, and 0. It returns 0 for values greater than 171 (because of the C, C++ implementation...).
If x is an array of size n, then $\Gamma(x) = (\Gamma(x_1), \Gamma(x_2), \ldots, \Gamma(x_n))$.

• **lgamma**(x) calculates the natural logarithm of the gamma function $\ln \Gamma(x)$. It is defined only for positive $(x > 0)$ values. If x is an array of size n, then $\ln \Gamma(x) = (\ln \Gamma(x_1), \ln \Gamma(x_2), \ldots, \ln \Gamma(x_n))$.

• **beta**(x, y) calculates the modified beta function $B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$. If x is an array of size n, then $B(x, y) = (B(x_1, y), B(x_2, y), \ldots, B(x_n, y))$.

• **cyl_bessel_i**(ν, x) calculates the modified Bessel function of the first kind $\operatorname{I}_\nu(x)$. If x is an array of size n, then $\operatorname{I}_\nu(x) = (\operatorname{I}_\nu(x_1), \operatorname{I}_\nu(x_2), \ldots, \operatorname{I}_\nu(x_n))$.

• **cyl_bessel_k**(ν, x) calculates the modified Bessel function of the second kind $\operatorname{K}_\nu(x)$. If x is an array of size n, then $\operatorname{K}_\nu(x) = (\operatorname{K}_\nu(x_1), \operatorname{K}_\nu(x_2), \ldots, \operatorname{K}_\nu(x_n))$.

- **cyl_bessel_j**(ν, x) calculates the Bessel function of the first kind $\mathrm{J}_\nu(x)$. If x is an array of size n, then $\mathrm{J}_\nu(x) = (\mathrm{J}_\nu(x_1), \mathrm{J}_\nu(x_2), \ldots, \mathrm{J}_\nu(x_n))$.
- **cyl_neumann**(ν, x) calculates the Bessel function of the second kind $\mathrm{N}_\nu(x)$. If x is an array of size n, then $\mathrm{N}_\nu(x) = (\mathrm{N}_\nu(x_1), \mathrm{N}_\nu(x_2), \ldots, \mathrm{N}_\nu(x_n))$.

3.8 Report generator

On clicking Results » Create Report appears a window with a report of the current project containing the model structure and the model parameter values. The report can be saved in an html file. (This feature is still in a very early development stage.)

3.9 Cloning

It happens frequently that the user wants to fit the same type of experiment in a bit different environment, or a bit different sample, etc. It would be inconvenient to build up almost the same model several times and then to correlate almost all the parameters. Therefore in FitSuite we can clone the models. This can be done by just right clicking on the model name in the window *Problems* and selecting 🖱 » Clone from the pop-up menu. Thereafter a dialog arises in which we can choose the number of clones and the parameters and/or matrices which are (not to) to be correlated. After this we will have copies of the chosen model and of the data belonging to it. These data may be replaced by right clicking on them in the window *Problems* and choosing 🖱 » Replace Data from arising pop-up menu.

In the clone dialog we may specify the parameters, which will be the same in the clone models and the cloned model, the number of clones. We may additionally add 'post-cloning' commands. In these commands, the labels of the dependent variable columns specified in the data file format dialog (see page 33), appearing if we checked **Replace data with measurement series** and clicked OK in the clone dialog, can be used as parameters in the comand list. The string of the label is referred to as **%s**, if the label is a number, then we may refer to its value as **%v**. Similarly, the name of the new clone model is represented by **%c**, and the name of the new data set as **%d**. Let see an example for such a post-cloning command list (The symbol ↪ notes the line breaks forced by LaTeX to fit the lines in the page. This symbol and the line numbers on the left are not present in the command list.):

```
1 //Comments are started with // as in C++ and end at the end of the line.
2 renMod %c # NoEff%s; //rename the clone model(s) from %c to NoEff%s In the following
  ↪ lines %c will be already NoEff%s
```

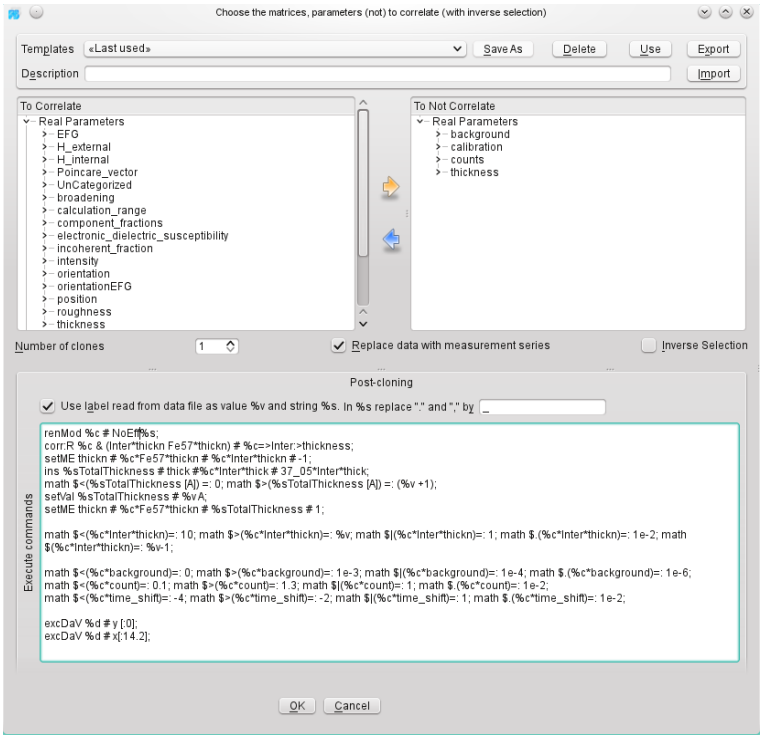

Figure 10: The dialog used to clone a model.

```
corr:R %c & (Inter*thickn Fe57*thickn) # %c=>Inter:>thickness; //correlate the parameters %c & (Inter*thickn Fe57*thickn), the new correlated parameter name will be %c=>Inter:>thickness
setME thickn # %c*Fe57*thickn # %c*Inter*thickn # -1; //set the element row %c* Fe57*thickn , column %c*Inter*thickn of the transformation matrix thickn (i.e. thickness) to -1.
ins %sTotalThickness # thick #%c*Inter*thick # 37_05*Inter*thick; //Insert a new fit/simulation parameter with name %sTotalThickness in the transformation matrix thickn ( thickness) before %c*Inter*thick and initialize it according to the fit/simulation parameter 37_05*Inter*thick
math $<(%sTotalThickness [A]) =: 0; //Set the minimum value to 0 ångström
math $>(%sTotalThickness [A]) =: (%v +1); //Set the maximum value to (%v+1) ångström
setVal %sTotalThickness # %v A; //Set the value to %v ångström
setME thickn # %c*Fe57*thickn # %sTotalThickness # 1; //set the element row %c* Fe57*thickn , column %sTotalThickness of the transformation matrix thickn (i.e. thickness) to 1.

```

```
11 //set the values of some other uncorrelated parameters.
12 math $<(%c*Inter*thickn)=: 10; math $>(%c*Inter*thickn)=: %v; //several
   ↪ commands may appear in the same line, if they are closed by semicolon.
13 math $|(%c*Inter*thickn)=: 1; math $.(%c*Inter*thickn)=: 1e-2;
14 math $(%c*Inter*thickn)=: %v-1;
15
16 math $<(%c*background)=: 0; math $>(%c*background)=: 1e-3;
17 math $|(%c*background)=: 1e-4; math $.(%c*background)=: 1e-6;
18 math $<(%c*count)=: 0.1; math $>(%c*count)=: 1.3;
19 math $|(%c*count)=: 1; math $.(%c*count)=: 1e-2;
20 math $<(%c*time_shift)=: -4; math $>(%c*time_shift)=: -2;
21 math $|(%c*time_shift)=: 1; math $.(%c*time_shift)=: 1e-2;
22 math $<(%c*L_width)=: 0; math $>(%c*L_width)=: 10; math $.(%c*L_width)
   ↪ =: 0.1;
23
24 excDaV %d # y [:0]; //Exclude the data points with negative dependent variable
25 excDaV %d # x[:14.2]; //Exclude the data points with independent variable value less, than
   ↪ 14.2.
26
27 fix; //Fix all the parameters
28 free %c*background %c*counts %c*time_shift %c*Inter*thickness; //Free some of
   ↪ the parameters belonging to the new clone model.
```

3.10 Merging projects

If we work with a lot of data sets simultaneously, it occurs frequently that we start the fitting one data set, then another and save these in different project files, as keeping all the data in a single project file in the early stage of the fitting process, when it is still not clear which parameters have significant effect, are worth to be fitted, are to be correlated, etc. But later we may want to see the fits together, to fit the parameters simultaneously, to be able to observe the trends. In such a case, we can merge projects clicking (File ⟩ Merge with Project), but only if the merged projects do not have models and simulation/fit parameters with the same name, and do not contain models of different model type repositories.

