



# **Fityk manual**

***Release 0.9.0***

January 06, 2010



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is the program for? . . . . .	1
1.2	How to read this manual . . . . .	1
1.3	GUI vs CLI . . . . .	1
<b>2</b>	<b>Getting started</b>	<b>3</b>
2.1	The minimal example . . . . .	3
2.2	Invoking fityk . . . . .	4
2.3	Graphical interface . . . . .	5
<b>3</b>	<b>Reference</b>	<b>7</b>
3.1	Data from experiment . . . . .	7
3.2	Model . . . . .	14
3.3	Fitting . . . . .	20
3.4	Settings . . . . .	23
3.5	Other commands . . . . .	24
<b>4</b>	<b>Extensions</b>	<b>27</b>
4.1	How to add your own built-in function . . . . .	27
<b>A</b>	<b>Appendix A. List of functions</b>	<b>31</b>
<b>B</b>	<b>Appendix B. Command shortenings</b>	<b>35</b>
<b>C</b>	<b>Appendix C. Literature</b>	<b>37</b>
<b>D</b>	<b>Appendix D. License</b>	<b>39</b>
<b>E</b>	<b>Appendix E. About this manual</b>	<b>41</b>



# INTRODUCTION

## 1.1 What is the program for?

Fityk is a program for nonlinear fitting of analytical functions (especially peak-shaped) to data (usually experimental data). The most concise description: peak fitting software. There are also people using it to remove the baseline from data, or to display data only.

It is reportedly used in crystallography, chromatography, photoluminescence and photoelectron spectroscopy, infrared and Raman spectroscopy, to name but a few. Although the author has a general understanding only of experimental methods other than powder diffraction, he would like to make it useful to as many people as possible.

Fityk offers various nonlinear fitting methods, simple background subtraction and other manipulations to the dataset, easy placement of peaks and changing of peak parameters, support for analysis of series of datasets, automation of common tasks with scripts, and much more. The main advantage of the program is flexibility - parameters of peaks can be arbitrarily bound to each other, e.g. the width of a peak can be an independent variable, the same as the width of another peak, or can be given by complex (and general for all peaks) formula.

Fityk is free software; you can redistribute and modify it under the terms of the GPL, version 2 or (at your option) any later version. See *Appendix C. Literature* for details. You can download the latest version of fityk from <http://www.unipress.waw.pl/fityk> (or <http://fityk.sf.net>). To contact the author, visit the same page.

## 1.2 How to read this manual

After this introduction, you may read the *Getting started*, look for tutorials in *wiki* and postpone reading the manual until you need to write a script, put constraints on variables, add user-defined function or understand better how the program works.

In case you are not familiar with the term *weighted sum of squared residuals* or you are not sure how it is weighted, have a look at *Nonlinear optimization*. Remember that you must set correctly *standard deviations* of y's of points, otherwise you will get wrong results.

## 1.3 GUI vs CLI

The program comes in two versions: the GUI (Graphical User Interface) version - more comfortable for most users, and the CLI (Command Line Interface) version (named *cfityk* to differentiate, Unix only).

If the CLI version was compiled with the *GNU Readline Library*, command line editing and command history as per *bash* will be available. Especially useful is TAB-expanding. Data and curves fitted to data are visualized with *gnuplot* (if it is installed).

The GUI version is written using the *wxWidgets* library and can be run on Unix species with GTK+ and on MS Windows. There are also people using it on MacOS X (have a look at the fityk-users mailing list archives for details).



# GETTING STARTED

## 2.1 The minimal example

Let us analyze a diffraction pattern of NaCl. Our goal is to determine the position of the center of the highest peak. It is needed for calculating the pressure under which the sample was measured, but this later detail in the processing is irrelevant for the time being.

The data file used in this example is distributed with the program and can be found in the `samples` directory.

First load data from file `nacl01.dat`. You can do this by typing:

```
@0 < nacl01.dat
```

in the CLI version (or in the GUI version in the input box - at the bottom, just above the status bar). In the GUI, you can also select *Data* → *Load File* from the menu and choose the file.

If you use the GUI, you can zoom-in to the biggest peak using left mouse button on the auxiliary plot (the plot below the main plot). To zoom out, press the **View whole** toolbar button. Other ways of zooming are described in *Mouse usage*. If you want the data to be drawn with larger points or a line, or if you want to change the color of the line or background, press right mouse button on the main plot and use **Data point size** or **Color** menu from the pop-up menu. To change the color of data points, use the right-hand panel.

Now all data points are active. Because only the biggest peak is of interest for the sake of this example, the remaining points can be deactivated. Type:

```
A = (23.0 < x < 26.0)
```

or change to *range* mode (press **Data-Range Mode** button on toolbar) and select range to be deactivated with right mouse button.

As our example data has no background to worry about, our next step is to define a peak with reasonable initial values and fit it to the data. We will use Gaussian. To see its formula, type: `info Gaussian` or look for it in the documentation (in *Appendix A. List of functions*). Incidentally, most of the commands can be abbreviated, e.g. you can type: `i Gaussian`.

To define peak, type:

```
%p = Gaussian(~60000, ~24.6, ~0.2); F = %p
```

or:

```
%p = guess Gaussian
```

or select **Gaussian** from the list of functions on the toolbar and press the **auto-add** toolbar button. There are also other ways to add peak in GUI such as *add-peak* mode. These mouse-driven ways give function a name like `%_1`, `%_2`, etc.

Now let us fit the function. Type: `fit` or select *Fit* → *Run* from the menu (or press the toolbar button).

When fitting, the weighted sum of squared residuals (see *Nonlinear optimization*) is being minimized.

**Note:** The default *weights of points* are not equal.

To see the peak parameters, type: `info+ %p` or (in the GUI) move the cursor to the top of the peak and try out the context menu (right button), or use the right-hand panel.

That's it! To do the same a second time (for example to a similar data set) you can write all the commands to a file:

```
commands > myscript.fit
```

and later use it as script:

```
commands < myscript.fit
```

Alternatively, use *Session* → *Logging* → *History dump* and *Session* → *Execute script*.

If you start fityk from command line, you can load data and/or execute scripts by giving filenames as arguments, e.g.

```
bash$ fityk myscript.fit
```

## 2.2 Invoking fityk

On startup, the program executes a script from the `$HOME/.fityk/init` file (on MS Windows XP: `C:\Documents and Settings\USERNAME\.fityk\init`). Following this, the program executes command passed with `--cmd` option, if given, and processes command line arguments:

- if the argument starts with `=>`, the string following `=>` is regarded as a command and executed (otherwise, it is regarded as a filename).
- if the filename has extension `.fit` or the file begins with a `# Fityk` string, it is assumed to be a script and is executed.
- otherwise, it is assumed to be a data file. It is possible to specify columns in data file in this way: `file.xy:1:4::`. Multiple y columns can be specified (`file.xy:1:3,4,5::` or `file.xy:1:3..5::`) – it will load each y column as a separate dataset, with the same values of x.

There are also other parameters to the CLI and GUI versions of the program. Option `-h` (on MS Windows `/h`) gives the full listing:

```
wojdyr@ubu:~/fityk/src$ ./fityk -h
Usage: fityk [-h] [-V] [-c <str>] [-I] [-r] [script or data file...]
-h, --help                show this help message
-V, --version             output version information and exit
-c, --cmd=<str>           script passed in as string
-g, --config=<str>       choose GUI configuration
-I, --no-init             don't process $HOME/.fityk/init file
-r, --reorder             reorder data (50.xy before 100.xy)
```

The example of non-interactive using CLI version on Linux:

```
wojdyr@ubu:~/foo$ cfityk -h
Usage: cfityk [-h] [-V] [-c <str>] [script or data file...]
-h, --help                show this help message
-V, --version             output version information and exit
-c, --cmd=<str>           script passed in as string
-I, --no-init             don't process $HOME/.fityk/init file
-q, --quit                don't enter interactive shell
wojdyr@ubu:~/foo$ ls \*.rdf
```

```

dat_a.rdf dat_r.rdf out.rdf
wojdyr@ubu:~/foo$ cfityk -q -I "--> set verbosity=quiet, autoplot=never" \
> \*.rdf "--> i+ min(x if y > 0) in @*"
in @0 dat_a: 1.8875
in @1 dat_r: 1.5105
in @2 out: 1.8305

```

## 2.3 Graphical interface

### 2.3.1 Plots and other windows

The GUI window of fityk consists of (from the top): menu bar, toolbar, main plot, auxiliary plot, output window, input field, status bar and of sidebar at right-hand side. The input field allows you to type and execute commands in a similar way as is done in the CLI version. The output window (which is configurable through a pop-up menu) shows the results. Incidentally, all GUI commands are converted into text and are visible in the output window, providing a simple way to learn the syntax.

