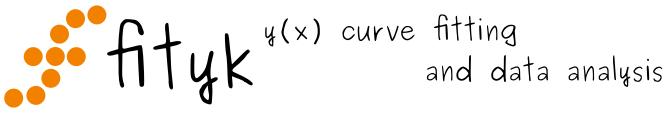


Fityk manual

Release 1.3.1

Oct 03, 2017

Contents

CHAPTER 1

Introduction

Fityk is a program for nonlinear fitting of analytical functions (especially peak-shaped) to data (usually experimental data).

To put it differently, it is primarily peak fitting software, but can handle other types of functions as well.

Apart from the actual fitting, the program helps with data processing and provides ergonomic graphical interface (and also command line interface and scripting API – but if the program is popular in some fields, it's thanks to its graphical interface).

It is reportedly[1] used in crystallography, chromatography, photoluminescence and photoelectron spectroscopy, infrared and Raman spectroscopy, to name but a few.

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.

In simple cases, the program can be operated with mouse only. Let say that you want to model the data with multiple peaks or other function shapes. You select a built-in function type (such as Gaussian, Voigt, sigmoid, polynomial and dozens of others) place it with the mouse, place other functions and click a button to fit it.

But the program can also handle quite complex scenarios. You can define your own function types. You can specify sophisticated dependencies between parameters of the functions (say, peak widths given as a function of peak positions). You can fit multiple datasets together using common set of parameters. You can model zero-shift in your instrument or do more complicated refinement of the X scale. And you can automate all this work. If you don't know how to handle your case, do not hesistate to ask on the users group[2] or contact the author.

To download the latest version of the program or to contact the author visit fityk.nieto.pl[3].

Reference for academic papers: M. Wojdyr, J. Appl. Cryst. 43, 1126-1128[4] (2010) [reprint[5]]

[1] https://scholar.google.com/citations?view_op=view_citation&citation_for_view=aCtDUBMAAAAJ:u5HHmVD_uO8C
[2] http://groups.google.com/group/fityk-users/
[3] http://fityk.nieto.pl/
[4] http://dx.doi.org/10.1107/S0021889810030499
[5] http://wojdyr.github.io/fityk-JAC-10-reprint.pdf

Open Source

Fityk is open-source (GPL2+[6]). If you are interested, please find the source code at GitHub[7].

It uses a few open source projects:

- NLopt[8] for several optional fitting methods
- one of the fitting methods uses MPFIT[9] library (MINPACK-1 Least Squares Fitting Library in C), which includes software developed by the University of Chicago, as Operator of Argonne National Laboratory.
- xylib[10] library handles reading data files
- Lua[11] interpreter is embedded for scripting
- and a few popular libraries and tools that make programming much easier: wxWidgets (GUI), Boost (misc), zlib (compression), readline (CLI), SWIG (bindings), Catch (testing).

About this manual

This manual is written in ReStructuredText. All corrections and improvements are welcome. Use the `Show Source` link to get the source of the page, edit it and send me either the modified version or a patch.

Alternatively, go to GitHub[12], open corresponding rst file, press *Fork and edit this file* button, do edits in your web browser and click *Propose file change*.

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

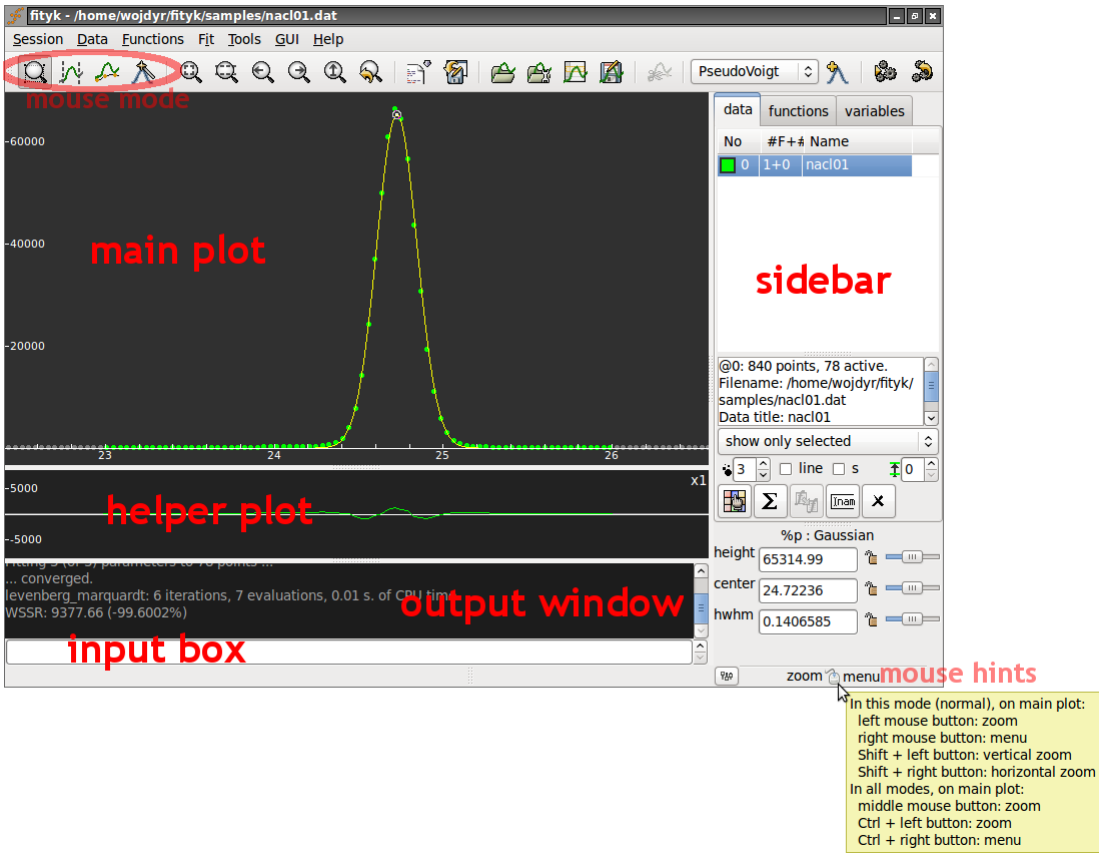
[6] http://creativecommons.org/licenses/GPL/2.0/
[7] https://github.com/wojdyr/fityk/
[8] http://ab-initio.mit.edu/wiki/index.php/NLopt
[9] http://www.physics.wisc.edu/~craigm/idl/cmpfit.html
[10] http://xylib.sourceforge.net/
[11] http://www.lua.org/
[12] https://github.com/wojdyr/fityk/tree/master/doc

CHAPTER 2

Getting Started

Graphical Interface

That's how the GUI (Graphical User Interface) looks like:



The **main plot** can display data points, model that is to be fitted to the data and individual functions in the model. Use can configure what is displayed and how (through *GUI* → *Configure* or context menu).

The **helper plot** shows how well the model fits the data. You may have one, two or no helper plots (*GUI* → *Show*). By default, the plot shows the difference between the data and the model. It can also show weighted or cumulative

difference, and a couple of other things.

The helper plot is also handy for zooming – with left and middle mouse buttons. Selecting a horizontal span with the left button zooms into this span. The middle button goes back to the whole dataset (the same as  in the toolbar).

The **sidebar** is for switching between datasets, inspecting functions, and for changing function parameters. It also provides quick access to a few properties of the main plot, such as the size of data points.

On the main plot, the meaning of the left and right mouse button depends on the current **mouse mode**. Mouse modes are switched using toolbar buttons:

-  *normal mode* – the left button zooms in and the right button shows pop-up menu,
-  *data-range mode* – for activating and de-activating data, i.e. for selecting regions of interest,
-  *baseline mode* – manual baseline subtraction (in may never need it),
-  *add-peak mode* – for placing peaks and other functions.

The status bar shows a hint what the mouse does in the current mode.

Finally, the **input field** and the **output window** provide alternative, console-like way of interacting with the program. Also, the GUI operations that change the state of the program (data, model, non-visual settings) are translated into textual commands and printed in the output window.

> **Note:** To save configuration of the GUI (visible windows, colors, etc.) for next session use *GUI → Save current config*.

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 irrelevent for the time being.

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

Textual commands that correspond to performed operations are shown in this section in CLI boxes.

First load data from the `nacl01.dat` file. Select *Data → Load File* from the menu (or  from the toolbar) and choose the file.

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

You can zoom-in to the biggest peak using the left mouse button on the residual (helper) plot. To zoom out, press  on the toolbar.

Now all data points are active. Only the biggest peak is of our interest, so we want to deactivate the remaining points. Change to the *range mode* (toolbar: ) and deactivate not needed points with the right mouse button.

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

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 `i Gaussian`) or look for it in the section *Built-In Functions*.

Select *Gaussian* from the list of functions on the toolbar and press  .

```
guess Gaussian
```

Automatic peak detection works in this case, but if it wouldn't, you may set the initial peak position, height and width manually. Either with mouse in the *add-peak mode*, or with a command.

```
F += Gaussian(~60000, ~24.6, ~0.2)
```

Parameters of an existing function can be changed in the sideber, or by dragging that little square handle attached to each function (you should see a handle at the top of your Gaussian).

If the peaks/functions are not named explicitely (like in this example), they get automatic names `%_1`, `%_2`, etc.

Now let us fit the function. Select *Fit* → *Run* from the menu or press  .

```
fit
```

Important: Fitting minimizes the **weighted** sum of squared residuals (see *Nonlinear Optimization*). The default *weights of points* are not equal.

Now you can check the peak position together with other parameters on the sidebar. Alternatively, right click the peak handle and select *Show Info* from the context menu.

```
info prop %_1
```

That's it!