3.11 Model groups

The user may group models from version 1.0.3. The model groups are used just to select a few models from the available ones in the current project, in order to simulate, fit, plot only them. To create model groups just select them in the

window **Problems** with shift + cursor, right click with mouse and in the arising pop-up menu select Group Model(s). In the dialog showing up thereafter the user may choose the models to be grouped and the name of the group according to which we can use them later on see modelgroup demo.

3.12 Simulation, Fit

The simulation can be started by clicking Fit/Simulation » Simulate. The program checks, whether there was a former simulation, and the parameter values were changed or not, and calculates only when it is necessary. It may happen that this program decision was not appropriate. In such cases you may force simulation by clicking Fit/Simulation » Force Simulation. It is possible to simulate only a single model (fitting problem), or models of a model group (see subsubsection 3.11) choosing the proper menu items of Fit/Simulation » Simulate Only ... and Fit/Simulation » Force Simulation of ....

The independent variable of the simulation/fit may be specified by setting properly in the *Problems* window on the left side in the column with name **grid type**, if there is a possibility. Just click on the proper cell, and change it, if it is possible. (Some models have only one grid type.)

The iteration (fitting) can be started by clicking Fit/Simulation » Fit. It is possible to fit only a single model (fitting problem), or models of a model group choosing the proper menu items of Fit/Simulation » Fit Only ....

Choosing the menu item Fit/Simulation » Select Fitting Method you can select the fitting method and set their parameters. At present we have the following methods:

3.12.1 Powell's method

Powell's method which is a slightly modified version of the code available in *Numerical Recipes in C* available at http://www.nrbook.com section 10.5. The method has the following parameters, options:

- *MaxIter*: is the maximum number of allowed iteration steps. If a minimum was not found in so many steps, the optimization is finished. (The Nelder – Mead, Polak – Ribiere, Fletcher – Reeves, Broyden – Fletcher – Goldfarb – Shanno, and Levenberg – Marquardt methods use this as well.)

- *tolerance*: is the tolerance τ_{tol} with which the minima of the function f is determined. The result f_n of the n^{th} iteration step is accepted if: $\tau_{\text{tol}}\left(|f_{n-1}| + |f_n|\right) \geqslant |f_{n-1} - f_n|$.

(The Nelder – Mead, Broyden – Fletcher – Goldfarb – Shanno, Polak – Ribiere, Fletcher – Reeves, and Levenberg – Marquardt methods use this as well. In the last two it has a different meaning, role.)

Powell's method uses line minimization methods, as *Golden-section search* (see in *Numerical Recipes in C* available at http://www.nrbook.com section 10.1 and/or on Wikipedia[8]), *Brent's method*, *Brent's method using derivatives*(see in *Numerical Recipes in C* section 10.2 and 10.3, respectively). The user may be choose one of them. They have the following parameters, options:

- *MaxIterLine*: is the maximum number of iteration steps in the 1 dimensional optimization method, if a minimum was not found in so many steps, the current line optimization is finished. (The main method may continue its own iteration. The fitting is not finished just because *MaxIterLine* was reached.)

- *tolLine*: is the fractional tolerance t_{Line} with which the minimum along the given direction should be found by *Brent's method* or *Brent's method using derivatives.* Optimization along a direction is finished if $2t_{\text{Line}}|x| \geqslant |x_0 - x|$, where x_0 is the true local minimum and x is the calculated one.

- *GLimit*: In the optimization methods Powell, Fletcher – Reeves and Polak – Ribiere the minimum of the function f (which in case of fitting of experimental data sets is the χ^2) is searched along directions specified by them. The first step to find a minimum along a direction is to find three points a, b and c where b is between a and c furthermore $f(a)$ and $f(c)$ are both greater than $f(b)$. The three points are searched by starting from an initial triplet moving similarly to an inchworm, i.e. updating (a, b, c) by $(a = b, b = c, c = u)$, where u is the minima of the parabolic fit on $(a, f(a); \quad (b, f(b)); \quad (c, f(c))$. This type of move is accepted only if the obtained u is between c and $u_{\text{lim}} = b + G_{\text{Limit}}(c - b)$, otherwise the $(a = b, b = c, c = u_{\text{lim}})$ move is made. This is quite oversimplified just to explain the use of **GLimit**. For further details, see the routine *mnbrak* in 'Numerical Recipes in C (Fortran)'.

There is one additional option with which may choose the order of line minimization directions randomly in each iteration step of the Powell's, method. This may be useful sometimes.

[8]https://en.wikipedia.org/wiki/Golden section search

3.12.2 Nelder – Mead method

Nelder – Mead method (*Numerical Recipes in C* based, section 10.4). A good description, with animation (and with a bit different terminology) is available on wikipedia[9]. Nelder – Mead method is sensitive to scaling of the parameters. It has the following parameters and options:

- *MaxIter*: as in Powell's method.
- *tolerance*: as in Powell's method.
- *Reflection factor*: see *Numerical Recipes in C*. Its default value is 1.
- *Stretch factor*: see *Numerical Recipes in C*. (On wikipedia it is called the expansion coefficient) Its default value is 2.
- *Contraction factor*: see *Numerical Recipes in C*. (On wikipedia you find a contraction and shrink coefficient as well, in FitSuite, we use the same value for both transformation, therefore we have only this contraction factor.) Its default value is $\frac{1}{2}$.
- There are parameters which determine, how the initial simplex is generated:
  - It is possible to have a simplex which contains the initial parameter vector (consisted only of the free components): as one of its vertices, or as its center. In the former case we have the possibility to chose that vertex randomly, or by giving an ordinal number. These make possible to try to restart fitting, by a different simplex in order try to get out of local minimums, without changing the free parameters. Sometimes it is worth to start a fit again, even if it reached convergence, as the new initial simples generally will be quite different from the one with which the convergence was reached, therefore this way further improvement may be possible. We should stop this only if no change was experienced several times, as then we probably are in the 'attraction basin' of at least a local minimum. If we chose the initial parameter vector to be the centre of the initial simplex, than it may happen, that the method stops with an objective function value greater than the initial one. As that is always the least function value belonging to the vertices of the final simplex, while the starting value belongs to the centre of the initial simplex. This may be a bit disturbing sometimes, but on the other hand it may help to extricate ourselves from some nasty local minima.

[9] https://en.wikipedia.org/wiki/Nelder-Mead method

– *Initial simplex size*: is a parameter determining the length of the vectors pointing to the vertices of the simplex from the center of the simplex. The initial simplex is always a regular one, therefore in the case of this method is also worth to rescale the free parameters to the same order of magnitude (see subsubsection 2.6.3). Fitting again with different initial simplex sizes may also help to explore the nature of the minimum we reached, the extent of its 'basin of attraction'.

3.12.3 Polak – Ribiere and Fletcher – Reeves

Polak – Ribiere and Fletcher – Reeves methods (*Numerical Recipes in C* based, section 10.6 and on wikipedia[10]). They have the same (a bit less) options and parameters as Powell's method, but they use derivatives.

3.12.4 Broyden – Fletcher – Goldfarb – Shanno

Broyden – Fletcher – Goldfarb – Shanno (on wikipedia[11]), which is a variant of the Davidson – Fletcher – Powell method (*Numerical Recipes in C* based, section 10.7, may see also on wikipedia[12]). The method uses derivatives and has the following parameters, options:

- *MaxIter*: as in Powell's method.
- *tolerance*: It is the same, as in Powell's method, but here it is used to check whether the gradient is already small (near to 0) enough, i.e. the following condition $\tau_{\text{tol}} \max(|f_n|, 1) \geqslant \max_l \left| \frac{\partial f}{\partial p_l} \cdot \max(|p_l|, 1) \right|$ is satisfied, where τ_{tol} is the tolerance, f_n is the result of the n^{th} iteration step and $\mathbf{p}$ is the parameter vector.
- *MaxStep*: is the scaled maximum step length allowed in line searches in BFGS. The length of a step along the direction of $\boldsymbol{\nabla} f(\mathbf{p})$ will be no greater, than $S_{\max} \cdot \max(|\mathbf{p}_{\text{free}}|, \dim \mathbf{p}_{\text{free}})$, where $S_{\max}$ denotes *MaxStep*, $\mathbf{p}_{\text{free}}$ the vector of free parameters. Its default value is 100. For further details, see *stpmx* variable in the corresponding Numerical Recipes in C function (Fortran subroutine).
- *Alpha*: ensures sufficient decrease in function value in line searches in BFGS, i.e. the function value $f(\mathbf{p}^{\text{new}})$ at the new point fulfills the condition $f(\mathbf{p}^{\text{new}}) \leqslant f(\mathbf{p}^{\text{previous}}) + \alpha(\mathbf{p}^{\text{new}} - \mathbf{p}^{\text{previous}}) \cdot \boldsymbol{\nabla} f(\mathbf{p}^{\text{previous}})$, where α is

[10] https://en.wikipedia.org/wiki/Nonlinear conjugate gradient method
[11] https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno algorithm
[12] https://en.wikipedia.org/wiki/Davidon-Fletcher-Powell formula

Alpha. Its default value is 10^{-4}. For further details, see *ALF* variable in the corresponding Numerical Recipes in C function (Fortran subroutine).