The main plot can display data points, model that is to be fitted to the data and component functions of the model. Use the pop-up menu (click right button on the plot) to configure it. Some properties of the plot (e.g. colors of data points) can be changed using the sidebar.

One of the most useful things which can be displayed by the auxiliary plot is the difference between the data and the model (also controlled by a pop-up menu). Hopefully, a quick look at this menu and a minute or two's worth of experiments will show the potential of this auxiliary plot.

Configuration of the GUI (visible windows, colors, etc.) can be saved using *GUI* → *Save current config*. Two different configurations can be saved, which allows easy changing of colors for printing. On Unix platforms, these configurations are stored in a file in the user's home directory. On Windows - they are stored in the registry (perhaps in the future they will also be stored in a file).

### 2.3.2 Mouse usage

The usage of the mouse on menu, dialog windows, input field and output window is (hopefully) intuitive, so the only remaining topic to be discussed here is how to effectively use the mouse on plots.

Let us start with the auxiliary plot. The right button displays a pop-up menu with a range of options, while the left allows you to select the range to be displayed on the x-axis. Clicking with the middle button (or with left button and *Shift* pressed simultaneously) will zoom out to display all data.

On the main plot, the meaning of the left and right mouse button depends on current *mode* (selected using either the toolbar or menu). There are hints on the status bar. In normal mode, the left button is used for zooming and the right invokes the pop-up menu. The same behaviour can be obtained in any mode by pressing *Ctrl* (or *Alt*).

The middle button can be used to select a rectangle that you want to zoom in to. If an operation has two steps, such as rectangle zooming (i.e. first you press a button to select the first corner, then move the mouse and release the button to select the second corner of the rectangle), this can be cancelled by pressing another button when the first one is pressed.



# REFERENCE

In this part of the manual:

- all features, with exception of the graphical interface features, are described,
- several concepts (such as *simple-variable* and *compound-variable*), that reflect the internal design of the program, are introduced,
- the syntax of the fityk mini-language is explained,
- it is rarely mentioned, that in the GUI typing the commands can be usually avoided and most of the operations can be done with mouse clicking.

The fityk mini-language consists of *commands*.

Basically, there is one command per line. If for some reason it is more comfortable to place more than one command in one line, they can be separated with a semicolon (;).

Most of the commands can have arguments separated by a comma (,), e.g. `delete %a, %b, %c`.

Most of the commands can be shortened: e.g. you can type `inf` or `in` or `i` instead of `info`. See [Appendix B. Command shortenings](#) for details.

The symbol '#' starts a comment - everything from the hash (#) to the end of the line is ignored.

## 3.1 Data from experiment

### 3.1.1 Loading data

The basic file format is ascii text file with every line corresponding to one data point. If there are more than two columns of numbers, it can be specified which columns corresponds to x and y, and, optionally, also sigma. Numbers in line can be separated by whitespace, commas or semicolons. Lines that can't be read as numbers are ignored.

The `xylib` library is used to read data from file. New formats can be easily added.

Points are loaded from files using the command:

```
dataslot < filename[:xcol:ycol:scol:block] [filetype options...]
```

where

- `dataslot` should be replaced with `@0`, unless many datasets are to be used simultaneously (for details see: [Working with multiple datasets](#)),
- `xcol`, `ycol`, `scol` (supported only in text file) are columns corresponding to x, y and std. dev. of y. Column 0 means index of the point: 0 for the first point, 1 for the second, etc.
- `block` is only supported by formats with multiple blocks of data.

- *filetype* usually can be omitted, because in most of the cases the filetype can be detected; the list of supported filetypes is at the end of this section
- *options* depend on a filetype and usually are omitted

If the filename contains blank characters, a semicolon or comma, it should be put inside single quotation marks (together with colon-separated indices, if any).

Multiple y columns and/or blocks can be specified, see the examples below:

```
@0 < foo.vms
@0 < foo.fii text first-line-header
@0 < foo.dat:1:4:: # x,y - 1st and 4th columns
@0 < foo.dat:1:3,4:: # load two dataset (with y in columns 3,4)
@0 < foo.dat:1:3..5:: # load three dataset (with y in columns 3,4,5)
@0 < foo.dat:1:4..6,2:: # load four dataset (y: 4,5,6,2)
@0 < foo.dat:1:2...: # load 2nd and all the next columns as y
@0 < foo.dat:1:2:3: # read std. dev. of y from 3rd column
@0 < foo.dat:0:1:: # x - 0,1,2,..., y - first column
@0 < 'foo.dat:0:1::' # the same
@0 < foo.raw:::0,1 # load two first blocks of data (as one dataset)
```

Information about loaded data can be obtained with:

```
info data [in @0]
```

## Supported filetypes

**text** ASCII format. If option first-line-header is given, the first line is read as title.

**dbws** format used by DBWS (program for Rietveld analysis) and DMPLLOT.

**cpi** Sietronics Sieray CPI format

**uxd** Siemens/Bruker UXD format (powder diffraction data)

**bruker\_raw** Simens-Bruker RAW format (version 1,2,3)

**canberra\_mca** Spectral data stored by Canberra MCA systems

**rigaku\_dat** Rigaku dat format (powder diffraction data)

**vamas** VAMAS ISO-14976 (only experiment modes: “SEM” or “MAPSV” or “MAPSVDP” and only “REGULAR” scan mode are supported)

**philips\_udf** Philips UDF (powder diffraction data)

**philips\_rd** Philips RD raw scan format V3 (powder diffraction data)

**spe** Princeton Instruments WinSpec SPE format (only 1-D data is supported)

**pdCIF** CIF for powder diffraction

... what else would you like to have here?

### 3.1.2 Active and inactive points

We often have the situation that only a part of the data from a file is of interest. In fityk, each point is either *active* or *inactive*. Inactive points are excluded from fitting and all calculations. A data *transformation*:

```
A = boolean-condition-to-be-active
```

can be used to change the state of points.

In the GUI, there is a `Data-Range Mode` that allows to activate and deactivate points with mouse.

### 3.1.3 Standard deviation (or weight)

When fitting data, we assume that only the y coordinate is subject to statistical errors in measurement. This is a common assumption. To see how the y standard deviation  $\sigma$  influences fitting (optimization), look at the weighted sum of squared residuals formula in *Nonlinear optimization*. We can also think about weights of points – every point has a weight assigned, that is equal  $w_i = 1/\sigma_i^2$ .

Standard deviation of points can be *read from file* together with the x and y coordinates. Otherwise, it is set either to  $\max(\sqrt{y}, 1.0)$  or to 1, depending on the value of *data-default-sigma* option. Setting std. dev. as a square root of the value is common and has theoretical ground when y is the number of independent events. You can always change standard deviation, e.g. make it equal for every point with command: `S=1`. See *Data transformations* for details.

**Note:** It is often the case that user is not sure what standard deviation should be assumed, but it is her responsibility to pick something.

### 3.1.4 Data transformations

Every data point has four properties: x coordinate, y coordinate, standard deviation of y and active/inactive flag. Lower case letters x, y, s, a stand for these properties before transformation, and upper case X, Y, S, A for the same properties after transformation. M stands for the number of points.

Data can be transformed using assignments. Command `Y=-y` will change the sign of the y coordinate of every point.

You can apply transformation to selected points: `Y[3]=1.2` will change point with index 3 (which is 4th point, because first has index 0), and `Y[3..6]=1.2` will do the same for points with indices 3, 4, 5, but not 6. `Y[2..]=1.2` will apply the transformation to points with index 2 and above. You can guess what `Y[. . 6]=1.2` does.

Most of operations are executed sequentially for points from the first to the last one. n stands for the index of currently transformed point. The sequence of commands:

```
M=500; x=n/100; y=sin(x)
```

will generate the sinusoid dataset with 500 points.

If you have more than one dataset, you have to specify explicitly which dataset transformation applies to. See *Working with multiple datasets* for details.

**Note:** Points are kept sorted according to their x coordinate, so changing x coordinate of points will also change the order and indices of points.