By the way, you can save all the issued commands to a file (*Session* → *Save History*)

```
info history > myscript.fit
```

and later use it as a macro (*Session* → *Execute script*).

```
exec myscript.fit
```

Command Line

Fityk comes with a small domain-specific language (DSL). All operations in Fityk are driven by commands of this language. Commands can be typed in the input box in the GUI, but if all you want to do is to type commands, the program has a separate CLI version (cfityk) for this.

Do not worry
you do not need to learn these commands. It is possible to use menus and dialogs in the GUI and completely avoid typing commands.

When you use the GUI and perform an action using the menu, you can see the corresponding command in the output window. Fityk has less than 30 commands. Each performs a single actions, such as loading data from file, adding function, assigning variable, fitting, or writing results to a file.

A sequence of commands written down in a file makes a script (macro), which can automate common tasks. Complex tasks may need to be programmed in a general-purpose language. That is why Fityk has embedded Lua interpreter (Lua is a lightweight programming language). It is also possible to use Fityk library from a program in Python, C, C++, Java, Ruby or Perl, and possibly from other languages supported by SWIG.

Now a quick glimpse at the syntax. The =-> prompt below marks an input:

```
=-> print pi
3.14159
=-> # this is a comment -- from `#' to the end of line
=-> p '2+3=', 2+3  # p stands for print
2+3 = 5
=-> set numeric_format='%.9f'  # show 9 digits after dot
=-> pr pi, pi^2, pi^3  # pr, pri and prin also stand for print
3.141592654 9.869604401 31.006276680
```

Usually, one line has one command, but if it is really needed, two or more commands can be put in one line:

```
=-> $a = 3; $b = 5  # two commands separated with `;'
```

or a backslash can be used to continue a command in the next line:

```
=-> print \
... 'this'
this
```

If the user works simultaneously with multiple datasets, she can refer to a dataset using its number: the first dataset is `@0`, the second – `@1`, etc:

```
=-> fit # perform fitting of the default dataset (the first one)
=-> @2: fit # fit the third dataset (@2)
=-> @2 @3: fit # fit the third dataset (@2) and then the fourth one (@3)
=-> @*: fit # fit all datasets, one by one
```

Settings in the program are changed with the command `set`:

```
set key = value
```

For example:

```
=-> set logfile = 'C:\log.fit' # log all commands to this file
=-> set verbosity = 1 # make output from the program more verbose
=-> set epsilon = 1e-14
```

The last example changes the ϵ value, which is used to test floating-point numbers a and b for equality (it is well known that due to rounding errors the equality test for two numbers should have some tolerance, and the tolerance should be tailored to the application): $|a-b| < \epsilon$.

To run a single command with different settings, add `with key=value` before the command:

```
=-> print pi == 3.14  # default epsilon = 10^-12
0
=-> with epsilon = 0.1 print pi == 3.14  # abusing epsilon
1
```

Putting it all together, a line typically has a single command, often prefixed with datasets+`:`, sometimes prefixed with `with`. In general it is:

```
[[@...:] [with ...] command [";" command]...] [#comment]
```

All the commands are described in next chapters.

CHAPTER 3

Data

Loading Data

Data files are read using the xylib library[13].

In the GUI
click  . If it just works for your files, you may go straight to *Active and Inactive Points*.

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 and CSV files) 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* - selects one or more blocks of data from a multi-block file such as VAMAS
- *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).

A few examples should clarify it:

```
@0 < foo.vms
@0 < 'foo.vms'  # filename can be quoted
@0 < foo.fii text first_line_header  # with filetype options
@0 < foo.csv:1:4::  # x,y - 1st and 4th columns
@0 < foo.csv:1:2:3:  # read std. dev. of y from 3rd column
@0 < foo.csv:0:1::  # x - index (0,1,2,...), y - first column
@0 < foo.raw::::0,1  # load two first blocks of data (as one dataset)
```

[13] http://xylib.sourceforge.net/

You may also specify multiple *y* columns. It will load each *x*/*y* pair as a separate dataset. In this case you need to use `@+ < ...` (`@+` denotes new dataslot):

```
@+ < foo.csv:1:3,4:: # load two dataset (with y in columns 3,4)
@+ < foo.csv:1:3..5:: # load three dataset (with y in columns 3,4,5)
@+ < foo.csv:1:4..6,2:: # load four dataset (y: 4,5,6,2)
@+ < foo.csv:1:2..:: # load 2nd and all the next columns as y
```

Information about loaded data can be obtained with:

```
info data
```

Supported Filetypes

text ASCII text, multicolumn numeric data. The details are given in the next section.

csv CSV or TSV file. Similar to text but supports quoted (`"`) values and uses different heuristic to interpret ambiguous cases.

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

canberra_cnf Canberra CNF format

cpi Sietronics Sieray CPI format

uxd Siemens/Bruker UXD format (powder diffraction data)

bruker_raw Simens-Bruker RAW format (version 1,2,3)

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

And a few others. The full list is available at: http://xylib.sourceforge.net/.

Reading Text Files

The *xylib* library can read TSV or CSV formats (tab or comma separated values). In fact, the values can be separated by any whitespace character or by one of ,;: punctations, or by any combination of these.

Empty lines and comments that start with hash (#) are skipped.

Since there is a lot of files in the world that contain numeric data mixed with text, unless the `strict` option is given any text that can not be interpreted as a number is regarded a start of comment (the rest of the line is ignored).

Note that the file is parsed regardless of blocks and columns specified by the user. The data read from the file are first stored in a table with *m* columns and *n* rows. If some of the lines have 3 numbers in it, and some have 5 numbers, we can either discard the lines that have 3 numbers or we can discard the numbers in 4th and 5th column. Usually the latter is done, but there are exceptions. The shorter lines are ignored

- if it is the last line in the file (probably the program was terminated while writing the file),
- if it contains only one number, but the prior lines had more numbers (this may be a comment that starts with a number)
- if all the (not ignored) prior lines and the next line are longer

These rule were introduced to read free-format log files with textual comments inserted between lines with numeric data.

For now, xylib does not handle well nan's and inf's in the data.

Data blocks and columns may have names. These names are used to set a title of the dataset (see *Working with Multiple Datasets* for details). If the option `first_line_header` is given and the number of words in the first line is equal to the number of data columns, each word is used as a name of corresponding column. If the number of words is different, the first line is used as a name of the block. If the `last_line_header` option is given, the line preceding the first data line is used to set either column names or the block name.

If the file starts with the "`LAMMPS (`" string, the `last_line_header` option is set automatically. This is very helpful when plotting data from LAMMPS log files.

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. (Since active points do not need to be in one region, we do not use the *region of interest* term here, but such region can be easy selected). A data *transformation*:

```
A = boolean-condition
```

can be used to change the state of points.

In the GUI
data points can be activated and disactivated with mouse in the data-range mode (toolbar: ).

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's standard deviation, σ, 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 $\max(y^{1/2}, 1)$ or to 1, depending on the `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 the standard deviation, e.g. make it equal for every point with the command: `S=1`. See *Data Point 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.

Data Point Transformations

Every data point has four properties: x coordinate, y coordinate, standard deviation of y and active/inactive flag. These properties can be changed using symbols `X`, `Y`, `S` and `A`, respectively. It is possible to either change a single point or apply a transformation to all points. For example:

- `Y[3]=1.2` assigns the y coordinate of the 4th point (0 is first),
- `Y = -y` changes the sign of the y coordinate for all points.

On the left side of the equality sign you can have one of symbols `X`, `Y`, `S`, `A`, possibly with the index in brackets. The symbols on the left side are case insensitive.

The right hand side is a mathematical expression that can have special variables:

- lower case letters `x`, `y`, `s`, `a` represent properties of data points before transformation,
- upper case `X`, `Y`, `S`, `A` stand for the same properties after transformation,
- `M` stands for the number of points.
- `n` stands for the index of currently transformed point, e.g., `Y=y[M-n-1]` means that *n*-th point (*n*=0, 1, ... M-1) is assigned *y* value of the *n*-th point from the end.

Before the transformation a new array of points is created as a copy of the old array. Operations are applied sequentially from the first point to the last one, so while `Y[n+1]` and `y[n+1]` have always the same value, `Y[n-1]` and `y[n-1]` may differ. For example, the two commands:

```
Y = y[n] + y[n-1]
Y = y[n] + Y[n-1]
```

differ. The first one adds to each point the value of the previous point. The second one adds the value of the previous point *after* transformation, so effectively it adds the sum of all previous points. The index `[n]` could be omitted (`Y = y + y[n-1]`). The value of undefined points, like `y[-1]` and `Y[-1]`, is explained later in this section.

Expressions can contain:

- real numbers in normal or scientific format (e.g. `1.23e5`),
- constants `pi`, `true` (1), `false` (0)
- binary operators: `+`, `-`, `*`, `/`, `^`,
- boolean operators: `and`, `or`, `not`,
- comparisions: `>`, `>=`, `<`, `<=`, `==`, `!=`.
- one argument functions:
  - `sqrt`
  - `exp`
  - `log10`
  - `ln`
  - `sin`
  - `cos`
  - `tan`
  - `sinh`
  - `cosh`
  - `tanh`
  - `atan`
  - `asin`
  - `acos`
  - `erf`
  - `erfc`
  - `gamma`
  - `lgamma` (=ln(|`gamma()`|))

  - `abs`
  - `round` (rounds to the nearest integer)
- two argument functions:
  - `mod` (modulo)
  - `min2`
  - `max2` (`max2(3,5)` gives 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 returns *expression1* if condition is true and *expression2* otherwise.

A few examples.