- *TolxLnsrch*: gives convergence criterion on the location of the minimum in line searches in BFGS. Convergence is reached in line search, if the step length $\left|\mathbf{p}^{\text{new}} - \mathbf{p}^{\text{previous}}\right|$ satisfies the condition

$$\left|\mathbf{p}^{\text{new}} - \mathbf{p}^{\text{previous}}\right| < \tau_{\text{Lnsrch}} \min_i \frac{\left|\boldsymbol{\nabla} f(\mathbf{p}^{\text{previous}})\right| \max\left(\left|p_i^{\text{previous}}\right|, 1\right)}{\left|\partial_{p_i} f(\mathbf{p}^{\text{previous}})\right|},$$

  where τ_{Lnsrch} denotes *TolxLnsrch*.

  Its default value is 10^{-7}. For further details, see *TOLX* variable in the corresponding Numerical Recipes in C (lnsrch) function (Fortran subroutine).

3.12.5 Levenberg – Marquardt

Levenberg – Marquardt method (*Numerical Recipes in C* based, section 15.5 may see also on wikipedia[13]).

- *MaxIter*: as in Powell's method.
- *tolerance*: It is the same, as in Powell's method, but here it is used to check whether the relative change in the objective function is great enough. The iteration is stopped, when $\tau_{\text{tol}} \max(1, f_n) \geqslant |f_n - f_{n-1}|$, where τ_{tol} is the tolerance, f_n is the result of the n^{th} iteration step. **Because of** $\max(1, f_n)$ **it is clear, that for objective functions much less, than 1 it will not work properly.** This was written thus, as the χ^2 is rarely less, than 1, if we have known properly the uncertainties ...
- λ_0: It is the initial value of the damping factor λ in Levenberg-Marquardt method, which in case $\lambda = 0$ corresponds to *inverse Hessian method,* and in case of $\lambda \to \infty$ to *steepest descent method.*
- q: It gives the factor by which the damping factor λ is multiplied or divided in Levenberg-Marquardt method, when we increase or decrease its value.
- λ_{min} and λ_{max}: give the allowed minima and maxima of the damping factor λ of the Levenberg-Marquardt method.

[13] https://en.wikipedia.org/wiki/Levenberg-Marquardt algorithm

3.12.6 Levenberg – Marquardt method (LMDER) from the MINPACK

Levenberg – Marquardt method (LMDER) from the MINPACK [16, 17] package available at https://www.netlib.org/minpack. This was translated from Fortran into C++ and modified a bit by us, so it may work a bit differently, than the original one.

- *MaxFev*: It is the allowed maximum number of function evaluation.
- *ftol*: It is a nonnegative parameter. Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most *ftol*. Therefore, *ftol* measures the relative error desired in the sum of squares.
- *gtol*: It is a nonnegative parameter. Termination occurs when the cosine of the angle between the 'fitted vector statistic' $\boldsymbol{\kappa}$ and any column of the jacobian $(\boldsymbol{\nabla}_{\mathbf{p}}\kappa_i)$ is at most *gtol* in absolute value. Therefore, *gtol* measures the orthogonality desired between the function vector and the columns of the jacobian.
- *xtol*: It is a nonnegative parameter. Termination occurs when the relative error between two consecutive iterates is at most *xtol*. Therefore, *xtol* measures the relative error desired in the approximate solution.
- *factor*: It is a positive parameter used in determining the initial step bound. This bound is set to the product of *factor* and the euclidean norm of $\mathbf{Dp}$, where $\mathbf{D}$ is a diagonal matrix containing internally determined multiplicative scale factors for the free fitting parameters $\mathbf{p}$, if nonzero, or else to *factor* itself. In most cases factor should lie in the interval (0.1, 100). 100 is a generally recommended value.

3.12.7 Genetic algorithms

Genetic algorithms are used nowadays for a lot of things, in a large area of science, engineering, mathematics, etc. According to wikipedia[14]: *'In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. [...] The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the*

[14]https://en.wikipedia.org/wiki/Genetic algorithm

current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.'

In FitSuite we use an example algorithm available in the appendix of [18] called *'Continous genetic algorithm'*, which we modified, generalized based on the ideas available in the literature of genetic algorithms, especially in [18–20]. Here we try to enumerate, explain the ideas, options available in our algorithm.

The individuals are points in the subspace of the fitted free parameters, selected initially randomly in the region specified by minimum and maximum values (bounds) of the corresponding parameters. It is important to give a not too large interval for the parameters, as that can affect the convergence badly. Usually it is not hard to provide such information, as we know what values are already unrealistic for the given problem, we should set the bounds accordingly. The genetic algorithms should not be used to get the 'exact' optimum, but to get close enough to it and thereafter we can increase the precision using other methods. This method does not use derivatives.

- *Maximum number of generations:* It is obvious, that the algorithm should stop even if the convergence is not reached, after a reasonable number of generation was tried, if we do not want to end up occasionally in an endless cycle. This option gives that number.

- *Population size:* It is the number of individuals (points in the parameter space), which are evaluated in each generation. If we set it to a too large value, than we calculate in a lot of points, and this slows the convergence. If we set it to a too small value, than we calculate in a few points and we cannot check a large enough region of the parameter space, and this can slow the convergence as well.

- *Size of elite list:* After the fitness is evaluated for all the individuals of a generation, we select the new generation replacing the current individuals. It is clear, that it is not good if we do not keep some of the bests for the next generation, as we may throw out a very good individual. Size of elite list gives the number of the best individuals we keep from the current generation. This should be at least 1.

- *Convergence generation number:* If the best fitness value did not change too much through the last N generations (i.e. $\left|f^b_{i+N} - f^b_i\right| < \textit{"Tolerance"} \cdot \left|f^b_{i+N}\right|$, where f^b_i is the best value in the i-th generation), then it is reasonable to stop, as at least a local minimum is reached. This option specifies this N.

- *Tolerance:* It gives the tolerance according to which we decide whether convergence was reached. For further detalis see *'Convergence generation number'* above.
- *Mutation rate:* It is a number between 0 and 1. It specifies the number of parameters of the individuals which are replaced, changed in each new generation randomly, after selection, i.e. *"Number of mutated individual parameters" = "Mutation rate" ·("Population size" - "Size of elite list")·"Number of free parameters"*.
- *Selection ratio:* It is a number between 0 and 1. The number of the population members surviving from previous generations is *"Selection ratio" · "Population size"*. Therefore *"Population size"* ·(1-*"Selection ratio"*) is replaced by the selection mechanism.

With different **selection mechanisms** we select the members of the population which will survive the current generation, and which will be the parents of the new members in the next generation.

Let note with $p_i \quad (i = 0, \ldots, k)$, the probability, that the i-th individual is selected from $k+1$ individuals. (It is $k+1$, as we start the indexing by 0, and do not want to finish it with $k-1$.) In practice the selection is done in the following way. Let define the quantity

$$o_0 = 0, \quad o_{i+1} = \sum_{j=0}^{i} p_j, \tag{41}$$

and as it follows from the definitions

$$p_i = o_{i+1} - o_i, \quad \text{and} \quad o_{k+1} = 1, \tag{42}$$

therefore a random number $v \in [0, 1]$ will fall in the interval $(o_i, o_{i+1}]$ with probability p_i. Thus we select the i-th individual, if $o_i < v \leqslant o_{i+1}$.

The following selection methods differ in the method by which p_i is defined.

Linear rank based selection Using linear rank based selection mechanism the probability of selection of the i-th individual is

$$p_i = q - ir, \quad (i = 0, \ldots, k), \tag{43}$$

where the 0-th is of the highest rank, and the k-th is of the lowest rank [page 60 of ref. 19] and $\sum_{i=0}^{k} p_i = 1$. This equation can be transformed into the form

$$p_i(\alpha) = \frac{1}{k+1}\left(\alpha + 1 - \frac{2\alpha}{k} i\right) = p_{k-i}(-\alpha), \tag{44}$$

where $-1 \leqslant \alpha \leqslant 1$ is the selection pressure. If $\alpha = 0$, then there is no selection pressure, all the individuals are selected with the same probability $p_i = \frac{1}{k+1}$. If $\alpha = 1$ the selection pressure is maximal, as $p_0 = \frac{2}{k+1}, p_1 = \frac{2(k-1)}{k(k+1)}, \ldots, p_{k-1} = \frac{2}{k(k+1)}, p_k = 0$. If $\alpha = -1$ the contraselection pressure is maximal, as $p_0 = 0, p_1 = \frac{2}{k(k+1)}, \ldots, p_{k-1} = \frac{2(k-1)}{k(k+1)}, p_k = \frac{2}{k+1}$. The option of FitSuite *Selection pressure factor* is this α, with which we can influence the speed of the convergence, how fast we throw out the less fit individuals.

Geometric progression rank based selection Using the geometric progression rank based selection mechanism the probability of selection of the i-th individual is

$$p_i = cq(1-q)^i, \quad (i = 0, \ldots, k), \tag{45}$$

where the 0-th is of the highest rank, and the k-th is of the lowest rank [page 60 of ref. 19] and $\sum_{i=0}^{k} p_i = 1$. Therefore

$$p_i = \frac{q(1-q)^i}{1-(1-q)^{k+1}}, \quad (i = 0, \ldots, k). \tag{46}$$

It can be seen, that in cases:

- $q = 0 \Rrightarrow \lim_{q\to 0} p_i = \frac{1}{k+1} = \text{const.} \Rrightarrow$ there is no selection pressure.
- $q = 1 \Rrightarrow p_0 = 1, p_{i>0} = 0 \Rrightarrow$ The selection pressure is maximal.
- $q = -\infty \Rrightarrow p_{i<k} = 0, p_k = 1 \Rrightarrow$ The contraselection pressure is maximal.