Expressions can contain:

- real numbers in normal or scientific format (e.g. 1.23e5),
- constant pi,
- binary operators: +, -, \\*, /, ^,
- one argument functions:
  - sqrt
  - exp
  - log10
  - ln
  - sin
  - cos
  - tan

- sinh
- cosh
- tanh
- atan
- asin
- acos
- erf
- erfc
- gamma
- lgamma (=ln(lgamma ()))
- abs
- round (rounds to the nearest integer)

- two argument functions:

- min2
- max2 (e.g. max2 (3, 5) will give 5),
- randuniform(a, b) (random number from interval (a, b)),
- randnormal(mu, sigma) (random number from normal distribution),
- voigt(a, b) =  $\frac{b}{\pi} \int_{-\infty}^{+\infty} \frac{\exp(-t^2)}{b^2+(a-t)^2} dt$

- ternary ?: operator: condition ? expression1 : expression2, which performs *expression1* if condition is true and *expression2* otherwise. Conditions can be built using boolean operators and comparisons: AND, OR, NOT, >, >=, <, <=, ==, != (or <>), TRUE, FALSE.

The value of a data expression can be shown using the command `info`, see examples at the end of this section.

Linear interpolation of *y* (or any other property: *s,a,X,Y,S,A*) between two points can be calculated using special syntax:

```
y[x=expression]
```

If the given *x* is outside of the current data range, the value of the first/last point is returned.

**Note:** All operations are performed on real numbers.

Two numbers that differ less than *epsilon* (see *option epsilon*) i.e.  $\text{abs}(a-b) < \text{epsilon}$ , are considered equal.

Indices are also computed in real number domain, and then rounded to the nearest integer.

Transformations can be joined with comma (,), e.g.

```
X=y, Y=x
```

swaps axes.

Before and after executing transformations, points are always sorted according to their *x* coordinate. You can temporarily change the order of points using `order=t`, where *t* is one of *x, y, s, a, -x, -y, -s, -a*. This only makes sense for a sequence of transformations (joined with comma), as after finishing each transformation points will be reordered again. This feature is rarely useful.

Points can be deleted using the following syntax:

```
delete[index-or-range]
```

or

`delete(condition)`

and created simply by increasing the value of `M`.

There are two parametrized functions: `spline` and `interpolate`. The general syntax is:

```
parametrizedfunc [param1, param2](expression)
```

e.g.

```
spline[22.1, 37.9, 48.1, 17.2, 93.0, 20.7](x)
```

will give the value of a *cubic spline interpolation* through points (22.1, 37.9), (48.1, 17.2), ... in `x`. Spline function is used for manual background subtraction via the GUI. Function `interpolate` is similar, but gives a *polyline interpolation*.

There are also aggregate functions:

- `min` (the smallest value),
- `max` (the largest value),
- `sum` (sum of all values),
- `avg` (arithmetic mean of all values),
- `stddev` (standard deviation of all values),
- **`darea` (`darea(y)` gives the interpolated area under data points**, and can be used to normalize the area. `darea` is implemented as  $t*(x[n+1]-x[n-1])/2$ , where  $t$  is the value of the *expression*).

They have two forms:

```
aggregatefunc(expression)
```

```
aggregatefunc(expression if condition)
```

In the first form the value of *expression* is calculated for all points. In the second, only the points for which the *condition* is true are taken into account.

True value in data expression is represented numerically by 1., and false by 0, so `sum` can be also used to count points that fulfil given criteria.

A few examples:

```
Y[1...] = Y[n-1] + y[n] # integrate
x[...-1] = (x[n]+x[n+1])/2; # reduces
y[...-1] = y[n]+y[n+1]; # two times
delete(n%2==1) # number of points
delete(not a) # delete inactive points
X = 4*pi * sin(x/2*pi/180) / 1.54051 # changes x scale (2theta -> Q)
# make equal step, keep the number of points the same
X = x[0] + n * (x[M-1]-x[0]) / (M-1), Y = y[x=X], S = s[x=X], A = a[x=X]
# take the first 2000 points, average them and subtract as background
Y = y - avg(y if n<2000)
# fityk can also be used as a simple calculator
i 2+2 #4
i sin(pi/4)+cos(pi/4) #1.41421
i gamma(10) #362880
# examples of aggregate functions
i max(y) # the largest y value
i sum(y>avg(y)) # the number of points which have y value greater than arithmetic mean
Y = y / darea(y) # normalize data area
i darea(y-F(x) if 20<x<25)
```

There is also another kind of transformations, *dataset transformation*, which operate on a whole dataset, not single points. The syntax (for one dataset) is:

```
@0 = dataset-transformation @0
```

where *dataset-transformation* can be one of:

**sum\_same\_x** Merges points which distance in  $x$  is smaller than *epsilon*.  $x$  of a merged point is the average, and  $y$  and sigma are sums of components.

**avg\_same\_x** The same as `sum_same_x`, but  $y$  and sigma of a merged point is set as an average of components.

**shirley\_bg** Calculates Shirley background (useful in X-ray photoelectron spectroscopy).

**rm\_shirley\_bg** Calculates data with removed Shirley background.

### 3.1.5 Functions and variables in data transformation

information in this section are not often used in practice. Read it after reading *Model*.

Variables (`$foo`) and functions (`%bar`) can be used in data transformations, and a current value of data expression can be assigned to a variable. Values of the function parameters (e.g. `%fun.a0`) and pseudo-parameters Center, Height, FWHM and Area (e.g. `%fun.Area`) can also be used. Pseudo-parameters are supported only by functions, which know how to calculate these properties.

It is possible to calculate some properties of %functions:

- `numarea(%f, x1, x2, n)` gives area integrated numerically from  $x1$  to  $x2$  using trapezoidal rule with  $n$  equal steps.
- `findx(%f, x1, x2, y)` finds  $x$  in interval  $(x1, x2)$  such that  $\%f(x)=*y*$  using bisection method combined with Newton-Raphson method. It is a requirement that  $\%f(x1) < y < \%f(x2)$ .
- `extremum(%f, x1, x2)` finds  $x$  in interval  $(x1, x2)$  such that  $\%f'(x)=0$  using bisection method. It is a requirement that  $\%f'(x1)$  and  $\%f'(x2)$  have different signs.

A few examples:

```
$foo = {y[0]} # data expression can be used in variable assignment
$foo2 = {y[0] in @0} # dataset can be given if necessary
Y = y / $foo # and variables can be used in data transformation
Y = y - %f(x) # subtracts function %f from data
Y = y - @0.F(x) # subtracts all functions in F
Z += Constant(~0) # fit constant x-correction (this can be caused...
fit # ...by a shift in scale of the instrument collecting data),
X = x + @0.Z(x) # ...remove it from the dataset,
Z = 0 # ...and clear the x-correction in the model
info numarea(%fun, 0, 100, 10000) # shows area of function %fun
info %fun.Area # it is not always supported
info %_1(extremum(%_1, 40, 50)) # shows extremum value
# calculate FWHM numerically, value 50 can be tuned
$c = {%f.Center}
i findx(%f, $c, $c+50, %f.Height/2) - findx(%f, $c, $c-50, %f.Height/2)
i %f.FWHM # should give almost the same.
```

### 3.1.6 Working with multiple datasets

Let us call a set of data that usually comes from one file – a *dataset*. All operations described above assume only one dataset. If there are more datasets created, it must be explicitly stated which dataset the command is being applied to, e.g. `M=500 in @0`. Datasets have numbers and are referenced by '@' with the number, e.g. `@3`. `@*` means all datasets (e.g. `Y=y/10 in @*`).

To load dataset from file, use one of commands:

```
@n < filename:xcol:ycol:scol:block filetype options...
```

```
@+ < filename:xcol:ycol:scol:block filetype options...
```

The first one uses existing data slot and the second one creates a new slot. Using @+ increases the number of datasets, and command `delete @n` decreases it.

The syntax:

```
@n = dataset-transformation @m + @k + ...
```

```
@+ = dataset-transformation @m + @k + ...
```

can be used for example:

- to duplicate a dataset (`@+ = @n`),
- to create a new dataset as a sum of two or more existing sets (`@+ = @n + @m + ...`),
- to perform *dataset transformations*, e.g. to remove Shirley background (`@n = rm_shirley_bg @n`).

A sum of datasets contains all points from all component datasets. If you want to merge points with the same x value, use:

```
@+ = sum_same_x @n + @m + ...
```

Each dataset has a separate *model*, that can be fitted to the data. This is explained in the next chapter.