- The x scale of diffraction pattern can be changed from 2θ to Q:

```
X = 4*pi * sin(x/2*pi/180) / 1.54051 # Cu 2θ -> Q
```

- Negative y values can be zeroed:

```
Y = max2(y, 0)
```

- All standard deviations can be set to 1:

```
S = 1
```

- It is possible to select active range of data:

```
A = x > 40 and x < 60 # select range (40, 60)
```

All operations are performed on **real numbers**. Two numbers that differ less than ϵ (the value of ϵ is set by the *option epsilon*) are considered equal.

Points can be created or deleted by changing the value of `M`. For example, the following commands:

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

create 500 points and generate a sinusoid.

Points are kept sorted according to their x coordinate. The sorting is performed after each transformation.

> **Note:** Changing the x coordinate may change the order and indices of points.

Indices, like all other values, are computed in the real number domain. If the index is not integer (it is compared using ϵ to the rounded value):

- `x`, `y`, `s`, `a` are interpolated linearly. For example, `y[2.5]` is equal to `(y[2]+[3])/2`. If the index is less than 0 or larger than M-1, the value for the first or the last point, respectively, is returned.
- For `X`, `Y`, `S`, `A` the index is rounded to integer. If the index is less than 0 or larger than M-1, 0 is returned.

Transformations separated by commas (`,`) form a sequance of transformations. During the sequance, the vectors `x`, `y`, `s` and `a` that contain old values are not changed. This makes possible to swap the axes:

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

The special `index(arg)` function returns the index of point that has *x* equal *arg*, or, if there is no such point, the linear interpolation of two neighbouring indices. This enables equilibrating the step of data (with interpolation of *y* and σ):

```
X = x[0] + n * (x[M-1]-x[0]) / (M-1), Y = y[index(X)], S = s[index(X)]
```

It is possible to delete points for which given condition is true, using expression `delete(condition)`:

```
delete(not a) # delete inactive points

# reduce twice the number of points, averaging x and adding y
x = (x[n]+x[n+1])/2
y = y[n]+y[n+1]
delete(mod(n,2) == 1)
```

If you have more than one dataset, you may need to specify to which dataset the transformation applies. See *Working with Multiple Datasets* for details.

The value of a data expression can be shown using the `print` command. The precision of printed numbers is governed by the *numeric_format* option.

```
print M # the number of points
print y[index(20)] # value of y for x=20
```

Aggregate Functions

Aggregate functions have syntax:

```
aggregate(expression [if condition])
```

and return a single value, calculated from values of all points for which the given condition is true. If the condition is omitted, all points in the dataset are taken into account.

The following aggregate functions are recognized:

- `min()` — the smallest value,
- `max()` — the largest value,
- **`argmin()` — (stands for the argument of the minimum)** the x value of the point for which the expression in brackets has the smallest value,
- **`argmax()` — the x value of the point for which the expression** in brackets has the largest value,
- `sum()` — the sum,
- `count()` — the number of points for which the expression is true,
- `avg()` — the arithmetic mean,
- `stddev()` — the standard deviation,
- `centile(N, )` — percentile
- `darea()` — a function used to normalize the area (see the example below). It returns the sum of *expression**(*x*[*n*+1]-*x*[*n*-1])/2. In particular, `darea(y)` returns the interpolated area under data points.

Examples:

```
p avg(y) # print the average y value
p centile(50, y) # print the median y value
p max(y) # the largest y value
p argmax(y) # the position of data maximum
p max(y if x > 40 and x < 60)    # the largest y value for x in (40, 60)
```

```
p max(y if a) # the largest y value in the active range
p min(x if y > 0.1)] # x of the first point with y > 0.1
p count(y>100) # the number of points that have y above 100
p count(y>avg(y)) # aggregate functions can be nested
p y[min(n if y > 100)] # the first (from the left) value of y above 100

# take the first 2000 points, average them and subtract as background
Y = y - avg(y if n<2000)

Y = y / darea(y) # normalize data area

# make active only the points on the left from the first
# point with y > 0.1
a = x < min(x if y > 0.1)]
```

Functions and Variables in Data Transformation

You may postpone reading this section and read about the *Models* first.

Variables ($foo) and functions (%bar) can be used in data expressions:

```
Y = y / $foo  # divides all y's by $foo
Y = y - %f(x) # subtracts function %f from data
Y = y - @0.F(x) # subtracts all functions in F

# print the abscissa value of the maximum of the model
# (only the values in points are considered,
#  so it's not exactly the model's maximum)
print argmax(F(x))

# print the maximum of the sum of two functions
print max(%_1(x) + %_2(x))

# Fit constant x-correction (i.e. fit the instrumental zero error), ...
Z = Constant(~0)
fit
X = x + Z(x)          # ... correct the data
Z = 0                 # ... and remove the correction from the model.
```

In the GUI

in the *Baseline Mode* (), functions `Spline` and `Polyline` are used to subtract manually selected background. Clicking  results in a command like this:

```
%bg0 = Spline(14.2979,62.1253, 39.5695,35.0676, 148.553,49.9493)
Y = y - %bg0(x)
```

Clicking the same button again undoes the subtraction:

```
Y = y + %bg0(x)
```

The function edited in the *Baseline Mode* is always named `%bgX`, where *X* is the index of the dataset.

Values of the function parameters (e.g. `%fun.a0`) and pseudo-parameters `Center`, `Height`, `FWHM`, `IB` and `Area` (e.g. `%fun.Area`) can also be used. IB stands for Integral Breadth – width of rectangle with the same area and height as the peak, in other words Area/Height. Not all functions have pseudo-parameters.

It is also possible to calculate some properties of %functions:

- `%f.numarea(x1, x2, n)` gives area integrated numerically from *x1* to *x2* using trapezoidal rule with *n* equal steps.

- `%f.findx(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*).
- `%f.extremum(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:

```
print %fun.findx(-10, 10, 0)  # find the zero of %fun in [-10, 10]
print F.findx(-10, 10, 0)     # find the zero of the model in [-10, 10]
print %fun.numarea(0, 100, 10000) # shows area of function %fun
print %_1(%_1.extremum(40, 50)) # shows extremum value

# calculate FWHM numerically, value 50 can be tuned
$c = {%f.Center}
p %f.findx($c, $c+50, %f.Height/2) - %f.findx($c, $c-50, %f.Height/2)
p %f.FWHM # should give almost the same.
```

Working with Multiple Datasets

Let us call a set of data that usually comes from one file – a *dataset*. It is possible to work simultaneously with multiple datasets. Datasets have numbers and are referenced by `@` with the number, (e.g. `@3`). The user can specify which dataset the command should be applied to:

```
@0: M=500     # change the number of points in the first dataset
@1 @2: M=500 # the same command applied to two datasets
@*: M=500     # and the same applied to all datasets
```

If the dataset is not specified, the command applies to the default dataset, which is initially @0. The `use` command changes the default dataset:

```
use @2 # set @2 as default
```

To load dataset from file, use one of the 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 the command `delete @n` decreases it.

The dataset can be duplicated (`@+ = @n`) or transformed, more on this in *the next section*.

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:

- if there is a name associated with the column *ycol*, the title is based on it;
- otherwise, if there is a name associated with the data block read from file, the title is set to this name;
- otherwise, the title is based on the filename

Titles can be changed using the command:

```
@n: title = 'new-title'
```

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

Dataset Transformations

There are a few transformations defined for a whole dataset or for two datasets. The syntax is `@n = ...` or `@+ = ...`. The the right hand side expression supports the following operations:

`-@n` negation of all *y* values,

`d * @n` (e.g. `0.4*@0`) *y* values are multiplied by *d*,

`@n + @m` returns `@n` with added *y* values from interpolated `@m`,

`@n - @m` returns `@n` with subtracted *y* values from interpolated `@m`,

`@n and @m` returns points from both datasets (re-sorted),

and functions:

`sum_same_x(@n)` Merges points which have distance in *x* is smaller than *epsilon*. *x* of the merged point is the average, and *y* and σ are sums of components.

`avg_same_x(@n)` The same as `sum_same_x`, but *y* and σ are set as the average of components.

`shirley_bg(@n)` Calculates Shirley background (useful in X-ray photoelectron spectroscopy).

Examples:

```
@+ = @0 # duplicate the dataset
@+ = @0 and @1 # create a new dataset from @0 and @1
@0 = @0 - shirley_bg(@0) # remove Shirley background
@0 = @0 - @1 # subtract @1 from @0
@0 = @0 - 0.28*@1 # subtract scaled dataset @1 from @0
```

Exporting Data

Command:

```
print all: expression, ... > file.tsv
```

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

> **In the GUI**
> *Data → Export*

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

```
print all: x, y, s > file.tsv
```

Any expressions can be printed out:

```
p all: n+1, x, y, F(x), y-F(x), %foo(x), sin(pi*x)+y^2 > file.tsv
```

It is possible to select which points are to be printed by replacing `all` with `if` followed by a condition:

```
print if a: x, y # only active points are printed
print if x > 30 and x < 40: x, y # only points in (30,40)
```

The option *numeric_format* controls the format and precision of all numbers.

CHAPTER 4

Models

From Numerical Recipes[14], chapter 15.0:

> Given a set of observations, one often wants to condense and summarize the data by fitting it to a "model" that depends on adjustable parameters. Sometimes the model is simply a convenient class of functions, such as polynomials or Gaussians, and the fit supplies the appropriate coefficients. Other times, the model's parameters come from some underlying theory that the data are supposed to satisfy; examples are coefficients of rate equations in a complex network of chemical reactions, or orbital elements of a binary star. Modeling can also be used as a kind of constrained interpolation, where you want to extend a few data points into a continuous function, but with some underlying idea of what that function should look like.

This chapter shows how to construct the model.

Complex models are often a sum of many functions. That is why in Fityk the model *F* is constructed as a list of component functions and is computed as $F = \sum_i f_i$.