In FitSuite we have q as the option *Geometric progression factor*, with which we can influence the speed of the convergence, how fast we throw out the less fit individuals. In the program we may change it between 0 and 1.

Tournament based selection $m+1$ individuals are selected and from these we select a single one. The 'best' (most fittest) one is selected with probability p and the less fit individuals are selected with less and less probability, as their fitness is decreasing:

$$p_0 = p, \quad p_i = c(1-p)^i, \quad (i = 1, \ldots, m). \tag{47}$$

As

$$1 = \sum_{i=0}^{m} p_i = p + c\left(\frac{1-(1-p)^{m+1}}{1-(1-p)} - 1\right) = p + c\left(\frac{1-p-(1-p)^{m+1}}{p}\right), \tag{48}$$

therefore

$$c = \frac{p}{1-(1-p)^m}, \tag{49}$$

and thus

$$p_0 = p, \quad p_i = p\frac{(1-p)^i}{1-(1-p)^m}, \quad (i = 1, \ldots, m). \tag{50}$$

E.g.: $m = 1$, we have a binary tournament $p_0 = p, \quad p_1 = 1 - p$. In FitSuite $m = \lfloor$ *"Tournament ratio"* $\cdot$ *"Selection ratio"* $\cdot$ *"Population size"* $\rfloor$ except, when the m value would be less than 1, or we selected a binary tournament, then we have a binary tournament. ($\lfloor \ldots \rfloor$ denotes the floor function.) p is the *Tournament selection factor* in FitSuite. Practically $p \geqslant \frac{1}{m+1}$, therefore $p \geqslant \frac{1}{2}$ is generally a good default value.

Modified roulette selections In the original roulette selection, the probability of selection of the i-th individual is

$$p_i = \frac{f_i}{\sum f_i}, \tag{51}$$

where f_i is the merit of the i-th individual. The problem with this is, that it was devised for maximalization problems and no minimalization, which we need. Therefore we use another method described below.

As a version of roulette selection it is a custom to use the following [20]. Let be

$$\varrho_i = \frac{f_i - f_{\min}}{f_{\max} - f_{\min}}, \tag{52}$$

where $f_{\max}$ and $f_{\min}$ are maximal and minimal value of the objective function in the given population (and not minimum and maximum of the function $f(.)$!!!).

Using some of the dinamically scaled (as $f_{\max}$, $f_{\min}$ and thus ϱ varies from population to population) merit functions ($F(0) \leqslant 1, \quad 0 \leqslant F(0 \leqslant x \leqslant 1) \leqslant 1, \quad F(x) \underset{x \leqslant y}{\geqslant} F(y)$):

- In case of exponential roulette selection $F(\varrho) = \exp(-S\varrho)$, where S is referred to as the *Exponential roulette scaling factor* in FitSuite.
- $F(\varrho) = \frac{1}{2}\left[1 - \tanh\left(2\pi(2\varrho - 1)\right)\right]$ in case of hyperbolic tangent roulette selection. Instead of this we built in the program the more general function $F(\varrho) = \frac{1}{2}\left[1 - \tanh\left(a(\varrho - b)\right)\right]$, where a is referred to as the *Hyperbolic tangent roulette scaling factor* and b as the *Hyperbolic tangent roulette position* in FitSuite.

- $F(\varrho) = 1 - \varrho^n$ in case of power roulette selection, where n is referred to as the *Power roulette exponent* in FitSuite.

The selection is done in the following way. We pick randomly an individual with index i from the population, we calculate the corresponding $F(\varrho_i,)$ and we select this individual with probability $F(\varrho_i)$, i.e. if $v < F(\varrho_i)$, where $v \in [0, 1]$ is a generated random number. If we have not selected this individual, then we pick randomly an individual, and we check it the same way, and we repeat this until we do not find a proper one.

As $F(\varrho)$ is decreasing monotonically, therefore the greater ϱ_i is, the smaller is the probability that we select it.

The difference between the different functions $F(\varrho)$ arises, in the measure by which they differentiate the 'good' and 'bad' individuals. For example, the exponential function changes fast around 0, therefore it differentiates strongly the good ones, but weakly the bad ones. In case of the power function $n > 1$ we have a reverse behaviour, it heightens the differences between the bad ones, and suppresses between the good ones. The tangent hyperbolic heightens the differences between the average individuals, and it blurrs the differences between the exceptionals and good ones, and between the bad ones and the very bad ones to a certain degree.

Sharing If we have a lot of local optimum, it is not always the best strategy to have a too fast convergence, as we may stuck into a local minimum, as all the individuals get into its basin of attraction, because of the appearance of an excellent, but still not the best individual of an early population. One method to avoid this is sharing [pages 168-169 of ref. 19] which permits formation of stable subpopulations (like species in biology), in this way we may investigate many valleys (local minimums, biological niches) in parallel. It is clear, that decreasing the fitness of an individual, if it has a lot of very close neighbors (i.e. they share a lot of common genes, and because of this they use the same finite resources) will act against the above mentioned tendency. A sharing function $s(d)$ determines the degradation of fitness of an individual, due to a neighbor in distance d. It is a monotonical function, which is defined in such a way that $s(\infty) = 0 \leqslant s(d) \leqslant 1 = s(0)$. One used definition is

$$s(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_s}\right)^\gamma & \text{if} \quad d < \sigma_s \\ 0 & \text{otherwise} \end{cases}, \tag{53}$$

where σ_s is the *Sharing scaling factor* and γ is the *Sharing power* of FitSuite. We should decrease the fitness $f(x)$ of an individual at a position x by a factor of $m(x) = \sum_{\{y\}} s(d(x, y))$, where $\{y\}$ represents the set of all the individuals in the

population, therefore the modified fitness would be $f'(x) = \frac{f(x)}{m(x)}$ (as in FitSuite we minimize and not maximize we multiply by $m(x)$ the objective function). It is clear that if $\{y\}$ has a single element x, then $m(x) = 1$, and the fitness function is unchanged. If all the individuals have the same genes, then $m(x) = \sum s(0) = \sum 1 =$ *"Size of the population"*. There is one question we have not answered, what should be the distance d of two individuals corresponding to the vectors $\mathbf{r}$ and $\mathbf{q}$ (whose components correspond to free parameters $\mathbf{P}$). We have the following choice in FitSuite

$$d(\mathbf{r}, \mathbf{q}) = \sqrt{\sum_i \left| \frac{r_i - q_i}{P_i^{\max} - P_i^{\min}} \right|^2}, \tag{54}$$

as this metric takes into account each free parameter with the same weight ($P_i^{\min}$ and $P_i^{\max}$ are the bounds of the i-th free parameter).

Schematical steps of genetic algorithm used in FitSuite First initialize the population, and determine the fitness of each individual, rank them according their fitness (may use sharing too). The initializations is done by selecting randomly the free parameter values inside the ranges of the corresponding parameter bounds. One individual is not selected randomly, as it corresponds to the initial parameter values set by the user, thus we cannot get a worse result than the starting point. (Sorrily, this is not always quite true, as sometimes the objective function can be smaller for simulation results, which a human would not accept as a better fit. For such examples see the section about *Robust estimation* in *Numerical Recipes in C* or wikipedia[15])

After the initialization start the cycle of generations comprising the following steps:

- Select randomly according to one of the selection mechanisms presented above which individuals will survive the current generation and have offsprings.
- Select randomly the mating pairs from survivors, each pair will have a pair of offsprings.
- Determine the pair of offsprings for each pair using single point crossover which in short is the following. Let represent one parent by a vector $\mathbf{m}$ whose $(m_1 m_2 \dots m_{n_{\text{free}}})$ components are the corresponding free parameters, similarly the other parent by a vector $\mathbf{f} = (f_1 f_2 \dots f_{n_{\text{free}}})$ and the two

[15] https://en.wikipedia.org/wiki/Theil-Sen estimator

offsprings let be $\mathbf{a} = (a_1 a_2 \dots a_{n_{\text{free}}})$ and $\mathbf{b} = (b_1 b_2 \dots b_{n_{\text{free}}})$. Select randomly one of the free parameters, i.e. its index $x \in [1, n_{\text{free}}]$, this will be the crossover point. After a single crossover the offsprings will be the following

$$\begin{array}{rcccccccc} \mathbf{a} & = & m_1 & m_2 & \dots & m_{x-1} & m_x + r(f_x - m_x) & f_{x+1} & \dots & f_{n_{\text{free}}} \\ \mathbf{b} & = & f_1 & f_2 & \dots & f_{x-1} & f_x - r(f_x - m_x) & m_{x+1} & \dots & m_{n_{\text{free}}} \end{array},$$

where $r \in [0, 1]$ is a random number. It is clear that both offsprings will have the genes, i.e. free parameter of both parents and the segments before and after the crossover point belong to different parents; but inside these segments there is no mixing. At the crossover point we have some random mixing and the new parameter will be in the range bounded by f_x and m_x. If they are the same ($m_x = f_x$), the offspring will have the same gene there i.e. $a_x = b_x = m_x = f_x$, and there is no random change at that point. [18]

- Replace with offsprings the individuals which were not selected to survive. (The population size does not change.)
- Mutate randomly some of the individuals (except of the elite ones which are not mutated). As we have written above at the explanation of *Mutation rate*, we mutate $N_{\text{Mut}} =$ *"Mutation rate"* $\cdot$(*"Population size"* - *"Size of elite list"*)$\cdot$*"Number of free parameters"* free parameters in total. Therefore we choose randomly N_{Mut} times an individual (from the new population except the elite members; we may choose the same individual several times), these will be mutated, and then in each case we choose randomly one free parameter which will be changed randomly inside parameter bounds specified by the user in the Parameter Editor (see page 37).
- Evaluate the new generation, determine the new order of fitness, and the new elite individuals.
- Check convergence, if it is not reached start the cycle again with the new generation.