Each dataset also has a title (it does not have to be unique, however). When loading file, a title is automatically created, either using the filename or by reading it from the file (depending on the format of the file). Titles can be changed using the command:

```
set @n.title=new-title
```

To print the title of the dataset, type `info title in @n`.

You calculate values of a data expression for each dataset and print a list of results, e.g. `i+ avg(y) in @*`.

### 3.1.7 Exporting data

Command:

```
info dataslot (expression , ...) > file.tsv
```

can export data to an ASCII TSV (tab separated values) file.

To export data in a 3-column (x, y and standard deviation) format, use:

```
info @0 (x, y, s) > file.tsv
```

If `a` is not listed in the list of columns, like in the example above, only the active points are exported.

All expressions that can be used on the right-hand side of data transformations can also be used in the column list. Additionally, `F` and `Z` can be used with dataset prefix, e.g.

```
info @0 (n+1, x, y, F(x), y-F(x), Z(x), %foo(x), a, sin(pi*x)+y^2) > file.tsv
```

## 3.2 Model

### 3.2.1 Model - Introduction

The *model*  $F$  (the function that is fitted to the data) is computed as a sum of *component functions*,  $F = \sum_i f_i$ . Each component function is one of the functions defined in the program, such as Gaussian or polynomial.

To avoid confusion we will always use:

- the name *model* when referring to the total function fitted to data.
- and the name *function* only when referring to a component function.

Function  $f_i = f_i(x; \mathbf{a})$  is a function of  $x$ , and depends on a vector of parameters  $\mathbf{a}$ . This vector contains all fitted parameters.

Because we often have the situation, that the error in the  $x$  coordinate of data points can be modeled with function  $Z(x; \mathbf{a})$ , we introduce this term to the model, and the final formula is:

$$F(x; \mathbf{a}) = \sum_i f_i(x + Z(x; \mathbf{a}); \mathbf{a})$$

where  $Z(x; \mathbf{a}) = \sum_i z_i(x; \mathbf{a})$

Note that the same  $x$ -correction  $Z$  is used in all functions  $f_i$ .

Now we will have a closer look at component functions. Every function  $f_i$  has a type chosen from the function types available in the program. The same is true about functions  $z_i$ . One of these types is the *Gaussian*. It has the following formula:

$$f_G(x; a_0, a_1, a_2) = a_0 \exp \left[ -\ln(2) \left( \frac{x - a_1}{a_2} \right)^2 \right]$$

There are three parameters of Gaussian. These parameters do not depend on  $x$ . There must be one *variable* bound to each function's parameter.

### 3.2.2 Variables

Variables in Fityk have names prefixed with the dollar symbol (\$). A variable is created by assigning a value to it, e.g.

```
$foo=~5.3
$c=3.1
$bar=5*sin($foo)
```

The variables like the first one, `$foo`, created by assigning to it a real number prefixed with '~', will be called *simple-variables*. The '~' means that the value assigned to the variable can be changed when fitting the model to the data.

Each simple-variable is independent. In optimization terms, it corresponds to one dimension of the space where we will look for the minimum.

In the above example, the variable `$c` is actually a *constant*. `$bar` depends on the value of `$foo`. When `$foo` changes, the value of `$bar` also changes. Variables like `$bar` will be called *compound-variables*. Compound-variables can be build using operators `+`, `-`, `*`, `/`, `^` and the functions `sqrt`, `exp`, `log10`, `ln`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `atan`, `asin`, `acos`, `erf`, `erfc`, `lgamma`, `abs`, `voigt`. This is a subset of the functions used in *data transformations*.

Variables can be used in data tranformations, e.g. `Y=y/$a`.

The value of the data expression can be used in the variable definition, but it must be inside braces, e.g. `$bleh={3+5}` or, to create a simple variable: `$bleh=~{3+5}`.

Sometimes it is useful to freeze a variable, i.e. to prevent it from changing while fitting. There is no special syntax for it, but it can be done using data expressions in this way:

```
$a = ~12.3 # $a is fittable
$a = {$a} # $a is not fittable
$a = ~{$a} # $a is fittable again
```

It is also possible to define a variable as e.g. `$bleh=~9.1*exp(~2)`. In this case two simple-variables (with values 9.1 and 2) are created automatically.

Automatically created variables are named `$_1`, `$_2`, `$_3`, and so on.

Variables can be deleted using the command:

```
delete $variable
```

Some fitting algorithms need to randomize the parameters of the fitted function (i.e. they need to randomize simple variables). For this purpose, the simple variable can have a specified *domain*. Note that the domain does not imply any constraints on the value the variable can have – it is only a hint for fitting algorithms. Domains are used by Nelder-Mead method and Genetic Algorithms. The syntax is as follows:

```
$a = ~12.3 [11 +- 5] # center and width of the domain are given
$b = ~12.3 [ +- 5] # if the center of the domain is not specified,
                  # the value of the variable is used
```

If the domain is not specified, the value of `variable-domain-percent` option is used (domain is `+/- value-of-variable * variable-domain-percent / 100`)

### 3.2.3 Function types and functions

Let us go back to functions. Function types have names that start with upper case letter, e.g. `Linear` or `Voigt`. Functions (i.e. function instances) have names prefixed with a percent symbol, e.g. `%func`. Every function has a type and variables bound to its parameters.

`info types` shows the list of available function types. `info FunctionType` (e.g. `info Pearson7`) shows formula of the *FunctionType*.

Functions can be created by giving the type and the correct number of variables in brackets, e.g.

```
%f1 = Gaussian(~66254., ~24.7, ~0.264)
%f2 = Gaussian(~6e4, $ctr, $b+$c)
%f3 = Gaussian(height=~66254., hwhm=~0.264, center=~24.7)
```

Every expression which is valid on the right-hand side of a variable assignment can be used as a variable. If it is not just a name of a variable, an automatic variable is created. In the above examples, two variables were implicitly created for `%f2`: first for value `6e4` and the second for `$b+$c`.

If the names of function's parameters are given (like for `%f3`), the variables can be given in any order.

Function types can have specified default values for some parameters. The variables for such parameters can be omitted, e.g.:

```
=> i Pearson7
Pearson7(height, center, hwhm, shape=2) = height / (1 + ((x-center)/hwhm)^2 * (2^(1/shape)-1))^shape
=> %f4 = Pearson7(height=~66254., center=~24.7, fwhm=~0.264) # no shape is given
New function %f4 was created.
```

A deep copy of function (i.e. all variables that it depends on are also copied) can be made using the command:

```
%function = copy(%another_function)
```

Functions can be also created with the command `guess`, as described in [Guessing peak location](#).

You can change a variable bound to any of the function parameters in this manner:

```
--> %f = Pearson7(height=~66254., center=~24.7, fwhm=~0.264)
New function %f was created.
--> %f.center=~24.8
--> $h = ~66254
--> %f.height=$h
--> info %f
%f = Pearson7($h, $_5, $_3, $_4)
--> $h = ~60000 # variables are kept by name, so this also changes %f
--> %p1.center = %p2.center + 3 # keep fixed distance between %p1 and %p2
```

Functions can be deleted using the command:

```
delete %function
```

### 3.2.4 User-defined functions (UDF)

User-defined function types can be created using command `define`, and then used in the same way as built-in functions.

Example:

```
define MyGaussian(height, center, hwhm) = height*exp(-ln(2)*((x-center)/hwhm)^2)
```

- The name of new type must start with an upper-case letter, contain only letters and digits and have at least two characters.
- The name of the type is followed by parameters in brackets.
- Parameter name must start with lowercase letter and, contain only lowercase letters, digit and the underscore ('\_').
- The name "x" is reserved, do not put it into parameter list, just use it on the right-hand side of the definition.
- There are special names of parameters, that fityk understands:
  - if the functions is peak-like: `height`, `center`, `fwhm`, `area`, `hwhm`,
  - if the function is more like linear: `slope`, `intercept`, `avgy`.

Parameters with such names do not need default values. `fwhm` mean full width at half maximum (FWHM), `hwhm` means half width..., i.e. `fwhm/2`.

- Each parameter should have a default value (see examples below). Default values allow adding a peak with the command `guess` or with one click in the GUI.
- The default value can be a number or expression that contains the special names listed above with exception of `hwhm` (use `fwhm/2` instead).

UDFs can be defined either by giving a full formula, or as a sum of already defined functions, with possible *re-parametrization* (see `GaussianArea` and `GLSum` below for the example of the latter).