Each component function f_i is one of predefined functions, such as Gaussian or polynomial. This is not a limitation, because the user can add any function to the predefined functions.

To avoid confusion, the name *function* will be used only when referring to a component function, not when when referring to the sum (model), which mathematically is also a function. The predefined functions will be sometimes called *function types*.

Function $f_i = f_i(x; \boldsymbol{a})$ is a function of *x*, and depends on a vector of parameters $\boldsymbol{a}$. The parameters $\boldsymbol{a}$ will be fitted to achieve agreement of the model and data.

In experiments we often have the situation that the measured *x* values are subject to systematic errors caused, for example, by instrumental zero shift or, in powder diffraction measurements, by displacement of sample in the instrument. If this is the case, such errors should be a part of the model. In Fityk, this part of the model is called *x-correction*. The final formula for the model is:

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

where $Z(x; \boldsymbol{a}) = \sum_i z_i(x; \boldsymbol{a})$ is the *x*-correction. *Z* is constructed as a list of components, analogously to *F*, although in practice it has rarely more than one component.

Each component function is created by specifying a function type and binding *variables* to type's parameters. The next section explains what are *variables* in Fityk, and then we get back to functions.

[14] http://www.nrbook.com/a/bookcpdf.php

Variables

Variables have names prefixed with the dollar symbol ($) and are created by assigning a value:

```
$foo=~5.3            # simple-variable
$bar=5*sin($foo)     # compound-variable
$c=3.1               # constant (the simplest compound-variable)
```

The numbers prefixed with the tilde (~) are adjustable when the model is fitted to the data. Variable created by assigning *~number* (like `$foo` in the example above) will be called a *simple-variable*.

All other variables are called *compound-variables*. Compound variables either depend on other variables (`$bar` above) or are constant (`$c`).

Important: Unlike in popular programming languages, variable can store either a single numeric (floating-point) value or a mathematical expression. Nothing else. In case of expression, if we define `$b=2*$a` the value of `$b` will be recalculated every time `$a` changes.

To assign a value (constant) of another variable, use: `$b={$a}`. Braces return the current value of the enclosed expression. The left brace can be preceded by the tilde (~). The assignment `$b=~{$a}` creates a simple variable.

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*.

The braces may contain any data expression:

```
$x0 = {x[0]}
$min_y = {min(y if a)}
$c = {max2($a, $b)}
$t = {max(x) < 78 ? $a : $b}
```

Sometimes it is useful to freeze a variable, i.e. to prevent it from changing while fitting:

```
$a = ~12.3 # $a is fittable (simple-variable)
$a = {$a}  # $a is not fittable (constant)
$a = ~{$a} # $a is fittable (simple-variable) again
```

In the GUI
a variable can be switched between constant and simple-variable by clicking the padlock button on the sidebar. The icons and show that the variable is fittable and frozen, respectively.

If the assigned expression contains tildes:

```
$bleh=~9.1*exp(~2)
```

it automatically creates simple-variables corresponding to the tilde-prefixed numbers. In the example above two simple-variables (with values 9.1 and 2) are created. Automatically created variables are named `$_1`, `$_2`, `$_3`, and so on.

Variables can be deleted using the command:

```
delete $variable
```

Domains

Simple-variables may have a *domain*, which is used for two things when fitting.

Most importantly, fitting methods that support bound constraints use the domain as lower and/or upper bounds. See the section *Bound Constraints* for details.

The other use is for randomizing parameters (simple-variables) of the model. Methods that stochastically initialize or modify parameters (usually generating a set of initial points) need well-defined domains (minimum and maximum values for parameters) to work effectively. Such methods include Nelder-Mead simplex and Genetic Algorithms, but not the default Lev-Mar method, so in most cases you do not need to worry about it.

The syntax is as follows:

```
$a = ~12.3 [0:20] # initial values are drawn from the (0, 20) range
$a = ~12.3 [0:]   # only lower bound
$a = ~12.3 [:20]  # only upper bound
$a = ~15.0        # domain stays the same
$a = ~15.0 []     # no domain
$a = ~{$a} [0:20] # domain is set again
```

If the domain is not specified but it is required (for the latter use) by the fitting method, we assume it to be $\pm p\%$ of the current value, where p can be set using the `domain_percent` option.

Function Types and Functions

Function types have names that start with upper case letter (`Linear`, `Voigt`).

Functions have names prefixed with the percent symbol (`%func`). Every function has a type and variables bound to its parameters. One way to create a function is to specify both type and variables:

```
%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 implicitely 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` above), the variables can be given in any order.

Function types can 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, hwhm=~0.264) # no shape is given
New function %f4 was created.
```

Functions can be copied. The following command creates a deep copy (i.e. all variables are also duplicated) of %foo:

```
%bar = copy(%foo)
```

Functions can be also created with the command `guess`, as described in *Guessing Initial Parameters*.

Variables bound to the function parameters can be changed at any time:

```
=-> %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
```

Built-In Functions

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

Gaussian:

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

a_2 here is half width at half maximum (HWHM=FWHM/2, where FWHM stands for full width...), which is proportional to the standard deviation: $a_2 = \sqrt{2\ln 2}\sigma$.

SplitGaussian:

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

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:

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

SplitLorentzian:

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

LorentzianA:

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

Pearson VII (Pearson7):

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

split Pearson VII (SplitPearson7):

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

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.

split Pseudo-Voigt (SplitPseudoVoigt):

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

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:

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

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. The approximation is very fast, but not very exact.

FWHM is estimated using an approximation called *modified Whiting* (Olivero and Longbothum, 1977, JQSRT 17, 233[15]): $0.5346 w_L + \sqrt{0.2169 w_L^2 + w_G^2}$, where $w_G = 2\sqrt{\ln(2)}|a_2|, w_L = 2|a_2|a_3$.

VoigtA:

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

split Voigt (SplitVoigt):

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

Exponentially Modified Gaussian (EMG):

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

The exponentially modified Gaussian is a convolution of Gaussian and exponential probability density. *a* = Gaussian heigth, *b* = location parameter (Gaussian center), *c* = Gaussian width, *d* = distortion parameter (a.k.a. modification factor or time constant).

LogNormal:

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

[15] http://dx.doi.org/10.1016/0022-4073(77)90161-3

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:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

Sigmoid:

$$y = L + \frac{U-L}{1 + \exp\left(-\frac{x-x_{mid}}{w}\right)}$$

FCJAsymm:

Axial asymmetry peak shape in the Finger, Cox and Jephcoat model, see J. Appl. Cryst. (1994) 27, 892[16] and J. Appl. Cryst. (2013) 46, 1219[17].

Variadic Functions

Variadic function types have variable number of parameters. Two variadic function types are defined:

```
Spline(x1, y1, x2, y2, ...)
Polyline(x1, y1, x2, y2, ...)
```

This example:

```
%f = Spline(22.1, 37.9, 48.1, 17.2, 93.0, 20.7)
```

creates a function that is a *natural cubic spline* interpolation through points (22.1, 37.9), (48.1, 17.2), ...

The `Polyline` function is a polyline interpolation (spline of order 1).

Both `Spline` and `Polyline` functions are primarily used for the manual baseline subtraction via the GUI.

The derivatives of Spline function are not calculated, so this function is not refined by the default, derivative-based fitting algorithm.

Since the Polyline derivatives are calculated, it is possible to perform weighted least squares approximation by broken lines, although non-linear fitting algorithms are not optimal for this task.

User-Defined Functions (UDF)

User-defined function types can be added 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, digits and the underscore ('_').

[16] http://dx.doi.org/10.1107/S0021889894004218
[17] http://dx.doi.org/10.1107/S0021889813016233

- 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 (bell-shaped): `height`, `center`, `hwhm`, `area`,
  - if the functions is S-shaped (sigmoidal) or step-like: `lower`, `upper`, `xmid`, `wsig`,
  - if the function is more like linear: `slope`, `intercept`, `avgy`.

  The initial values of these parameters can be guessed (command `guess`) from the data. `hwhm` means half width at half maximum, the other names are self-explaining.
- Each parameter may have a default value (see the examples below). The default value can be either a number or an expression that depends on the parameters listed above (e.g. `0.8*hwhm`). The default value always binds a simple-variable to the parameter.

UDFs can be defined in a few ways:

- by giving a full formula, like in the example above,
- as a *re-parametrization* of existing function (see the `GaussianArea` example below),
- as a sum of already defined functions (see the `GLSum` example below),
- as a split (bifurcated) function: `x <` *expression* `?` *Function1(...)* `:` *Function2(...)* (see the `SplitL` example below).

When giving a full formula, the 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 can make the syntax clear:

```
# this is 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/hwhm/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)

# split-Gaussian, the same as built-in SplitGaussian (should be in one line)
define SplitG(height, center, hwhm1=fwhm*0.5, hwhm2=fwhm*0.5) =
  x < center ? Lorentzian(height, center, hwhm1)
             : Lorentzian(height, center, hwhm2)
```

There is a simple substitution mechanism that makes writing complicated functions easier. Substitutions must be assigned in the same line, after the 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/alpha * sin(theta/2), theta=x*pi/180
```

How it works internally
The formula is parsed, derivatives of the formula are calculated symbolically, expressions are simplified and bytecode for virtual machine (VM) is created.
When fitting, the VM calculates the value of the function and derivatives for every point.

Defined functions can be undefined using command `undefine`:

```
undefine GaussianArea
```

It is common to add own definitions to the `init` file. See the section *Starting fityk and cfityk* for details.

Cutoff

With default settings, the value of every function is calculated at every point. Peak functions, such as Gaussian, often have non-negligible values only in a small fraction of all points, so if you have many narrow peaks (like here[18]), the basic optimization is to calculate values of each peak function only near the function's center. If the option `function_cutoff` is set to a non-zero value, each function is evaluated only in the range where its values are greater than the `function_cutoff`.

This optimization is supported only by some built-in functions.

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* at the beginning of this chapter, 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, and

```
Z += %function
```

adds *%function* to Z.

A few examples:

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

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

# clear F
F = 0

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

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

[18] http://commons.wikimedia.org/wiki/File:Diff_NaBr.png

The next sections shows an easier way to add a function (command `guess`).

If there is more than one dataset, F and Z can be prefixed with the dataset number (e.g. `@1.F`).

The model can be copied. To copy the model from `@0` to `@1` we type one of the two commands:

```
@1.F = @0.F         # shallow copy
@1.F = copy(@0.F)   # deep copy
```

The former command uses the same functions in both models: if you shift a peak in `@1`, it will be also shifted in `@0`. The latter command (deep copy) duplicates all functions and variables and makes an independent model.

In the GUI
click the button  on the sidebar to make a deep copy.

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
```

In the GUI
buttons  and  on the sidebar make, respectively, the HWHM and shape of all functions the same. Pressing the buttons again will make all the parameters independent.