3.12.8 After fit

If the result of the iteration is not what you like, you can get back the state before the iteration by clicking Fit/Simulation 〉 Revert. Before a fit is started the project is saved in the file **.sfp~*, which is loaded on the command *Revert*.

From version 1.0.4 the results are not written in files automatically. They may be exported using the menu items Results 〉 Export ... . You may choose the data which should be exported, and set the format of the created files using (Settings 〉 Export Settings)

3.13 Uncertainty calculation

Uncertainties **of free parameters** may be calculated clicking on the menu item Fit/Simulation » Calculate Uncertainties. There are currently two approaches used in the program for this purpose. One is based on covariance matrix, the other is named bootstrap method (be aware that bootstrap method is very expensive in computation time). Explanation of the principles of these method can be found in *Numerical Recipes in C* available at http://www.nrbook.com section 15.6. You may choose one of them (or both) by proper settings after clicking menu item Fit/Simulation » Select Fitting Method, on the page with title *Parameter Uncertainties*. Here, you may also change the required confidence level. Still in the same dialog on page with title *Bootstrap method* you can set the parameters used by bootstrap method, namely the number of synthetic data sets and the convergence criterion. According to the literature the number of synthetic data sets should be (at least) about a few thousands, the current default value is 200, as we used most only for testing and because of the slowness of this method. The convergence criterion gives the criterion to stop the fitting of a synthetic data set, if the difference of fitted statistics belonging to the real experimental and the current synthetic data sets is smaller than this. **Bootstrap method needs uncertainties (errors) of the experimental data**, without that it will not work appropriately. If the experimental data has Poisson distribution, it is enough just to set the distribution properly.

3.14 Calculating statistics

to be written

3.15 Plotting

(Originally it was planned to use **gnuplot** for plotting of the results. That way we could have more beautiful and appropriate ('press ready') graphs. The problem is that it is a bit circumstantial to get control over the plot windows created by gnuplot. There is a solution, but only for *X11* systems and that would not be portable. Therefore we use the open source plot library Qwt which is based on Qt. This can be integrated in FitSuite and developed further without problems. It is not as beautiful as gnuplot, but it should be enough during fitting or simulations.)

Presently, the (Qwt based) plot windows are created (if they are not available already), when an iteration is started. The data and the results of simulations before and after each iteration are plotted. The theoretical results are represented by their ordinal numbers and with different colours in the plot legend. Clicking on the corresponding part of the legends the user may hide(show) the corresponding curves. Right clicking on the plot window, the user may zoom in (out) a selected

(first click on Zoom in in the pop-up menu and after that select with mouse) rectangular region of the plot. You may choose logarithmic scale for y axes, but this is not always perfect, as in Qwt it is done in a bit queer way (this may be changed in next versions of FitSuite). The colors can be set, changed in pop-up also clicking on Change Colors. Here you can change the available colors and add new ones to the list. Pushing the button *Set default* you save these settings on the computer, in order to have the same colors when you start FitSuite next time. For offspecular problems [21] we have spectrograms, and contours, which can be changed similarly. But in these cases, the levels should be set also, these may be chosen to be elements of a geometric or arithmetic sequence, by clicking in the menu of the corresponding dialog. In case of spectrograms you may create line section graphs along the edges of a polygon by choosing from the pop-up menu Polygon section, pressing the right mouse button you add the first vertex, moving the mouse to a new point and pressing spacebar a new vertex can be added. Pressing enter or the right mouse button you finish the polygon. In the profile window you may need to choose logarithmic scale. The created line section profile may not be appropriate if the axes belonging to the two independent variable have very different scale, or you have a polygon with too much vertices.

When the user would like to have an image file of the data and the fitted curve, we recommend gnuplot (or Origin, etc.). From version 1.0.4 the results are not written in files automatically (if it is not set so in Settings Export Settings). They may be exported using the menu items Results Export ... . You may choose the data which should be exported, and set the format of the created files using (Settings Export Settings)

Clicking on the menu items Plot Plot... you can choose the model (or model group) whose simulation/fit result(s) you would like to plot. Plot Plot All replots all the simulation(fit) results if you closed the plot windows before, but this works only if there was a fresh simulation(fit). Plot Close All closes all the currently open plot windows.

3.16 Sounds

Currently after the fit was finished a wav sound file is played. If you do not like this, just select another file or switch it off using Sound Settings (Settings Sound Settings).

3.17 Examples

Presently there are only a few example project files. They can be found in the directory *examples*, where we can find several subdirectories. In each one we can

find project files saved at different phase of the project definition process. The files ending with:

- *Simulation.sfp* (usually) contain simulations no data,
- *Fit.sfp* contain fits with data sets,
- *Simultan.sfp* contain several data sets fitted simultaneously.

In the directory:

- *XrayReflectionFePd* we can find projects for non-resonant X-ray reflectometry experiments.
- *ResonantXraySelfDiffusionFePd* and *XrayRes_NoResRelectionFeCr* we can find projects for resonant and non-resonant X-ray reflectometry measurements.
- *MossbauerSpectraFemtz* we can find projects for a Mössbauer experiment. The project containing simultaneous fitting problems contains three spectra for which the internal magnetic field $Hint$ and the *effective thicknesses* of sites $s2$ and $s3$ and a few other parameters were chosen to not to be correlated during cloning.
- *SMRStroboscopicMode* we can find a project for Stroboscopic Mössbauer simulations.
- *SynchrotronTransmission* we can find projects for synchrotron transmission experiments.
- *SynchrotronTimeDifferential* we can find projects for a time differential reflection experiment.
- *NeutronReflection* there are two specular simulation projects *CrFeSimulation.sfp* and *FePdSelfDiffusionSimulation.sfp* and two off-specular simulations *CrFeOffspecularSimulation.sfp, CrFeOffspecularSimulationModified.sfp*. In off-specular case the '3dimensional' results are plotted in a colored map. Usually, the color ranges are not appropriately chosen, but you change as it is written in the section 3.15. Under Windows (and sometimes under some Linuces) for some unknown reason there may be a segmentation fault, if you calculate in too much points. Therefore loading the off-specular problems set **n_omega**, **n_theta_sca**, etc. to smaller values, e.g.: 200 and 200 (or smaller).

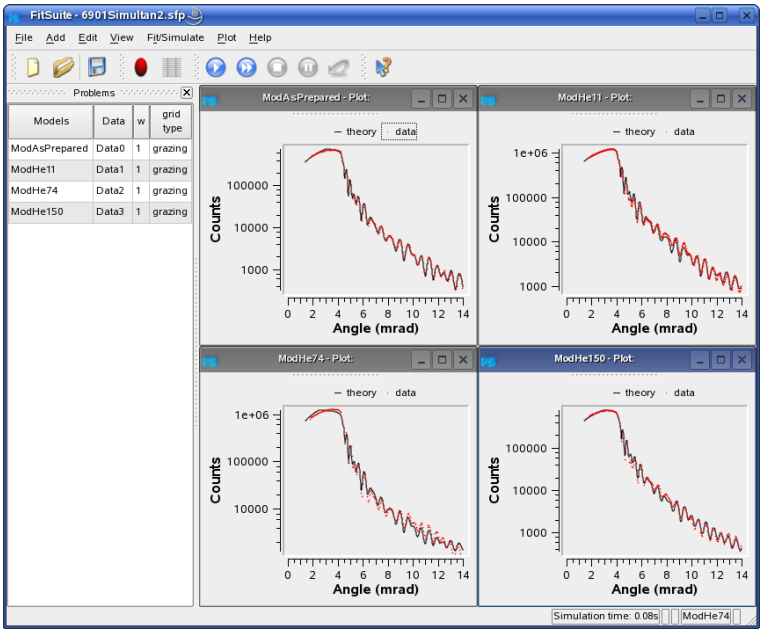

Figure 11: Fitted X-ray reflectometry spectra

- *OffspecularResonantXRay* is a project simulating off-specular problem for synchrotron radiation. It is a recently added problem. It takes a long time even to simulate, fit has not been made, tested.
- *miscellaneous* some simple functions added, just to test the fitting routines.