When giving a full formula, right-hand side of the equality sign is similar to the *definiton of variable*, but the formula can also depend on  $x$ . Hopefully the examples at the end of this section make the syntax clear.

#### How it works internally

The formula is parsed, derivatives of the formula are calculated symbolically, all expressions are simplified (but there is a lot of space for optimization here) and bytecode for virtual machine (VM) is created.

When fitting, the VM calculates the value of the function and derivatives for every point.

Possible (i.e. not implemented) optimizations include Common Subexpression Elimination and JIT compilation.

There is a simple substitution mechanism that makes writing complicated functions easier. Substitutions must be assigned in the same line, after keyword `where`. Example:

```
define ReadShockley(sigma0=1, a=1) = sigma0 * t * (a - ln(t)) where t=x*pi/180

# more complicated example, with nested substitutions
define FullGBE(k, alpha) = k * alpha * eta * (eta / tanh(eta) - ln (2*sinh(eta))) where eta = 2*pi
```

**Tip:** Use the `init` file for often used definitions. See *Invoking fityk* for details.

Defined functions can be undefined using command `undefine`.

Examples:

```
# first how some built-in functions could be defined
define MyGaussian(height, center, hwhm) = height*exp(-ln(2)*((x-center)/hwhm)^2)
define MyLorentzian(height, center, hwhm) = height/(1+((x-center)/hwhm)^2)
define MyCubic(a0=height, a1=0, a2=0, a3=0) = a0 + a1*x + a2*x^2 + a3*x^3
# supersonic beam arrival time distribution
define SuBeArTiDi(c, s, v0, dv) = c*(s/x)^3*exp(-((s/x)-v0)/dv)^2/x
# area-based Gaussian can be defined as modification of built-in Gaussian
# (it is the same as built-in GaussianA function)
define GaussianArea(area, center, hwhm) = Gaussian(area/fwhm/sqrt(pi*ln(2)), center, hwhm)
# sum of Gaussian and Lorentzian, a.k.a PseudoVoigt (should be in one line)
define GLSum(height, center, hwhm, shape) = Gaussian(height*(1-shape), center, hwhm)
+ Lorentzian(height*shape, center, hwhm)
# to change definition of UDF, first undefine previous definition
undefine GaussianArea
```

### 3.2.5 Speed of computations

With default settings, the value of every function is calculated at every point. Functions such as Gaussian often have non-negligible values only in a small fraction of all points. To speed up the calculation, set the option `cut-function-level` to a non-zero value. For each function the range with values greater than `cut-function-level` will be estimated, and all values outside of this range are considered to be equal zero. Note that not all functions support this optimization.

If you have a number of loaded dataset, and the functions in different datasets do not share parameters, it is faster to fit the datasets sequentially (`fit in @0; fit in @1; ...`) then parallelly (`fit in @*`).

Each simple-variable slows down the fitting, although this is often negligible.

### 3.2.6 Model, F and Z

As already discussed, each dataset has a separate model that can be fitted to the data. As can be seen from the *formula above*, the model is defined as a set functions  $f_i$  and a set of functions  $z_i$ . These sets are named  $F$  and  $Z$  respectively. The model is constructed by specifying names of functions in these two sets.

In many cases *x-correction*  $Z$  is not used. The fitted curve is thus the sum of all functions in  $F$ .

Command

```
F += %function
```

adds *%function* to  $F$ , command

```
Z += %function
```

adds *%function* to Z.

To remove *%function* from F (or Z) either do:

```
F -= %function
```

or delete *%function*.

If there is more than one dataset, F and Z must be prefixed with the dataset number (e.g. @1.F += *%function*).

The following syntax is also valid:

```
# create and add function to F
%g = Gaussian(height=~66254., hwhm=~0.264, center=~24.7)
@0.F += %g

# create automatically named function and add it to F
@0.F += Gaussian(height=~66254., hwhm=~0.264, center=~24.7)

# clear F
@0.F = 0

# clear F and put three functions in it
@0.F = %a + %b + %c

# show info about the first and the last function in @0.F
info @0.F[0], @0.F[-1]

# the same as %bcp = copy(%b)
%bcp = copy(@0.F[1])

# make @1.F the exact (shallow) copy of @0.F
@1.F = @0.F

# make @1.F a deep copy of @0.F (all functions and variables
# are duplicated).
@1.F = copy(@0.F)
```

It is often required to keep the width or shape of peaks constant for all peaks in the dataset. To change the variables bound to parameters with a given name for all functions in F, use the command:

```
F.param = variable
```

Examples:

```
# Set hwhm of all functions in F that have a parameter hwhm to $foo
# (hwhm here means half-width-at-half-maximum)
F.hwhm = $foo

# Bound the variable used for the shape of peak %_1 to shapes of all
# functions in F
F.shape = %_1.shape

# Create a new simple-variable for each function in F and bound the
# variable to parameter hwhm. All hwhm parameters will be independent.
F.hwhm = ~0.2
```

### 3.2.7 Guessing peak location

It is possible to guess peak location and add it to F with the command:

```
[%name =] guess PeakType [[x1:x2]] [initial values...] [in @n]
```

e.g.

```
%f1 = guess Gaussian [22.1:30.5] in @0
```

```
# the same, but assign function's name automatically
guess Gaussian [22.1:30.5] in @0
```

```
# the same, but search for the peak in the whole dataset
guess Gaussian in @0
```

```
# the same, but works only if there is exactly one dataset loaded
guess Gaussian
```

```
guess Linear in @* # adds a function to every dataset
```

```
# guess width and height, but set center and shape explicitly
guess PseudoVoigt [22.1:30.5] center=$ctr, shape=~0.3 in @0
```

- If the range is omitted, the whole dataset will be searched.
- Name of the function is optional.
- Some of the parameters can be specified with syntax *parameter=variable*.
- As an exception, if the range is omitted and the parameter *center* is given, the peak is searched around the *center*, +/- value of the option *guess-at-center-pm*.

Fityk offers only a primitive algorithm for peak-detection. It looks for the highest point in a given range, and then tries to find the width of the peak.

If the highest point is found near the boundary of the given range, it is very probable that it is not the peak top, and, if the option *can-cancel-guess* is set to true, the guess is cancelled.

There are two real-number options related to *guess*: *height-correction* and *width-correction*. The default value for them is 1. The guessed height and width are multiplied by the values of these options respectively.

Linear function is guessed using linear regression. It is actually fitted (but weights of points are not used), not guessed.

### 3.2.8 Displaying information

If you are using the GUI, most of the available information can be displayed with mouse clicks. Alternatively, you can use the *info* command. Using *info+* instead of *info* sometimes gives more verbose output.

Below is the list of arguments of *info* related to this chapter. The full list is in *info: show information*

**info guess [range]** Shows where the *guess* command would find a peak.

**info functions** Lists all defined functions.

**info variables** Lists all defined variables.

**info @n.F** Shows information about F in dataset *n*.

**info @n.Z** Shows information about Z in dataset *n*.

**info formula in @n** Shows the mathematical formula of the fitted model. Some primitive simplifications are applied to the formula. To prevent it, put plus sign (+) after *info*.

**info @n.dF(x)** Compares the symbolic and numerical derivatives in *x* (useful for debugging).

**info peaks in @n** Show parameters of functions from dataset *n*. With the plus sign (+) after *info*, symmetric errors of the parameters are also included.

The model can be exported to file as data points, using the syntax described in *Exporting data*, or as mathematical formula, using the `info` command redirected to a file:

```
info[+] formula in @n > filename
```

The style of the formula output, governed by the `formula-export-style` option, can be either `normal` (`exp(-x^2)`) or `gnuplot` (`exp(-x**2)`).

The list of parameters of functions can be exported using the command:

```
info[+] peaks in @n > filename
```

With `@*` formulae or parameters used in all datasets are written.