Guessing Initial Parameters

The program can automatically set initial parameters of peaks (using peak-detection algorithm) and lines (using linear regression). Choosing initial parameters of a function by the program will be called *guessing*.

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

```
guess [%name =] PeakType [(initial values...)] [[x1:x2]]
```

Examples:

```
# add Gaussian in the given range
@0: guess Gaussian [22.1:30.5]

# the same, but name the new function %f1
@0: guess %f1 = Gaussian [22.1:30.5]

# search for the peak in the whole dataset
@0: guess Gaussian

# add one Gaussian to each dataset
```

```
@*: guess Gaussian

# set the center and shape explicitely (determine height and width)
guess PseudoVoigt(center=$ctr, shape=~0.3) [22.1:30.5]
```

- Name of the function is optional.
- Some of the parameters can be specified in brackets.
- If the range is omitted, the whole dataset will be searched.

Fityk offers a simple algorithm for peak-detection. It finds the highest point in the given range (`center` and `height`), and than tries to find the width of the peak (`hwhm`, and `area` = *height* × *hwhm*).

If the highest point is at boundary of the given range, the points from the boundary to the nearest local minimum are ignored.

The values of height and width found by the algorithm are multiplied by the values of options `height_correction` and `width_correction`, respectively. The default value for both options is 1.

Another simple algorithm can roughly estimate initial parameters of sigmoidal functions.

The linear traits `slope` and `intercept` are calculated using linear regression (without weights of points). `avgy` is calculated as average value of *y*.

In the GUI
select a function from the list of functions on the toolbar and press  to add (guess) the selected function.
To choose a data range change the GUI mode to  and select the range with the right mouse button.

Displaying Information

The `info` command can be show useful information when constructing the model.

`info types` shows the list of available function types.

`info FunctionType` (e.g. `info Pearson7`) shows the formula (definition).

`info guess [range]` shows where the `guess` command would locate a peak.

`info functions` lists all defined functions.

`info variables` lists all defined variables.

`info F` lists components of *F*.

`info Z` lists components of *Z*.

`info formula` shows the full mathematical formula of the fitted model.

`info simplified_formula` shows the same, but the formula is simplified.

`info gnuplot_formula` shows same as `formula`, but the output is readable by gnuplot, e.g. `x^2` is replaced by `x**2`.

`info simplified_gnuplot_formula` shows the simplified formula in the gnuplot format.

`info peaks` show a formatted list of parameters of functions in *F*.

`info peaks_err` shows the same data, additionally including uncertainties of the parameters.

`info models` a script that reconstructs all variables, functions and models.

The last two commands are often redirected to a file (`info peaks > filename`).

The complete list of `info` arguments can be found in *Information Display*.

In the GUI
most of the above commands has clickable equivalents.

CHAPTER 5

Curve Fitting

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[19], chapter 15.0, page 656):

> 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 the weighted sum of squared residuals (WSSR), also called chi-square:

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

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*.

The most popular method for curve-fitting is Levenberg-Marquardt. Fityk can also use a few general-purpose optimization methods. These methods are slower, some of them are orders of magnitude slower. Sometimes alternative methods find global minimum when the L-M algorithm is stuck in a local minimum, but in majority of cases the default L-M method is superior.

[19] http://www.nrbook.com/a/bookcpdf.php

Uncertainty of Parameters

(It is easier for me to find a quote than to express it myself).

From the book J. Wolberg, *Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments*, Springer, 2006, p.50:

> (...) we turn to the task of determining the uncertainties associated with the a_k's. The usual measures of uncertainty are standard deviation (i.e., σ) or variance (i.e., σ^2) so we seek an expression that allows us to estimate the σ_{a_k}'s. It can be shown (...) that the following expression gives us an unbiased estimate of σ_{a_k}:
>
> $$\sigma_{a_k}^2 = \frac{S}{n-p} C_{kk}^{-1}$$

Note that σ_{a_k} is a square root of the value above. In this formula *n-p*, the number of (active) data points minus the number of independent parameters, is equal to the number of degrees of freedom. *S* is another symbol for χ^2.

Terms of the *C* matrix are given as (p. 47 in the same book):

$$C_{jk} = \sum_{i=1}^{n} w_i \frac{\partial f}{\partial a_j} \frac{\partial f}{\partial a_k}$$

σ_{a_k} above is often called a *standard error*. Having standard errors, it is easy to calculate *confidence intervals*. But all these values should be used with care. Now another book will be cited: H. Motulsky and A. Christopoulos, *Fitting Models to Biological Data Using Linear and Nonlinear Regression: A Practical Guide to Curve Fitting*, Oxford University Press, 2004. This book can be downloaded for free[20] as a manual to GraphPad Prism 4.

> The standard errors reported by most nonlinear regression programs (...) are "approximate" or "asymptotic". Accordingly, the confidence intervals computed using these errors should also be considered approximate.
>
> It would be a mistake to assume that the "95% confidence intervals" reported by nonlinear regression have exactly a 95% chance of enclosing the true parameter values. The chance that the true value of the parameter is within the reported confidence interval may not be exactly 95%. Even so, the asymptotic confidence intervals will give you a good sense of how precisely you have determined the value of the parameter.
>
> The calculations only work if nonlinear regression has converged on a sensible fit. If the regression converged on a false minimum, then the sum-of-squares as well as the parameter values will be wrong, so the reported standard error and confidence intervals won't be helpful.

Bound Constraints

Simple-variables can have a *domain*. Fitting method `mpfit` (Lev-Mar implementation) and the methods from the NLOpt library use domains to constrain the parameters – they never let the parameters go outside of the domain during fitting.

In the literature, bound constraints are also called box constraints or, more generally, inequality constraints. Now a quotation discouraging the use of constraints. Peter Gans, *Data Fitting in the Chemical Sciences by the Method of Least Squares*, John Wiley & Sons, 1992, chapter 5.2.2:

> Before looking at ways of dealing with inequality constraints we must ask a fundamental question: are they necessary? In the physical sciences and in least-squares minimizations in particular, inequality constraints are not always justified. The most common inequality constraint is that some number that relates to a physical quantity should be positive, $p_j > 0$. If an unconstrained minimalization leads to a negative value, what are we to conclude? There are three possibilities; (a) the refinement has converged to a false minimum; (b) the model is wrong; (c) the parameter is not well defined by the data and is not significantly different from zero. In each of these three cases a remedy is at hand that does not involve constrained minimization: (a) start the refinement from good first estimates of the

[20] http://www.graphpad.com/manuals/prism4/RegressionBook.pdf

parameters; (b) change the model; (c) improve the quality of the data by further experimental work. If none of these remedies cure the problem of non-negativity constraints, then something is seriously wrong with the patient, and constrained minimization will probably not help.

Setting the domain is described in the section *Domains*.

For a convenience, the `box_constraints` option can globally disable (and re-enable) the constraints.

Fitting Related Commands

To fit model to data, use command:

```
fit [max-eval] [@n ...]
```

Specifying *max-eval* is equivalent to setting the `max_wssr_evaluations` option, for example `fit 200` is a shorthand for `with max_wssr_evaluations=200 fit`.

Like with all commands, the generic dataset specification (`@n: fit`) can be used, but in special cases the datasets can be given at the end of the command. The difference is that `fit @*` fits all datasets simultaneously, while `@*: fit` fits all datasets one by one, separately.

The fitting method can be set using the set command:

```
set fitting_method = method
```

where method is one of: `levenberg_marquardt`, `mpfit`, `nelder_mead_simplex`, `genetic_algorithms`, `nlopt_nm`, `nlopt_lbfgs`, `nlopt_var2`, `nlopt_praxis`, `nlopt_bobyqa`, `nlopt_sbplx`.

All non-linear fitting methods are iterative and evaluate the model many times, with different parameter sets, until one of the stopping criteria is met. There are three common criteria:

- the maximum number of evaluations of the objective function (WSSR), (option `max_wssr_evaluations`, 0=unlimited). Sometimes evaluations of WSSR and its derivatives is counted as 2.
- limit on processor time, in seconds (option `max_fitting_time`, 0=unlimited).
- (Unix only) receiving the `INT` signal which can be sent by pressing Ctrl-C in the terminal.

and method-specific criteria, which generally stop when no further progress is expected.

Setting `set fit_replot = 1` updates the plot periodically during fitting, to visualize the progress.

`info fit` shows measures of goodness-of-fit, including χ^2, reduced χ^2 and R-squared:

$$R^2 \equiv 1 - \frac{\sum_i (y_i - f_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Parameter uncertainties and related values can be shown using:

- `info errors` – values of σ_{a_k}.
- `info confidence 95` – confidence limits for confidence level 95% (any level can be choosen)
- `info cov` – the C^{-1} matrix.
- `print $variable.error` – standard error of specified simple-variable, `print %func.height.error` also works.