4 Sources, documentation

The sources containing the Fortran subroutines, Cfunctions used for simulations with the libraries created from them can be found in directory *Repositories*. Their documentation and the files used as help in the **Parameter Editor** are also there. These are still not complete.

The experimental data files are collected in *adatok* and the project files in *examples*.

5 'Installation'

The current version was tested only under **Ubuntu Linux 24.0.4 (64 bit)** and under **Windows 10**. The program is mainly written and tested under Linux, there-

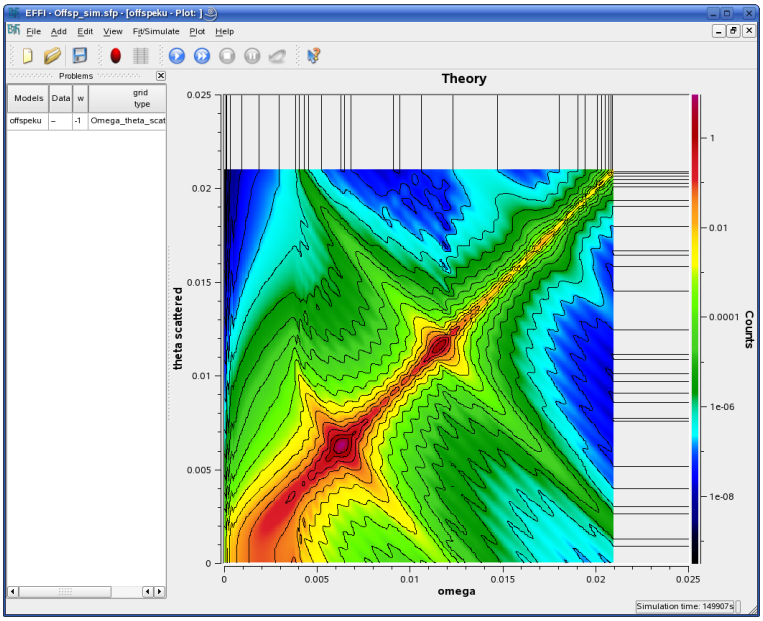

Figure 12: Off-specular synchrotron Mössbauer reflection simulation.

fore the Windows version sorrily is still much more error-prone. In principle the program can also be compiled for other Linux (Unix) distributions, earlier (Linux, Windows) versions and Mac

5.1 Linux

Download the setup file **fitsuite-2.3.0-LinuxDistribution-architecture.sh** from this web site or this site. This is a Self Extracting Tar GZip compressed packages (needs /bin/sh, tar and gunzip for extracting) and you may need to set the file permission in order to be allowed to execute. This shell script may have the following arguments −−**prefix=***full path to directory were FitSuite should be installed,* −−**help**. After starting the script from a terminal will ask, whether you accept the license or not, and that 'Do you want to include the subdirectory fitsuite.2.3.0-LinuxDistribution-architecture?'. For the last question answer boldly 'n'(o), as the program will be in *prefix/FitSuite/2.3.0/*. (If the 'keyboard repeat rate' is too fast, it may happen, that the program does not react properly and you cannot install it, in that case change your personal keyboard configuration . . .) Start the program fitsuite.2.3.0 available in this directory. I tried to include all the 'non standard' libraries needed by FitSuite, these are installed in *prefix/FitSuite/2.3.0/lib*, hopefully there will no problem with this. If the program does not find some of the libraries available here set using the command 'export LD_LIBRARY_PATH = prefix/-

FitSuite/2.3.0/lib:$LD_LIBRARY_PATH' (use full path!). If other libraries are missed by the program, please check whether they are installed on your system, etc.

The binary created for *X86_64* of course will not run under 32 bit systems, do not try to run binaries created for later Suse Linux versions in earlier distributions, as probably the required system, gcc libraries may be too old.

5.2 Windows

Download the setup file **fitsuite-2.3.0-Win32.exe** from this web site or this site, and start it.

6 License

FitSuite is a scientific software provided free as it is under the terms of GNU GPL license, **except for one additional condition:** should you use FitSuite to any extent for your publication, you should cite the article on FitSuite currently available at https://arxiv.org/abs/0907.2805v1, by mentioning also the name of the program and the version number. (e.g.: FitSuite 1.0.4) The above link will be updated when the regular journal article will be out. Please use to Forum to disseminate the bibliographic data of your published papers that make use of FitSuite.

Fitsuite uses the open source version of Qt (Qt licensing).

Source code of the main program is available on personal request to the author: (sajti.szilard@wigner.hun-ren.hu), since it is essential for us and our funding agencies to keep track of the distribution. Source code of the routines of the theories are already included in the binaries.

The program uses 3rdparty libraries Qwt5, LAPACK, TSFIT, these could be installed by the user. They are included in the directory *3rdParty* only in order to make the 'installation' of FitSuite 2.3.0 easier. Qwt has its own license (a bit modified version of GNU LGPL) given in the file 3rdParty/qwt/COPYING, for further details see that. TSFIT may be distributed under GNU GPL see 3rdParty/TSFIT/-COPYING. For LAPACK see 3rdParty/lapack-3.1.0/COPYING (it is not GPL but something like that).

References

[1] H. Spiering, L. Deák, and L. Bottyán: *Effino* Hyp. Interact. **125**:(**1-4**) (2000) 197–204. doi: 10.1023/a:1012637721433.

[2] K. Kulcsár, D. L. Nagy, and L. Pócs: *A complete package of programs for the evaluation of Mössbauer and gamma spectra* **in:** *Proc. Conf. on Mössbauer Spectrometry*, (Dresden, 1971) 594.

[3] B. Window: *Hyperfine field distributions from Mössbauer spectra* J. Phys. E: Sci. Instrum. **4**:(**5**) (1971) 401–402. doi: 10.1088/0022-3735/4/5/022.

[4] J. Hesse and A. Rübartsch: *Model independent evaluation of overlapped Mössbauer spectra* J. Phys. E: Sci. Instrum. **7**:(**7**) (1974) 526–532. doi: 10.1088/0022-3735/7/7/012.

[5] G. L. Caër and J. M. Dubois: *Evaluation of hyperfine parameter distributions from overlapped Mössbauer spectra of amorphous alloys* J. Phys. E: Sci. Instrum. **12**:(**11**) (1979) 1083–1090. doi: 10.1088/0022-3735/12/11/018.

[6] I. Vincze: *Fourier evaluation of broad Mössbauer spectra* Nucl. Instr. and Meth. **199**:(**1-2**) (1982) 247–262. doi: 10.1016/0167-5087(82)90210-1.

[7] R. A. Brand and G. L. Caër: *Improving the validity of Mössbauer hyperfine parameter distributions: The maximum entropy formalism and its applications* Nucl. Instr. and Meth. B **34**:(**2**) (1988) 272–284. doi: 10.1016/0168-583x(88)90754-9.

[8] T. Hauschild and M. Jentschel: *Comparison of maximum likelihood estimation and chi-square statistics applied to counting experiments* Nucl. Instr. and Meth. A **457**:(**1-2**) (2001) 384–401. doi: 10.1016/s0168-9002(00)00756-7.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery: *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. (New York, NY, USA: Cambridge University Press, 1992) ISBN: 0-521-43108-5. [Online]. Available: http://www.nrbook.com.

[10] B. Efron: *Bootstrap methods: Another look at the jackknife* Ann. Statist. **7**:(**1**) (1979) 1–26. doi: 10.1214/aos/1176344552.

[11] B. Efron: *The jackknife, the bootstrap and other resampling plans*. (Society for Industrial & Applied Mathematics (SIAM), 1982) doi: 10.1137/1.9781611970319.

[12] B. Efron and R. Tibshirani: *Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy* Stat. Sci. **1**:(**1**) (1986) 54–75. doi: 10.1214/ss/1177013815.

[13] B. Efron and R. J. Tibshirani: *An Introduction to the Bootstrap*. (New York: CRC, 1993)

[14] B. Winkler: *Bootstrapping goodness of fit statistics in loglinear Poisson models* Institut für Statistic, Universität München, Sonderforschungsbereich 386 Discussion Paper 53, 1996. [Online]. Available: http://epub.ub.uni-muenchen.de/1449/.

[15] C.-J. Lin: *Projected gradient methods for non-negative matrix factorization* Neural Comput. **19**:(**10**) (2007) 2756–2779. doi: 10.1162/neco.2007.19.10.2756.

[16] J. Moré, B. Garbow, and K. Hillstrom: *User guide for MINPACK-1* Argonne National Laboratory, Technical Report ANL-80-74, 1980.

[17] J. Moré, D. Sorenson, B. Garbow, and K. Hillstrom: *The MINPACK project* **in:** *Sources and Development of Mathematical Software*, W. Cowell, Ed., (Prentice-Hall, 1984) 88–111.

[18] R. L. Haupt and S. E. Haupt: *Practical genetic algorithms*. (New York, NY, USA: Wiley-Blackwell, 2004) doi: 10.1002/0471671746.

[19] Z. Michalewicz: *Genetic algorithms + data structures = evolution programs*. (Springer Science + Business Media, 1996) doi: 10.1007/978-3-662-03315-9.