## 3.3 Fitting

### 3.3.1 Nonlinear optimization

This is the core. We have a set of observations (data points), to which we want to fit a *model* that depends on adjustable parameters. Let me quote *Numerical Recipes*, chapter 15.0, page 656 (if you do not know the book, visit <http://www.nr.com>):

The basic approach in all cases is usually the same: You choose or design a figure-of-merit function (merit function, for short) that measures the agreement between the data and the model with a particular choice of parameters. The merit function is conventionally arranged so that small values represent close agreement. The parameters of the model are then adjusted to achieve a minimum in the merit function, yielding best-fit parameters. The adjustment process is thus a problem in minimization in many dimensions. [...] however, there exist special, more efficient, methods that are specific to modeling, and we will discuss these in this chapter. There are important issues that go beyond the mere finding of best-fit parameters. Data are generally not exact. They are subject to measurement errors (called noise in the context of signal-processing). Thus, typical data never exactly fit the model that is being used, even when that model is correct. We need the means to assess whether or not the model is appropriate, that is, we need to test the goodness-of-fit against some useful statistical standard. We usually also need to know the accuracy with which parameters are determined by the data set. In other words, we need to know the likely errors of the best-fit parameters. Finally, it is not uncommon in fitting data to discover that the merit function is not unimodal, with a single minimum. In some cases, we may be interested in global rather than local questions. Not, “how good is this fit?” but rather, “how sure am I that there is not a very much better fit in some corner of parameter space?”

Our function of merit is WSSR - the weighted sum of squared residuals, also called chi-square:

$$\chi^2(\mathbf{a}) = \sum_{i=1}^N \left[ \frac{y_i - y(x_i; \mathbf{a})}{\sigma_i} \right]^2 = \sum_{i=1}^N w_i [y_i - y(x_i; \mathbf{a})]^2$$

Weights are based on standard deviations,  $w_i = 1/\sigma_i^2$ . You can learn why squares of residuals are minimized e.g. from chapter 15.1 of *Numerical Recipes*.

So we are looking for a global minimum of  $\chi^2$ . This field of numerical research (looking for a minimum or maximum) is usually called optimization; it is non-linear and global optimization. Fityk implements three very different optimization methods. All are well-known and described in many standard textbooks.

The standard deviations of the best-fit parameters are given by the square root of the corresponding diagonal elements of the covariance matrix. The covariance matrix is based on standard deviations of data points. Formulae can be found e.g. in *GSL Manual*, chapter *Linear regression. Overview* (weighted data version).

**Note:** Some programs scale standard deviations of the parameters with the standard deviation of the fit  $\sigma_f = \sqrt{\chi^2/n_{DoF}}$ , where  $n_{DoF}$  is the number of degrees of freedom, i.e. the number of active data points minus the number of independent parameters.

Fityk is **not** doing this.

### 3.3.2 Fitting related commands

To fit model to data, use command

```
fit[+] [number-of-iterations] [in @n ...]
```

The plus sign (+) prevents initialization of the fitting method. It is used to continue the previous fitting where it left off.

All non-linear fitting methods are iterative. *number-of-iterations* is the maximum number of iterations. There are also other stopping criteria, so the number of executed iterations can be smaller.

```
fit in @* fits all datasets simultaneously.
```

Fitting methods can be set using the set command:

```
set fitting-method = method
```

where method is one of: Levenberg-Marquardt, Nelder-Mead-simplex, Genetic-Algorithms.

All non-linear fitting methods are iterative, and there are two common stopping criteria:

- the number of iterations and it can be specified after the `fit` command.
- and the number of evaluations of the objective function (WSSR), specified by the value of option `max-wssr-evaluations` (0=unlimited). It is approximately proportional to time of computations, because most of time in fitting process is taken by evaluating WSSR.

There are also other criteria, different for each method.

If you give too small *number-of-iterations* to the command `fit`, and fit is not converged, it makes sense to use command `fit+` to process further iterations.

Setting `set autoplot = on-fit-iteration` will plot a model after every iteration, to visualize progress. (see *autoplot*)

```
info fit shows goodness-of-fit.
```

To see symmetric errors use `info errors`. `info+ errors` additionally shows the variance-covariance matrix. Individual symmetric errors of simple-variables can be accessed as `$variable.error` or e.g. `%func.height.error`

Available methods can be mixed together, e.g. it is sensible to obtain initial parameter estimates using the Simplex method, and then fit it using Levenberg-Marquardt.

Values of all parameters are stored before and after fitting (if they change). This enables simple undo/redo functionality. If in the meantime some functions or variables were added or removed, the program can still load the old parameters, but the result can be unexpected. The following history-related commands are provided:

**fit undo** move back to the previous parameters (undo fitting).

**fit redo** move forward in the parameter history

**info fit-history** show number of items in the history

**fit history n** load the *n*-th set of parameters from history

**fit history clear** clear the history

### 3.3.3 Levenberg-Marquardt

This is a standard nonlinear least-squares routine, and involves computing the first derivatives of functions. For a description of the L-M method see *Numerical Recipes*, chapter 15.5 or Siegmund Brandt, *Data Analysis*, chapter 10.15. Essentially, it combines an inverse-Hessian method with a steepest descent method by introducing a  $\lambda$

factor. When  $\lambda$  is equal to 0, the method is equivalent to the inverse-Hessian method. When  $\lambda$  increases, the shift vector is rotated toward the direction of steepest descent and the length of the shift vector decreases. (The shift vector is a vector that is added to the parameter vector.) If a better fit is found on iteration,  $\lambda$  is decreased – it is divided by the value of `lm-lambda-down-factor` option (default: 10). Otherwise,  $\lambda$  is multiplied by the value of `lm-lambda-up-factor` (default: 10). The initial  $\lambda$  value is equal to `lm-lambda-start` (default: 0.0001).

The Marquardt method has two stopping criteria other than the common criteria.

- If it happens twice in sequence, that the relative change of the value of the objective function (WSSR) is smaller than the value of the `lm-stop-rel-change` option, the fit is considered to have converged and is stopped.
- If  $\lambda$  is greater than the value of the `lm-max-lambda` option (default:  $10^{15}$ ), usually when due to limited numerical precision WSSR is no longer changing, the fitting is also stopped.

### 3.3.4 Nelder-Mead downhill simplex method

To quote chapter 4.8.3, p. 86 of Peter Gans, *Data Fitting in the Chemical Sciences by the Method of Least Squares*:

A simplex is a geometrical entity that has  $n+1$  vertices corresponding to variations in  $n$  parameters. For two parameters the simplex is a triangle, for three parameters the simplex is a tetrahedron and so forth. The value of the objective function is calculated at each of the vertices. An iteration consists of the following process. Locate the vertex with the highest value of the objective function and replace this vertex by one lying on the line between it and the centroid of the other vertices. Four possible replacements can be considered, which I call contraction, short reflection, reflection and expansion.[...] It starts with an arbitrary simplex. Neither the shape nor position of this are critically important, except insofar as it may determine which one of a set of multiple minima will be reached. The simplex then expands and contracts as required in order to locate a valley if one exists. Then the size and shape of the simplex is adjusted so that progress may be made towards the minimum. Note particularly that if a pair of parameters are highly correlated, *both* will be simultaneously adjusted in about the correct proportion, as the shape of the simplex is adapted to the local contours.[...] Unfortunately it does not provide estimates of the parameter errors, etc. It is therefore to be recommended as a method for obtaining initial parameter estimates that can be used in the standard least squares method.

This method is also described in previously mentioned *Numerical Recipes* (chapter 10.4) and *Data Analysis* (chapter 10.8).

There are a few options for tuning this method. One of these is a stopping criterium `nm-convergence`. If the value of the expression  $2(M-m)/(M^*+*m)$ , where  $M$  and  $m$  are the values of the worst and best vertices respectively (values of objective functions of vertices, to be precise!), is smaller then the value of `nm-convergence` option, fitting is stopped. In other words, fitting is stopped if all vertices are almost at the same level.

The remaining options are related to initialization of the simplex. Before starting iterations, we have to choose a set of points in space of the parameters, called vertices. Unless the option `nm-move-all` is set, one of these points will be the current point – values that parameters have at this moment. All but this one are drawn as follows: each parameter of each vertex is drawn separately. It is drawn from a distribution that has its center in the center of the *domain* of the parameter, and a width proportional to both width of the domain and value of the `nm-move-factor` parameter. Distribution shape can be set using the option `nm-distribution` as one of: `uniform`, `gaussian`, `lorentzian` and `bound`. The last one causes the value of the parameter to be either the greatest or smallest value in the domain of the parameter – one of the two bounds of the domain (assuming that `nm-move-factor` is equal 1).

### 3.3.5 Genetic Algorithms

[TODO]

## 3.4 Settings

**Note:** This chapter is not about GUI settings (things like colors, fonts, etc.), but about settings that are common for both CLI and GUI version.

Command `info set` shows the syntax of the `set` command and lists all possible options.

`set option` shows the current value of the *option*.

`option = value` changes the *option*.