In the GUI
select *Fit* → *Info* from the menu to see uncertainties, confidence intervals and and the covariance matrix.

Note: In Fityk 0.9.0 and earlier `info errors` reported values of $\sqrt{C_{kk}^{-1}}$, which makes sense if the standard deviations of y's are set accurately. This formula is derived in *Numerical Recipes*.

Finally, the user can *undo* and *redo* fitting:

- `fit undo` – restore previous parameter values,
- `fit redo` – move forward in the parameter history,
- `info fit_history` – show number of items in the fitting history,
- `fit history n` – load the *n*-th set of parameters from history
- `fit clear_history` – clear the history

Parameters are saved before and after fitting. Only changes to parameter values can be undone, other operations (like adding or removing variables) cannot.

Levenberg-Marquardt

This is a standard nonlinear least-squares routine, and involves computing the first derivatives of functions. For a description of the algorithm 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 λ factor. When λ is equal to 0, the method is equivalent to the inverse-Hessian method. When λ 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, λ is decreased.

Two implementation of this method are available: one from the MPFIT[21] library, based on the old good MINPACK[22] code (default method since ver. 1.3.0), and a custom implementation (default method in earlier fityk versions).

To switch between the two implementation use command:

```
set fitting_method = mpfit                # switch to MPFIT
set fitting_method = levenberg_marquardt # switch to fityk implem. of L-M
```

The following stopping criteria are available for *mpfit*:

- the relative change of WSSR is smaller than the value of the `ftol_rel` option (default: 10^-10),
- the relative change of parameters is smaller than the value of the `xtol_rel` option (default: 10^-10),

and for *levenberg_marquardt*:

- the relative change of WSSR is smaller than the value of the `lm_stop_rel_change` option twice in row,
- λ is greater than the value of the `lm_max_lambda` option (default: 10^15), which normally means WSSR is not changing due to limited numerical precision.

Nelder-Mead Downhill Simplex

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.[...]

[21] http://www.physics.wisc.edu/~craigm/idl/cmpfit.html
[22] http://en.wikipedia.org/wiki/MINPACK

> 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 than 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).

NLopt

A few methods from the NLopt[23] library are available:

- `nlopt_nm` – Nelder-Mead method, similar to the one described above,
- `nlopt_lbfgs` – low-storage BFGS,
- `nlopt_var2` – shifted limited-memory variable-metric,
- `nlopt_praxis` – PRAXIS (PRincipal AXIS),
- `nlopt_bobyqa` – BOBYQA,
- `nlopt_sbplx` – Sbplx (based on Subplex),

All NLopt methods have the same stopping criteria (in addition to the common criteria):

- an optimization step changes the WSSR value by less than the value of the `ftol_rel` option (default: 10^-10) multiplied by the WSSR,
- an optimization step changes every parameter by less than the value of the `xtol_rel` option (default: 10^-10) multiplied by the absolute value of the parameter.

[23] http://ab-initio.mit.edu/wiki/index.php/NLopt

CHAPTER 6

Scripts

Working with Scripts

Fityk can run two kinds of scripts:

- Fityk scripts composed of the commands described in previous sections,
- and Lua scripts (extension `.lua`), in the Lua language.

Scripts are executed using the *exec* command:

```
exec file1.fit
exec file2.lua
exec file3.fit.gz  # read script compressed with gzip
```

Note: Fityk can save its state to a script (`info state > file.fit`). It can also save all commands executed (directly or via GUI) in the session to a script (`info history > file.fit`).

Embedded Lua interpreter can execute any program in Lua 5.2. One-liners can be run with command `lua`:

```
=-> lua print(_VERSION)
Lua 5.1
=-> lua print(os.date("Today is %A."))
Today is Thursday.
=-> lua for n,f in F:all_functions() do print(n, f, f:get_template_name()) end
0       %_1     Constant
1       %_2     Cycle
```

(The Lua `print` function in fityk is redefined to show the output in the GUI instead of writing to `stdout`).

Like in the Lua interpreter, = at the beginning of line can be used to save some typing:

```
=-> = os.date("Today is %A.")
Today is Thursday.
```

Similarly, `exec=` also interprets the rest of line as Lua expressions, but this time the resulting string is executed as a fityk command:

```
=-> = string.format("fit @%d", math.random(0,5))
fit @4
=-> exec= string.format("fit @%d", math.random(0,5))
# fitting random dataset (useless example)
```

The embedded Lua interpreter interacts with the rest of the program through the global object `F`:

```
=-> = F:get_info("version")
Fityk 1.2.1
```

All methods of `F` are documented in the section *Fityk library API*.

A few more examples.

Let's say that we work with a number of datasets, and for each of them we want to save output of the `info peaks` command to a file named *original-data-filename*.out. This can be done in one line:

```
=-> @*: lua F:execute("info peaks >'%s.out'" % F:get_info("filename"))
```

Now a more complex script that would need to be put into a file (with extension `.lua`) and run with `exec`. :

```
-- load data from files file01.dat, file02.dat, ... file13.dat
for i=1,13 do
    F:execute("@+ < file%02d.dat:0:1::" % i)
end

-- print some statistics about the loaded data
n = F:get_dataset_count()
print(n .. " datasets loaded.")

total_max_y = 0
for i=0, n-1 do
    max_y = F:calculate_expr("max(y)", i)
    if max_y > total_max_y then
        total_max_y = max_y
    end
end
print("The largest y: " .. total_max_y)
```

If a fityk command executed from Lua script fails, the whole script is stopped, unless you catch the error:

```
-- wrap F:execute() in pcall to handle possible errors
status, err = pcall(function() F:execute("fit") end)
if status == false then
    print("Error: " .. err)
end
```

The Lua interpreter was added in ver. 1.0.3. If you need help with writing Lua scripts - feel free to ask. Usage scenarios give us a better idea what functions should be available from the Lua interface.

Fityk also has a simple mechanism to interact with external programs. It is useful mostly on Unix systems. `!` runs a program, `exec!` runs a program, reads its standard output and executes it as a Fityk script. Here is an example of using Unix utilties `echo`, `ls` and `head` to load the newest CIF file from the current directory:

```
=-> ! pwd
/home/wojdyr/fityk/data
=-> ! ls -t *.cif | head -1
lab6_3-2610-q.cif
=-> exec! echo "@+ < $(ls -t *.cif | head -1)"
> @+ < lab6_3-2610-q.cif
2300 points. No explicit std. dev. Set as sqrt(y)
```

Fityk DSL

As was described in *Command Line*, each line has a syntax:

[[@...:] [with ...] command [";" command]...] [#comment]

The datasets listed before the colon (`:`) make a *foreach* loop. Here is a silly example:

```
=-> $a=0
=-> @0 @0 @0: $a={$a+1}; print $a
1
2
3
```

Command that follows the colon is run for each specified dataset in the context of that dataset. This is to say that:

```
=-> @2 @4: guess Voigt
```

is equivalent to:

```
=-> use @2
=-> guess Voigt
=-> use @4
=-> guess Voigt
```

(except that the latter sets permenently default dataset to `@4`.

`@*` stands for all datasets, from `@0` to the last one.

Usually, when working with multiple datasets, one executes a command either for a single dataset or for all of them:

```
=-> @3: guess Voigt  # just for @3
=-> @*: guess Voigt  # for all datasets
```

The whole line is parsed and partly validated before the execution. This may lead to unexpected errors when the line has multiple semicolon-separated commands:

```
=-> $a=4; print $a  # print gives unexpected error
Error: undefined variable: $a

=-> $b=2
=-> $b=4; print $b  # $b is already defined at the check time
4
```

Therefore, it is recommended to have one command in one line.

Grammar

The grammar is expressed in EBNF-like notation:

- `(*this is a comment*)`
- `A*` means 0 or more occurrences of A.
- `A+` means 1 or more occurrences of A.
- `A % B` means `A (B A)*` and the `%` operator has the highest precedence. For example: `term % "+" comment` is the same as `term ("+" term)* comment`.
- The colon `:` in quoted string means that the string can be shortened, e.g. `"del:ete"` means that any of `del`, `dele`, `delet` and `delete` can be used.