[20] K. D. M. Harris, R. L. Johnston, and B. M. Kariuki: *The genetic algorithm: Foundations and applications in structure solution from powder diffraction data* Acta Cryst. **A54**:(**5**) (1998) 632–645. doi: 10.1107/s0108767398003389.

[21] L. Deák, L. Bottyán, D. L. Nagy, H. Spiering, Y. N. Khaidukov, and Y. Yoda: *Perturbative theory of grazing-incidence diffuse nuclear resonant scattering of synchrotron radiation* Phys. Rev. B **76**:(**22**) (2007) 224420. doi: 10.1103/physrevb.76.224420.

Glossary

big O notation is used to give the order of magnitude, e.g. $\mathcal{O}(h^2)$ should be read as *order of* h^2.

check box is a graphical user interface element (widget) that permits the user to make multiple selections from a number of options. Normally, check boxes are shown on the screen as a square box that can contain white space (for false) or a tick mark or X (for true). (see wikipedia)

correlation is an operation transformation matrix used to eliminate the redundancies arising because of common model parameters, see eq. (1), its inverse is the decorrelation. (see subsection 2.2)

decorrelation is the inverse of correlation, which is not unequivocal, therefore further user interaction may be needed. (see subsection 2.2)

degree of freedom (DOF) is the number of data points minus the number of fitted parameters.

distribution midrange is the middle of the distribution range.

distribution range is the width of the histogram, outside of which the approximated distribution is assumed to be negligible.

EFFI (Environment For FItting) is the name of the program written by Hartmut Spiering.

experimental scheme contains the information necessary for description of the system consisted of the experimental apparatus(es), about the experimental method performed with them and of the system under study e.g: a measured sample. (see subsection 2.1)

extraction type (spec file) is a structure, which may be defined by the user, used to collect the information needed to extract a scan from a spec file into a FitSuite data set.

feasible region in case of optimization problems with constraints is the region in the parameter space determined by the constraints where the optimum should be found.

feasible set see feasible region

fitted statistic In case of simultaneous fitting, we can have experimental data with different distributions, therefore the statistics used to fit for each fitting problem may be different. We fit a resulting statistic, their (weighted) sum. This is not a problem, as if we start from the MLE, from which all of them are derived, we would obtain also such a resulting statistic. In the case of the resulting statistic, the names like classical χ^2, Pearson's χ^2, etc. will not have any meaning, therefore in the program it is referred to just as the 'Fitted Statistic'.

goodness-of-fit statistic is a statistic measuring the goodness of a fit. (see subsection 2.4)

GUI Graphical User Interface

histogram element is the element of the vector containing all the dependent values (frequencies, or probabilities) available in the histogram.

integer parameter Some parameters are integer numbers, these are not fitted, and are handled separately from real number based parameters, in order to avoid rounding errors.

menu key is a key which usually can be found between Alt Gr and Ctrl on the Microsoft Windows-oriented keyboards with a symbol like on it. It is used to open a pop-up menu instead of right clicking. To have the proper pop-up menu the corresponding window, widget, other GUI object should have focus. (see wikipedia)

model is an *object* to which belong algorithms for calculation of characteristic spectra. (see subsection 2.1)

model parameters are all the parameters, which are needed by the models in order to calculate the spectra, without using transformation matrix technique. Usually there are a lot of common parameters, wherefore transformation matrix technique is used. (see subsection 2.2)

object The word *object* is used in several sense in this text, including its everyday meaning too. It may mean:

- in most of the cases means a physical object or concept (see the corresponding item in the glossary, and subsection 2.1)),
- a program language concept used in Object Oriented Programming.

objective function is the function whose location of minimum and/or maximum is to be found by the optimization method.

optimization method is an algorithm used to find the location(s), where a given function assumes its minimum or maximum.

parameter distribution see subsection 2.3

parameter insertion inserts a new simulation/fit parameter, inserting a new column in the transformation matrix (see subsection 2.2)

parameter vector is a vector (array) consisted of the simulation/fit or model parameters as components.

penalty function method is a method used for optimization problems with constraints, modifying the objective function by adding penalty functions outsied of the feasible region. (see subsection 2.6.1)

pop-up menu is a menu in a graphical user interface (GUI) that appears upon user interaction, such as a right mouse click or pressing the menu key, which usually can be found between Alt Gr and Ctrl on the keyboard. (see wikipedia)

property Properties in FitSuite represent the physical quantities (e.g. thickness, roughness, hyperfine field, susceptibility, electric field gradient, effective thickness, etc.) and some numbers characterizing the experimental scheme e.g. number of channels, symmetries of the sites, etc. (see subsection 2.1)

reduced χ^2 is the χ^2 divided by degree of freedom (see eq. (14))

repetition group (of physical objects) is a group of physical objects, e.g. layers which (more accurately the same sort of objects) are repeated in the same order several times.

repetition group number shows how many times the elements of the repetition group are repeated in the real physical system

root object is at the root of the object tree structure. It is analogous to the root (main in some operating systems) directory in the file systems used in computing.

simulation/fit parameters are all the parameters, which using transformation matrix technique are needed to calculate the spectra. Optimally the number of simulation/fit parameters is less than the number of model parameters, as already some redundancy is eliminated, by proper choice of the transformation matrix. (see subsection 2.2)

spec file is a file format of Certified Scientific Software's spec™ (X-Ray Diffraction and Data Acquisition software) used for experimental data in ESRF and many other places.

submodel is a *model* which is part of another model, it represents a physical system, to which we can relate intermediate results, (sub)spectra, which are used calculating the spectra measured in the experiment.

subspectrum see subsection 2.7

tooltip The tooltip is a common graphical user interface element. It is used in conjunction with a cursor, usually a mouse pointer. The user hovers the cursor over an item, without clicking it, and a small "hover box" appears with supplementary information regarding the item being hovered over (see wikipedia)

transformation matrix is a matrix (technique) used to eliminate the redundancy arising because of common parameters and/or to take into account linear interdependencies of the parameters. (see subsection 2.2)

transformation matrix split is an operation transformation matrix used to split a submatrix into two in order to have smaller, more transparent submatrices, which build up the sparse transformation matrix. (see eq. (2) and the second paragraph of subsection 2.2)

transformation matrix unification is the reverse of split, the cross-elements are set yo zero.

white space characters used to represent white space in text. (see wikipedia)

Index

! negation operator 40
? wildcard 39
& and operator 40
* wildcard 39
*.mod 23
*.sfp 2
@ (parameter name filter) 41
@c (constant parameter filter) 44
@ca (calibration constant parameter filter) 44
@Cf (correlation function name filter) 43
@cn (model defined constant parameter filter) 44
@Co (correlated parameter filter) 44
@cu (user defined constant parameter filter) 44
@cugd (current grid parameter filter) 44
@d (distributed parameter filter) 44
@de (decorrelated parameter filter) 44
@di (disabled parameter filter) 45
@dm (parameter distribution midrange filter) 44
@dr (parameter distribution range filter) 44
@fi (fix parameter filter) 43
@fr (free parameter filter) 43
@gd (grid parameter filter) 44
@hi (hidden parameter filter) 44
@hu [hidden (by the user) parameter filter] 45
@in (inserted parameter filter) 44
@m (model name filter) 41
@mg (model group name filter) 41
@mt (model type name filter) 41
@o (object name filter) 42
@og (object group filter) 43
@ot (object type name filter) 42
@p (property name filter) 42
@pc (property component name filter) 42
@rc (recently changed parameter filter) 45
@s (linked (symbolic) object name filter) 43
@smt (submodel type name filter) 42
@st (linked (symbolic) object type name filter) 43
@T (transformation matrix name filter) 43
@u (unit parameter filter) 44
[. . .] wildcard 39

A

Add
  Data 31
  New Model 23
add
  data to model 35
  model 23
addModelToGroup 54
addToMagnitude 52
addToMaximum 52
addToMinimum 51
addToResolution 52
addToValue 51
Alpha 69

B

bootstrap method 12-14, 79
bound 15, 19
brent 67
Brent‘s
  method 67

using derivatives 67
Broyden – Fletcher – Goldfarb - - Shanno (BFGS) method 69

C
C 14, 67-70, 82
c 37
C++ 71
ca 37
calculateStatistics 55
Case sensitive (filter option) 40
χ^2
classical 7
modified Neyman's 8
Pearson's 8
reduced 10
clone 63, 64
cn 37
coefficient
penalty 15
command
addModelToGroup 54
addToMagnitude 52
addToMaximum 52
addToMinimum 51
addToResolution 52
addToValue 51
argument 38
calculateStatistics 55
connectModelData 54
correlate 46
decorrelate 46
default option 45
divideMagnitude 53
divideMaximum 53
divideMinimum 53
divideResolution 53
divideValue 52
excludeDataValues 54
export 47
exportModelParameters 48
exportModelParameterTable 48
exportMPT 48
fix 47
free 47
help 55
hide 47
includeDataValues 55
insertSP 54
list 46
listUnitsOf 50
math 55
multiplyMagnitude 52
multiplyMaximum 52
multiplyMinimum 52
multiplyResolution 52
multiplyValue 52
option 45
plotModelParameterTable 49
plotMPT 49
renameModel 54
setCaseSensitive 40
setConstant 47
setMagnitude 51
setMatrixElement 53
setMatrixPartialColumn 54
setMaximum 51
setMinimum 51
setResolution 51
setStrictPattern 40
setToAbsoluteValue 53
setValue 50
setVariable 47
simulate 55
swapParameterForMatrixDiagonal 53
unhide 47
unsetCaseSensitive 40
unsetStrictPattern 40
word 38
commands 45
common parameter 3, 4, 10

compact format 32
confidence
interval 12
level 11, 12, 14, 79
connectModelData 54
constant 36
calibration 36, 37
model defined 37
user defined 37
constraint 5-7, 15-18
Contraction factor 68
convergence criterion 70, 79
correlate 38
command 46
parameters 4, 46, 63
correlation
function 24, 36
covariance matrix 12, 14, 79