It is possible to change the value of the option temporarily using syntax:

```
with option1=value1 [,option2=value2] command args...
```

The examples at the end of this chapter should clarify this.

**autoplot** See *autoplot*.

**can-cancel-guess** See *Guessing peak location*.

**cut-function-level** See *Speed of computations*.

**data-default-sigma** See *Standard deviation (or weight)*.

**epsilon** It is used for floating-point comparison:  $a$  and  $b$  are considered equal when  $|a-b| < \epsilon$ . You may want to decrease it when you work with very small values, like  $10^{-10}$ .

**exit-on-warning** If the option `exit-on-warning` is set, any warning will close the program. This ensures that no warnings can be overlooked.

**fitting-method** See *Fitting related commands*.

**formula-export-style** See *details in the section "Model"*.

**guess-at-center-pm** See *Guessing peak location*.

**height-correction** See *Guessing peak location*.

**lm-\*** Setting to tune *Levenberg-Marquardt* fitting method.

**max-wssr-evaluations** See *Fitting related commands*.

**nm-\*** Setting to tune *Nelder-Mead downhill simplex* fitting method.

**pseudo-random-seed** Some fitting methods and functions, such as `randnormal` in data expressions use a pseudo-random number generator. In some situations one may want to have repeatable and predictable results of the fitting, e.g. to make a presentation. Seed for a new sequence of pseudo-random numbers can be set using the option `pseudo-random-seed`. If it is set to 0, the seed is based on the current time and a sequence of pseudo-random numbers is different each time.

**refresh-period** During time-consuming computations (like fitting) user interface can remain not changed for this time (in seconds). This option was introduced, because on one hand frequent refreshing of the program's window notably slows down fitting, and on the other hand irresponsive program is a frustrating experience.

**variable-domain-percent** See *the section about variables*.

**verbosity** Possible values: quiet, normal, verbose, debug.

**width-correction** See *Guessing peak location*.

Examples:

```
set fitting-method # show info
set fitting-method = Nelder-Mead-simplex # change default method
set verbosity = verbose
with fitting-method = Levenberg-Marquardt fit 10
with fitting-method=Levenberg-Marquardt, verbosity=only-warnings fit 10
```

## 3.5 Other commands

### 3.5.1 plot: viewing data

In the GUI version there is hardly ever a need to use this command directly.

The command `plot` controls visualization of data and the model. It is used to plot a given area - in GUI it is plotted in the program's main window, in CLI the popular program `gnuplot` is used, if available.

```
plot xrange yrange in @n
```

*xrange* and *yrange* have one of two following syntaxes:

- [min:max]
- .

The second is just a dot (.), and it implies that the appropriate range is not to be changed.

Examples:

```
plot [20.4:50] [10:20] # show x from 20.4 to 50 and y from 10 to 20
plot [20.4:] # x from 20.4 to the end,
# y range will be adjusted to encompass all data
plot . [:10] # x range will not be changed, y from the lowest point to 10
plot [:] [:] # all data will be shown
plot # all data will be shown
plot . . # nothing changes
```

The value of the option *autoplot* changes the automatic plotting behaviour. By default, the plot is refreshed automatically after changing the data or the model. It is also possible to visualize each iteration of the fitting method by replotting the peaks after every iteration.

### 3.5.2 info: show information

First, there is an option *verbosity* (not related to command **info**) which sets the amount of messages displayed when executing commands.

If you are using the GUI, most information can be displayed with mouse clicks. Alternatively, you can use the `info` command. Using the `info+` instead of `info` sometimes displays more detailed information.

The output of **info** can be redirected to a file using syntax:

```
info args > filename # this truncates the file
info args >> filename # this appends to the file
```

The following `info` arguments are recognized:

- variables
- *\$variable\_name*
- types
- *TypeName*
- functions
- *%function\_name*
- datasets
- data [in @n]

- title [in @n]
- filename [in @n]
- commands
- commands [n:m]
- view
- set
- fit [in @n]
- fit-history
- errors [in @n]
- formula [in @n]
- peaks [in @n]
- guess [x-range] [in @n]
- *data-expression* [in @n]
- [ @n.]F
- [ @n.]Z
- [ @n.]dF(*data-expression*)
- *der mathematic-function*
- version

info der shows derivatives of given function:

```
=-> info der sin(a) + 3*exp(b/a)
f(a, b) = sin(a)+3*exp(b/a)
df / d a = cos(a)-3*exp(b/a)*b/a^2
df / d b = 3*exp(b/a)/a
```

### 3.5.3 commands, dump, sleep, reset, quit, !

All commands given during program execution are stored in memory. They can be listed by:

```
info commands [n:m]
```

or written to file:

```
info commands [n:m] > filename
```

To put all commands executed so far during the session into the file `foo.fit`, type:

```
info commands[:] > foo.fit
```

With the plus sign (+) (i.e. `info+ commands [n:m]`) information about the exit status of each command will be added.

To log commands to a file when they are executed, use: Commands can be logged when they are executed:

```
commands > filename      # log commands
commands+ > filename     # log both commands and output
commands > /dev/null     # stop logging
```

Scripts can be executed using the command:

`commands < filename`

You can select lines that are to be executed:

`commands < filename[m:n] # this executes lines from m to n`

It is also possible to execute standard output from an external program:

`commands ! program [args...]`

The command:

`dump > filename`

writes the current state of the program (including all datasets) to a single `.fit` file.

The command `sleep sec` makes the program wait `sec` seconds before continuing.

The command `quit` works as expected. If it is found in a script it quits the program, not only the script.

Commands that start with `!` are passed (without `'!`') to the `system()` call.

# EXTENSIONS

## 4.1 How to add your own built-in function

**Note:** Add built-in function only if *user-defined function (UDF)* is too slow or too limited.

To add a built-in function, you have to change the source of the program and then recompile it. Users who want to do this should be able to compile the program from source and know the basics of C, C++ or another programming language.

The description that follows is not complete. If something is not clear, you can always send me e-mail, etc.

“fp” you can see in fityk source means a real (floating point) number (typedef double fp).

The name of your function should start with uppercase letter and contain only letters and digits. Let us add function Foo with the formula:  $\text{Foo}(\text{height}, \text{center}, \text{hwhm}) = \text{height} / (1 + ((\text{x} - \text{center}) / \text{hwhm})^2)$ . C++ class representing Foo will be named FuncFoo.

In src/func.cpp you will find a list of functions:

```
...  
FACTORY_FUNC(Polynomial6)  
FACTORY_FUNC(Gaussian)  
...
```

Now, add:

```
FACTORY_FUNC(Foo)
```

Then find another list:

```
...  
FuncPolynomial6::formula,  
FuncGaussian::formula,  
...
```

and add the line

```
FuncFoo::formula,
```

Note that in the second list all items but the last are followed by comma.

In the file src/bfunc.h you can now begin writing the definition of your class:

```
class FuncFoo : public Function  
{  
    DECLARE_FUNC_OBLIGATORY_METHODS(Foo)
```

If you want to make some calculations every time parameters of the function are changed, you can do it in method `do_precomputations`. This possibility is provided for calculating expressions, which do not depend on `x`. Write the declaration here:

```
void do_precomputations(std::vector<Variable*> const &variables);
```

and provide a proper definition of this method in `src/bfunc.cpp`.

If you want to optimize the calculation of your function by neglecting its value outside of a given range (see option `cut-function-level` in the program), you will need to use the method:

```
bool get_nonzero_range (fp level, fp &left, fp &right) const;
```

This method takes the level below which the value of the function can be approximated by zero, and should set the left and right variables to proper values of `x`, such that if  $x < \text{left}$  or  $x > \text{right}$  than  $|f(x)| < \text{level}$ . If the function sets left and right, it should return true.

If your function does not have a “center” parameter, and there is a center-like point where you want the peak top to be drawn, write:

```
bool has_center() const { return true; }  
fp center() const { return vv[1]; }
```

In the second line, between return and the semicolon, there is an expression for the `x` coordinate of peak top; `vv[0]` is the first parameter of function, `vv[1]` is the second, etc.

Finally, close the definition of the class with:

```
};
```

Now go to file `src/bfunc.cpp`.

Write the function formula in this way:

```
const char \*FuncFoo::formula  
= "Foo(height, center, hwhm) = height / (1 + ((x-center)/hwhm)^2)";
```

The syntax of the formula is the similar as that of the *UDF*, but for built-in functions only the left hand side of the formula is parsed. The right hand side is for documentation only.