The functions that can be used in p_expr and v_expr are available *here* and *here*, respectively. v_expr contains only a subset of functions from p_expr (partly, because we need to calculate symbolical derivatives of v_expr)

Line structure

```
line       ::=   [statement] [comment]
statement  ::=   [Dataset+ '':''] [with_opts] command % '';''
with_opts  ::=   ``w:ith'' (Lname ``='' value) % '','' 
comment    ::=   ``#'' AllChars*
```

Commands

The kCmd* names in the comments correspond to constants in the code.

```
command  ::=   (
               ``deb:ug'' RestOfLine | (*kCmdDebug*)
               ``def:ine'' define | (*kCmdDefine*)
               ``del:ete'' delete | (*kCmdDelete*)
               ``del:ete'' delete_points | (*kCmdDeleteP*)
               ``e:xecute'' exec | (*kCmdExec*)
               ``f:it'' fit | (*kCmdFit*)
               ``g:uess'' guess | (*kCmdGuess*)
               ``i:nfo'' info_arg % '','' [redir] | (*kCmdInfo*)
               ``l:ua'' RestOfLine | (*kCmdLua*)
               ``='' RestOfLine | (*kCmdLua*)
               ``pl:ot'' [range] [range] Dataset* [redir] | (*kCmdPlot*)
               ``p:rint'' print_args [redir] | (*kCmdPrint*)
               ``quit'' | (*kCmdQuit*)
               ``reset'' | (*kCmdReset*)
               ``s:et'' (Lname ``='' value) % '','' | (*kCmdSet*)
               ``sleep'' expr | (*kCmdSleep*)
               ``title'' ``='' filename | (*kCmdTitle*)
               ``undef:ine'' Uname % '','' | (*kCmdUndef*)
               ``use'' Dataset | (*kCmdUse*)
               ''!'' RestOfLine | (*kCmdShell*)
              Dataset ``<'' load_arg | (*kCmdLoad*)
              Dataset ``='' dataset_expr | (*kCmdDatasetTr*)
              Funcname ``='' func_rhs | (*kCmdNameFunc*)
              param_lhs ``='' v_expr | (*kCmdAssignParam*)
              Varname ``='' v_expr | (*kCmdNameVar*)
              Varname ``='' ``copy'' ``('' var_id '')'' | (*kCmdNameVar*)
              model_id (``=''|''+='') model_rhs | (*kCmdChangeModel*)
              (p_attr ``['' expr ``]'' ``='' p_expr) % '','' | (*kCmdPointTr*)
              (p_attr ``='' p_expr) % '','' | (*kCmdAllPointsTr*)
              ``M'' ``='' expr ) (*kCmdResizeP*)
```

Other rules

```
define          ::=   Uname ``('' (Lname [ ``='' v_expr]) % '','' '')'' ``=''
                      ( v_expr |
                      component_func % ``+'' |
                      ``x'' ``<'' v_expr ''?'' component_func '':''  component_func
                      )
component_func  ::=   Uname ``('' v_expr % '','' '')''
delete          ::=   (Varname | func_id | Dataset | ``file'' filename) % ''''
```

```
delete_points   ::=   ``('' p_expr '')''
exec            ::=   filename |
                      ''!'' RestOfLine |
                      ``='' RestOfLine
fit             ::=   [Number] [Dataset*] |
                      ``undo'' |
                      ``redo'' |
                      ``history'' Number |
                      ``clear_history''
guess           ::=   [Funcname ``=''] Uname [''('' (Lname ``='' v_expr) % '','' '')'']
info_arg        ::=   ...TODO
print_args      ::=   [(``all'' | (``if'' p_expr '':'')]
                      (p_expr | QuotedString | ``title'' | ``filename'') % ''''
redir           ::=   (``>''|''>>'') filename
value           ::=   (Lname | QuotedString | expr) (*value type depends on the option*)
model_rhs       ::=   ``0'' |
                      func_id |
                      func_rhs |
                      model_id |
                      ``copy'' ``('' model_id '')''
func_rhs        ::=   Uname ``('' ([Lname ``=''] v_expr) % '','' '')'' |
                      ``copy'' ``('' func_id '')''
load_arg        ::=   filename Lname* |
                      ''.''
p_attr          ::=   (``X'' | ``Y'' | ``S'' | ``A'')
model_id        ::=   [Dataset ''.''] (``F''|''Z'')
func_id         ::=   Funcname |
                      model_id ``['' Number ``]''
param_lhs       ::=   Funcname ''.'' Lname |
                      model_id ``['' (Number | ``*'') ``]'' ''.'' Lname
var_id          ::=   Varname |
                      func_id ''.'' Lname
range           ::=   ``['' [expr] '':''  [expr] ``]''
filename        ::=   QuotedString | NonblankString
```

Mathematical expressions

```
expr          ::=   expr_or ? expr_or :  expr_or
expr_or       ::=   expr_and % ``or''
expr_and      ::=   expr_not % ``and''
expr_not      ::=   ``not'' expr_not | comparison
comparison    ::=   arith % (``<''|''>''|''==''|''>=''|''<=''|''!='')
arith         ::=   term % (``+''|''-'')
term          ::=   factor % (``*''|''/'')
factor        ::=   (`+'|'-`) factor | power
power         ::=   atom ['**' factor]
atom          ::=   Number | ``true'' | ``false'' | ``pi'' |
                    math_func | braced_expr | ?others?
math_func     ::=   ``sqrt'' ``('' expr '')'' |
                    ``gamma'' ``('' expr '')'' |
                    ...
braced_expr   ::=   ``{'' [Dataset+ '':'']  p_expr ``}''
```

The `atom` rule also accepts some fityk expressions, such as $variable, %function.parameter, %function(expr), etc.

`p_expr` and `v_expr` are similar to `expr`, but they use additional variables in the `atom` rule.

`p_expr` recognizes `n`, `M`, `x`, `y`, `s`, `a`, `X`, `Y`, `S` and `A`. All of them but `n` and `M` can be indexed (e.g. `x[4]`). Example:

`(x+x[n-1])/2`.

`v_expr` uses all unknown names (`Lname`) as variables (example: `a+b*x^2`). Only a subset of functions (`math_func`) from `expr` is supported. The tilde (`~`) can be used to create simple-variables (`~5`), optionally with a domain in square brackets (`~5[1:6]`).

Since `v_expr` is used to define variables and user-defined functions, the program calculates symbolically derivatives of `v_expr`. That is why not all the function from `expr` are supported (they may be added in the future).

`dataset_expr` supports very limited set of operators and a few functions that take Dataset token as argument (example: `@0 - shirley_bg(@0)`).

Lexer

Below, some of the tokens produced by the fityk lexer are defined.

The lexer is context-dependend: `NonblankString` and `RestOfLine` are produced only when they are expected in the grammar.

`Uname` is used only for function types (Gaussian) and pseudo-parameters (%f.Area).

```
Dataset          ::=   ``@''(Digit+|''+''|''*'')
Varname          ::=   ``$'' Lname
Funcname         ::=   ``%'' Lname
QuotedString     ::=   ``''' (AllChars - ``''')* ``'''
Lname            ::=   (LowerCase | ``_'') (LowerCase | Digit | ``_'')*
Uname            ::=   UpperCase AlphaNum+
Number           ::=   ?number read by strtod()?
NonblankString   ::=   (AllChars - (WhiteSpace | '';'' | ``#'' ))*
RestOfLine       ::=   AllChars*
```

Fityk library API

Fityk comes with embedded Lua interpreter and this language is used in this section. The API for other supported languages is similar. Lua communicates with Fityk using object `F` of type `Fityk`. To call the methods listed below use `F:method()`, for example `F:get_dof()` (not `Fityk.get_dof()`).

Note: Other supported languages include C++, C, Python, Perl, Ruby and Java. Except for C, all APIs are similar.
Unlike in built-in Lua, in other cases it is necessary to create an instance of the Fityk class first. Then you use this object in the same way as `F` is used below.
The fityk.h[24] header file is the best reference. Additionally, C++ and Python have access to functions from the ui_api.h[25] header. These functions are used in command line versions of fityk (`cfityk` or its equivalent – `samples/cfityk.py`).
Examples of scripts in all the listed languages and in the samples[26] directory.

[24] https://github.com/wojdyr/fityk/blob/master/src/fityk.h
[25] https://github.com/wojdyr/fityk/blob/master/src/ui_api.h
[26] https://github.com/wojdyr/fityk/blob/master/samples/

Here is the most general function:

`Fityk.`**`execute`**`(`*`cmd`*`)`
Executes a fityk command. Example: `F:execute("fit")`.

The `%` operator for the string type is pre-set to support Python-like formatting:

```
= "%d pigs" % 3
= "%d %s" % {3, "pigs"}
```

Input / output

`Fityk.`**`input`**(*prompt*)
Query user. In the CLI user is asked for input in the command line, and in the GUI in a pop-up box. As a special case, if the prompt contains string "[y/n]" the GUI shows Yes/No buttons instead of text entry.

Example: TODO

`Fityk.`**`out`**(*s*)
Print string in the output area. The `print()` function in built-in Lua is redefined to do the same.

Settings

`Fityk.`**`set_option_as_string`**(*opt*, *val*)
Set option *opt* to value *val*. Equivalent of fityk command `set opt=val`.

`Fityk.`**`set_option_as_number`**(*opt*, *val*)
Set option *opt* to numeric value *val*.

`Fityk.`**`get_option_as_string`**(*opt*)
Returns value of *opt* (string).

`Fityk.`**`get_option_as_number`**(*opt*)
Returns value of *opt* (real number).

Data

`Fityk.`**`load`**(*spec*[, *d*])
Load data to @*d* slot. The first argument is either a string with path or LoadSpec struct that apart from the `path` has also the following optional members: `x_col`, `y_col`, `sig_col`, `blocks`, `format`, `options`. The meaning of these parameters is the same as described in *Loading Data*.

`Fityk.`**`load_data`**(*d*, *xx*, *yy*, *sigmas*[, *title*])
Load data to @*d* slot. *xx* and *yy* must be numeric arrays of the same size, *sigmas* must either be empty or have the same size. *title* is an optional data title (string).

`Fityk.`**`add_point`**(*x*, *y*, *sigma*[, *d*])
Add one data point ((*x*, *y*) with std. dev. set to *sigma*) to an existing dataset *d*. If *d* is not specified, the default dataset is used.

Example: `F:add_point(30, 7.5, 1).`

`Fityk.`**`get_dataset_count`**()
Returns number of datasets (n >= 1).

`Fityk.`**`get_default_dataset`**()
Returns default dataset. Default dataset is set by the "use @n" command.

`Fityk.`**`get_data`**([*d*])
Returns points for dataset *d*.

- in C++ – returns vector<Point>
- in Lua – userdata with array-like methods, indexed from 0.

Each point has 4 properties: `x`, `y`, `sigma` (real numbers) and `is_active` (bool).