D
data file formats 32
compact 32
MCA file 32, 34
one column 32
spec 34
three column 32
two column 32
data series 33
data set
replace 35
synthetic 12, 13, 79
Davidson – Fletcher – Powell (DFP)
method 69
decorrelate 38
command 46
parameters 4, 46
default option
command 45
degree of freedom 8-11, 14
derivative 12, 15, 17, 20, 21
of fitted statistic 11
displayed
number 38
distributed
parameter 6, 7, 18
distribution 34
data set
Gaussian 8, 13
Poisson 8, 13
midrange 6
range 6
divideMagnitude 53
divideMaximum 53
divideMinimum 53
divideResolution 53
divideValue 52
Dynasync 1

E
Edit
Fitting Parameters 36
Integer Parameters 38
Integer T Matrices 38
Regenerate Matrices 36, 37
T Matrices 38
EFFI 2
element
histogram 6, 7
entropy 6, 7, 18
maximum 6, 7
error
estimation 9, 11, 13, 14
experimental data 7, 9, 13, 34, 79
excludeDataValues 54
excluding bad data points 36
experimental
data 2, 34, 79
error 7, 9, 13, 34, 79
preprocessed 9, 34, 36
uncertainty 7, 9, 13, 34, 79
scheme 2, 3, 23
export

command 47
exportModelParameters
command 48
exportModelParameterTable
command 48
exportMPT
command 48
extraction type 34

F
factor 71
Contraction 68
Reflection 68
Stretch 68
file extension
mod 23
sfp 2
filter
argument 38
wildcard * 39 ? 39[. . .] 39
calibration constant parameter (@ca) 44
command 38
argument 38
word 38
constant parameter (@c) 44
correlated parameter (@Co) 44
correlation function name (@Cf) 43
current grid parameter (@cugd) 44
decorrelated parameter (@de) 44
disabled parameter (@di) 45
distributed parameter (@d) 44
fix parameter (@fi) 43
free parameter (@fr) 43
grid parameter (@gd) 44
hidden (by the user) parameter (@hu) 45
hidden parameter (@hi) 44
inserted parameter (@in) 44
linked (symbolic) object name (@s) 43
linked (symbolic) object type name (@st) 43
logical operator
and (&) 40
negation (!) 40
or ( ) 40
midranges of parameter distributions (@dm) 44
model defined constant parameter (@cn) 44
model group name (@mg) 41
model name (@m) 41
model type name (@mt) 41
object group (@og) 43
object name (@o) 42
object type name (@ot) 42
parameter name (@) 41
property component name (@pc) 42
property name (@p) 42
ranges of parameter distributions (@dr) 44
recently changed parameter (@rc) 45
submodel type name (@smt) 42
transformation matrix name (@T) 43
unit parameter (@u) 44
user defined constant parameter (@cu) 44
filter option
Case sensitive 40
Strict pattern 40
First line to read in 32
Fit/Simulation
Calculate Uncertainties 79
Fit 66
Fit Only 66
Force Simulation 66

Force Simulation of 66
Revert 78
Select Fitting Method 66, 79
Simulate 66
Simulate Only 66
FitSuite 1
Fitted Statistic 9, 11-14, 18, 34, 79
fitting
parameter 3
problem 2, 7, 8, 10, 18, 66
fix
command 47
Fletcher – Reeves method 67, 69
Fortran 2, 67, 69-71, 82
free
command 47
free parameters 36, 37, 79
ftol 71
function
correlation 24, 36
incomplete gamma 9
objective 15, 16, 18-20
penalty 15-18

G

Gaussian
MLE 8
Genetic algorithm 71
GLimit 67
gnuplot 79, 80
Golden-section search 67
Goodness Of Fit statistic 8, 9, 36
grid type 66
group
model 65, 66, 80
physical object 24
repetition
number 24
Group Model(s) 66
gtol 71
GUI 2

H

Help
What is this? 38
help
command 55
hide
command 47
curve 79
parameter 37
histogram
element 6, 7
hypothesis test 8, 9

I

import data set 31, 34
includeDataValues 55
incomplete gamma function 9
Initial simplex size 69
initial submatrix 4
Insert 24
inserting parameter 5
insertSP 54
integer
parameter 5
transformation matrix 5
Integer Parameter Editor 38
Integer Transformation Matrix Editor 38

L

Lagrange multiplier 5
λ_0 70
λ_{max} 70
λ_{min} 70
Let Be Constant 37
Let Not Be Constant 37
Levenberg – Marquardt method 70, 71
linked object 36
list
command 46

listUnitsOf 50

M
machine precision 21
magnitude 19, 38
math
  command 55
matrix
  operation 38
  split 4
  unification 5
MaxFev 71
maximum
  entropy 6, 7
  likelihood estimation 8
MaxIter 66, 68-70
MaxIterLine 67
MaxStep 69
MCA file 32, 34
mean square uncertainty 34
merge
  project 65
midrange
  distribution 6
minimum step width 19
MINPACK 71
MLE
  Gaussian 8
  Poisson 8
model 3
  group 65, 66, 80
  parameter 3
  structure 23, 63
Model Editor 23
multiplier
  Lagrange 5
multiplyMagnitude 52
multiplyMaximum 52
multiplyMinimum 52
multiplyResolution 52
multiplyValue 52

N
name convention 36
Nelder – Mead method 68
neutron reflectometry 9
Neyman's modified χ^2 8
normalization factor 9, 35, 36
normally distributed 10
Notes for data set 36
Nrep 24
Number of lines to read 32

O
objective function 15, 16, 18-20
optimization method 3
option
  command 45
or operator 40

P
parameter
  common 3, 4, 10
  correlation 4, 46, 63
  decorrelation 4, 46
  distribution 6, 7, 18
  filters 38
  fitting 3
  hidden 37
  insertion 5, 38
  integer 5
  magnitude 19, 38
  model 3
  name convention 36
  rescaling 19, 38
  resolution 19
  simulation 3
Parameter Editor 36
Pearson's χ^2 8
penalty
  coefficient 15
  function 15-18
physical

object 2
group 24
units 37
plotModelParameterTable
command 49
plotMPT
command 49
Poisson
MLE 8
Polak – Ribiere method 67, 69
Powell's method 66
preprocessed data 9, 34, 36
Problems Window 23
project
merge 65
projected gradient method 17
projection method 17
properties 3

Q
quantile 11, 14
Qwt library 79

R
range
distribution 6
reduced χ^2 10
Reflection factor 68
renameModel 54
repetition group
number 24
Replace Data 35, 63
replace data set 35
rescaling 19, 38
resolution 19
Results
Export 78, 80
Report 63
Romberg's method 20
root mean square uncertainty 34
rounding 38

S
setCaseSensitive
command 40
setConstant 47
setMagnitude 51
setMatrixElement 53
setMatrixPartialColumn 54
setMaximum 51
setMinimum 51
setResolution 51
setStrictPattern
command 40
Settings
Editor Settings 37, 38
Export Settings 78, 80
Sound Settings 80
setToAbsoluteValue 53
setValue 50
setVariable 47
sfp file 22
simulate 55
simulation
parameter 3
simultaneous
fit project 2
fitting 3, 4, 8, 10
problem 10
split
matrix 4, 38
statistic
classical χ^2 7
fitted 9, 11-14, 18, 34, 79
Gaussian MLE 8
Goodness Of Fit 8, 9, 36
Neyman's modified χ^2 8
Pearson's χ^2 8
Poisson MLE 8
reduced χ^2 10
Statistical Properties 36
Stretch factor 68
Strict pattern (filter option) 40

submatrix 3
initial 4
subspectrum 21, 22
swapParameterForMatrixDiagonal 53
synthetic data set 12, 13, 79

T
tolerance 66, 68-70
tolLine 67
TolxLnsrch 70
tooltip 38
transformation matrix 3, 36
integer 5
technique 3
Transformation Matrix Editor 38

U
uncertainty
estimation 9, 11, 13, 14
experimental data 7, 9, 13, 34, 79
type 34
unhide
command 47
unify
matrix 5, 38
units
physical 37
unsetCaseSensitive
command 40
unsetStrictPattern
command 40

V
View
Show Open spec Files 34

W
wildcard 39
* 39
? 39
[...] 39

X
xtol 71