Write how to calculate the value of the function:

```
FUNC_CALCULATE_VALUE_BEGIN(Foo)  
fp xala2 = (x - vv[1]) / vv[2];  
fp inv_denomin = 1. / (1 + xala2 * xala2);  
FUNC_CALCULATE_VALUE_END(vv[0] * inv_denomin)
```

The expression at the end (i.e. `vv[0]*inv_denomin`) is the calculated value. `xala2` and `inv_denomin` are variables introduced to simplify the expression. Note the “fp” (you can also use “double”) at the beginning and semicolon at the end of both lines. The meaning of `vv` has already been explained.

Usually it is more difficult to calculate derivatives:

```
FUNC_CALCULATE_VALUE_DERIV_BEGIN(Foo)  
fp xala2 = (x - vv[1]) / vv[2];  
fp inv_denomin = 1. / (1 + xala2 * xala2);  
dy_dv[0] = inv_denomin;  
fp dcenter = 2 * vv[0] * xala2 / vv[2] * inv_denomin * inv_denomin;  
dy_dv[1] = dcenter;  
dy_dv[2] = dcenter * xala2;  
dy_dx = -dcenter;  
FUNC_CALCULATE_VALUE_DERIV_END(vv[0] * inv_denomin)
```

You must set derivatives `dy_dv[n]` for  $n=0,1,\dots$  (number of parameters of your function - 1) and `dy_dx`. In the last brackets there is a value of the function again.

If you declared `do_precomputations` or `get_nonzero_range` methods, do not forget to write definitions for them.

After compilation of the program check if the derivatives are calculated correctly using command “`info dF(x)`”, e.g. `i dF(30.1)`. You can also use `numarea`, `findx` and `extremum` (see *Functions and variables in data transformation* for details) to verify center, area, height and FWHM properties.

Hope this helps. Do not hesitate to change this description or ask questions if you have any. Consider sharing your function with other users (using [FitykWiki](#) or mailing list).



# APPENDIX A. LIST OF FUNCTIONS

The list of all functions can be obtained using `itypes`. Some formulae here have long parameter names (like “height”, “center” and “hwhm”) replaced with  $a_i$

**Gaussian:**

$$y = a_0 \exp \left[ -\ln(2) \left( \frac{x - a_1}{a_2} \right)^2 \right]$$

**SplitGaussian:**

$$y(x; a_0, a_1, a_2, a_3) = \begin{cases} \text{Gaussian}(x; a_0, a_1, a_2) & x \leq a_1 \\ \text{Gaussian}(x; a_0, a_1, a_3) & x > a_1 \end{cases}$$

**GaussianA:**

$$y = \sqrt{\frac{\ln(2)}{\pi}} \frac{a_0}{a_2} \exp \left[ -\ln(2) \left( \frac{x - a_1}{a_2} \right)^2 \right]$$

**Lorentzian:**

$$y = \frac{a_0}{1 + \left( \frac{x - a_1}{a_2} \right)^2}$$

**LorentzianA:**

$$y = \frac{a_0}{\pi a_2 \left[ 1 + \left( \frac{x - a_1}{a_2} \right)^2 \right]}$$

**Pearson VII (Pearson7):**

$$y = \frac{a_0}{\pi a_2 \left[ 1 + \left( \frac{x - a_1}{a_2} \right)^2 \right]}$$

**Split-Pearson-VII (SplitPearson7):**

$$y(x; a_0, a_1, a_2, a_3, a_4, a_5) = \begin{cases} \text{Pearson7}(x; a_0, a_1, a_2, a_4) & x \leq a_1 \\ \text{Pearson7}(x; a_0, a_1, a_3, a_5) & x > a_1 \end{cases}$$

**Pearson-VII-Area (Pearson7A):**

$$y = \frac{a_0 \Gamma(a_3) \sqrt{2^{\frac{1}{a_3}} - 1}}{a_2 \Gamma(a_3 - \frac{1}{2}) \sqrt{\pi} \left[ 1 + \left( \frac{x-a_1}{a_2} \right)^2 \left( 2^{\frac{1}{a_3}} - 1 \right) \right]^{a_3}}$$

**Pseudo-Voigt (PseudoVoigt):**

$$y = a_0 \left[ (1 - a_3) \exp \left( -\ln(2) \left( \frac{x - a_1}{a_2} \right)^2 \right) + \frac{a_3}{1 + \left( \frac{x-a_1}{a_2} \right)^2} \right]$$

Pseudo-Voigt is a name given to the sum of Gaussian and Lorentzian.  $a_3$  parameters in Pearson VII and Pseudo-Voigt are not related.

**Pseudo-Voigt-Area (PseudoVoigtA):**

$$y = a_0 \left[ \frac{(1 - a_3) \sqrt{\ln(2)}}{a_2 \sqrt{\pi}} \exp \left( -\ln 2 \left( \frac{x - a_1}{a_2} \right)^2 \right) + \frac{a_3}{\pi a_2 \left[ 1 + \left( \frac{x-a_1}{a_2} \right)^2 \right]} \right]$$

**Voigt:**

$$y = \frac{a_0 \int_{-\infty}^{+\infty} \frac{\exp(-t^2)}{a_3^2 + \left( \frac{x-a_1}{a_2} - t \right)^2} dt}{\int_{-\infty}^{+\infty} \frac{\exp(-t^2)}{a_3^2 + t^2} dt}$$

The Voigt function is a convolution of Gaussian and Lorentzian functions.  $a_0$  = heigth,  $a_1$  = center,  $a_2$  is proportional to the Gaussian width, and  $a_3$  is proportional to the ratio of Lorentzian and Gaussian widths.

Voigt is computed according to R.J.Wells, *Rapid approximation to the Voigt/Faddeeva function and its derivatives*, Journal of Quantitative Spectroscopy & Radiative Transfer 62 (1999) 29-48. (See also: <http://www.atm.ox.ac.uk/user/wells/voigt.html>). The approximation is very fast, but not very exact.

**VoigtA:**

$$y = \frac{a_0}{\sqrt{\pi} a_2} \int_{-\infty}^{+\infty} \frac{\exp(-t^2)}{a_3^2 + \left( \frac{x-a_1}{a_2} - t \right)^2} dt$$

**Exponentially Modified Gaussian (EMG):**

$$y = \frac{ac\sqrt{2\pi}}{2d} \exp \left( \frac{b-x}{d} + \frac{c^2}{2d^2} \right) \left[ \frac{d}{|d|} - \operatorname{erf} \left( \frac{b-x}{\sqrt{2}c} + \frac{c}{\sqrt{2}d} \right) \right]$$

**LogNormal:**

$$y = h \exp \left\{ -\ln(2) \left[ \frac{\ln \left( 1 + 2b \frac{x-c}{w} \right)}{b} \right]^2 \right\}$$

**Doniach-Sunjic (DoniachSunjic):**

$$y = \frac{h \left[ \frac{\pi a}{2} + (1 - a) \arctan \left( \frac{x - E}{F} \right) \right]}{F + (x - E)^2}$$

**Polynomial5:**

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$$



# APPENDIX B. COMMAND SHORTENINGS

The pipe symbol (|) shows the minimum length of the command. “define” means that command “define” can be shortened as “def”, “defi” or “defin”. Commands not listed here cannot be shortened. Arguments of “info” command also cannot be shortened (i.e. you must write “i fit”, not “i f”).

- clommands
- define
- flit
- guess
- info
- plot
- slet
- undefine
- wlith



## APPENDIX C. LITERATURE

The following books were helpful when writing the program (from scientific, not programming side).

William Press, Saul Teukolsky, William Vetterling, Brian Flannery. *Numerical Recipes in C*. <http://www.nr.com>

Peter Gans. *Data Fitting in the Chemical Sciences by the Method of Least Squares*. John Wiley & Sons. 1992.

Siegmund Brandt. *Data Analysis*. Springer Verlag. 1999.

*PeakFit 4.0 for Windows User's Manual*. AISN Software. 1997.



## APPENDIX D. LICENSE

Fityk is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Text of the license is distributed with the program in the file `COPYING`.



# APPENDIX E. ABOUT THIS MANUAL

This manual is written ReStructuredText. All changes, improvements, corrections, etc. are welcome. Use the [Show Source](#) link to get the source of the page, save it, edit, and send me either modified version or patch containing changes.

Following people have contributed to this manual (in chronological order): Marcin Wojdyr (maintainer), Stan Gierlotka, Jaap Folmer, Michael Richardson.