Example:

```
points = F:get_data()
for i = 0, #points-1 do
    p = points[i]
    if p.is_active then
        print(i, p.x, p.y, p.sigma)
```

```
    end
end
```

```
1       4.24    1.06    1
2       6.73    1.39    1
3       8.8     1.61    1
...
```

General Info

`Fityk.`**`get_info`**`(`*s*`[`, *d*`]`)
Returns output of the fityk `info` command as a string. If *d* is not specified, the default dataset is used (the dataset is relevant for few arguments of the `info` command).

Example: `F:get_info("history")` – returns a multiline string containing all fityk commands issued in this session.

`Fityk.`**`calculate_expr`**`(`*s*`[`, *d*`]`)
Returns output of the fityk `print` command as a number. If *d* is not specified, the default dataset is used.

Example: `F:calculate_expr("argmax(y)", 0)`.

`Fityk.`**`get_view_boundary`**`(`*side*`)`
Get coordinates of the plotted rectangle, which is set by the `plot` command. Return numeric value corresponding to given *side*, which should be a letter `L`(eft), `R`(ight), `T`(op) or `B`(ottom).

Model info

`Fityk.`**`get_parameter_count`**`()`
Returns number of simple-variables (parameters that can be fitted)

`Fityk.`**`all_parameters`**`()`
Returns array of simple-variables.

- in C++ – vector<double>
- in Lua – userdata with array-like methods, indexed from 0.

`Fityk.`**`all_variables`**`()`

Returns array of all defined variables.

- in C++ – vector<Var*>
- in Lua – userdata with array-like methods, indexed from 0.

Example:

```
variables = F:all_variables()
for i = 0, #variables-1 do
    v = variables[i]
    print(i, v.name, v:value(), v.domain.lo, v.domain.hi,
          v:gpos(), v:is_simple())
end
```

`Var.is_simple()` returns true for simple-variables.

`Var.gpos()` returns position of the variable in the global array of parameters (Fityk.all_parameters()), or -1 for compound variables.

`Fityk.`**`get_variable`**`(`*name*`)`
Returns variable *$name*.

`Fityk.`**`all_functions`**`()`
Returns array of functions.

- in C++ – vector<Func*>
- in Lua – userdata with array-like methods, indexed from 0.

Example:

```
f = F:all_functions()[0] -- first functions
print(f.name, f:get_template_name())          -- _1        Gaussian
print(f:get_param(0), f:get_param(1))         -- height  center
print("$" .. f:var_name("height"))            -- $_4
print("center:", f:get_param_value("center")) -- center: 24.72235945525
print("f(25)=", f:value_at(25))               -- f(25)=  4386.95533969
```

`Fityk.`**`get_function`**`(`*name*`)`
Return the function with given *name*, or NULL if there is no such function.

Example:

```
f = F:get_function("_1")
print("f(25)=", f:value_at(25))               -- f(25)=  4386.95533969
```

`Fityk.`**`get_components`**`(`*d*`[,`*fz*`])`
Returns %functions used in dataset *d*. If *fz* is `Z`, returns zero-shift functions.

Example:

```
func = F:get_components(1)[3] -- get 4th (index 3) function in @1
print(func)                   -- <Func %_6>
vname = func:var_name("hwhm")
print(vname)                  -- _21
v = get_variable(vname)
print(v, v:value())           -- <Var $_21>       0.1406587
```

`Fityk.`**`get_model_value`**`(`*x*`[,` *d*`])`
Returns the value of the model for dataset `@`*d* at *x*.

Fit statistics

`Fityk.`**`get_wssr`**`([`*d*`])`
Returns WSSR (weighted sum of squared residuals).

`Fityk.`**`get_ssr`**`([`*d*`])`
Returns SSR (sum of squared residuals).

`Fityk.`**`get_rsquared`**`([`*d*`])`
Returns R-squared.

`Fityk.`**`get_dof`**`([`*d*`])`
Returns the number of degrees of freedom (#points - #parameters).

`Fityk.`**`get_covariance_matrix`**`([`*d*`])`
Returns covariance matrix.

Examples in Lua

Show how the peak center moves between datasets:

```
-- file list-max.lua
prev_x = nil
for n = 0, F:get_dataset_count()-1 do
    local path = F:get_info("filename", n)
    local filename = string.match(path, "[^/\\]+$") or ""
    -- local x = F:calculate_expr("argmax(y)", n)
    local x = F:calculate_expr("F[0].center", n)
    s = string.format("%s: max at x=%.4f", filename, x)
    if prev_x ~= nil then
        s = s .. string.format("  (%+.4f)", x-prev_x)
    end
    prev_x = x
    print(s)
end
```

```
=-> exec list-max.lua
frame-000.dat: max at x=-0.0197
frame-001.dat: max at x=-0.0209  (-0.0012)
frame-002.dat: max at x=-0.0216  (-0.0007)
frame-003.dat: max at x=-0.0224  (-0.0008)
```

Write to file values of the model F(x) at chosen x's (in this example x = 0, 1.5, 3, ... 150):

```
-- file tabular-f.lua
file = io.open("output.dat", "w")
for x = 0, 150, 1.5 do
    file:write(string.format("%g\t%g\n", x, F:get_model_value(x)))
end
file:close()
```

```
=-> exec tabular-f.lua
=-> !head -5 output.dat
0       12.1761
1.5     12.3004
3       10.9096
4.5     9.12635
6       8.27044
```

CHAPTER 7

All the Rest

Settings

The syntax is simple:

- `set option = value` changes the *option*,
- `info set option` shows the current value,
- `info set` lists all available options.

In the GUI
the options can be set in a dialog (*Session* → *Settings*).
The GUI configuration (colors, fonts, etc.) is changed in a different way (*GUI* → ...) and is not covered here.

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

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

For example:

```
info set fitting_method  # show the current fitting method
set fitting_method = nelder_mead_simplex # change the method
# change the method only for this one fit command
with fitting_method = levenberg_marquardt fit
# and now the default method is back Nelder-Mead

# multiple comma-separated options can be given
with fitting_method=levenberg_marquardt, verbosity=quiet fit
```

The list of available options:

autoplot See *autoplot*.

cwd Current working directory or empty string if it was not set explicitely. Affects relative paths.

default_sigma Default y standard deviation. See *Standard Deviation (or Weight)*. Possible values: `sqrt` $\max(y^{1/2}, 1)$ and `one` (1).

domain_percent See *the section about variables*.

epsilon The ϵ value used to test floating-point numbers a and b for equality (it is well known that due to rounding errors the equality test for two numbers should have some tolerance, and the tolerance should be tailored to the application): $|a-b| < \epsilon$. Default value: 10^{-12}. You may need to decrease it when working with very small numbers.

fit_replot Refresh the plot when fitting (0/1).

fitting_method See *Fitting Related Commands*.

function_cutoff See *description in the chapter about model*.

height_correction See *Guessing Initial Parameters*.

lm_* Setting to tune the *Levenberg-Marquardt* fitting method.

logfile String. File where the commands are logged. Empty – no logging.

log_output When logfile is set, log output together with input (0/1).

max_fitting_time Stop fitting when this number of seconds of processor time is exceeded. See *Fitting Related Commands*.

max_wssr_evaluations See *Fitting Related Commands*.

nm_* Setting to tune the *Nelder-Mead downhill simplex* fitting method.

numeric_format Format of numbers printed by the `info` command. It takes as a value a format string, the same as `sprintf()` in the C language. For example `set numeric_format='%.3f'` changes the precision of numbers to 3 digits after the decimal point. Default value: `%g`.

on_error Action performed on error. If the option is set to `stop` (default) and the error happens in script, the script is stopped. Other possible values are `nothing` (do nothing) and `exit` (finish program – ensures that no error can be overlooked).

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.

verbosity Possible values: -1 (silent), 0 (normal), 1 (verbose), 2 (very verbose).

width_correction See *Guessing Initial Parameters*.

Data View

The command `plot` controls the region of the graph that is displayed:

```
plot [[xrange] yrange] [@n, ...]
```

xrange and *yrange* has syntax `[min:max]`. If the boundaries are skipped, they are automatically determined using the given datasets.

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

The CLI version on Unix systems visualizes the data using the `gnuplot` program, which has similar syntax for the plot range.

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         # all data will be shown
```

The values of the options `autoplot` and `fit_replot` change the automatic plotting behaviour. By default, the plot is refreshed automatically after changing the data or the model (`autoplot=1`). It is also possible to replot the model when fitting, to show the progress (see the options `fit_replot` and `refresh_period`).

Redirecting the plot command to a file saves a plot as an image:

```
plot [20.4:50] [10:20] > myplot.png
```

For now, it works only in fityk (not cfityk) and is less flexible than *Session* → *Save as Image*.

Information Display

First, there is an option `verbosity` which sets the amount of messages displayed when executing commands.

There are three commands that print explicitely requested information:

- `info` – used to show preformatted information
- `print` – mainly used to output numbers (expression values)
- `debug` – used for testing the program itself

The output of `info` and `print` can be redirected to a file:

```
info args > filename    # truncate the file
info args >> filename   # append to the file
info args > 'filename'  # the filename can (and sometimes must) be in quotes
```

The redirection can create a file, so there is also a command to delete it:

```
delete file filename
```

info

The following `info` arguments are recognized:

- *TypeName* – definition
- *$variable_name* – formula and value
- *%function_name* – formula
- `F` – the list of functions in *F*
- `Z` – the list of functions in *Z*
- `compiler` – options used when compiling the program
- `confidence level @n` – confidence limits for given confidence level
- `cov @n` – covariance matrix
- `data` – number of points, data filename and title
- `dataset_count` – number of datasets
- `errors @n` – estimated uncertainties of parameters
- `filename` – dataset filename

- `fit` – goodness of fit
- `fit_history` – info about recorded parameter sets
- `formula` – full formula of the model
- `functions` – the list of %functions
- `gnuplot_formula` – full formula of the model, gnuplot style
- `guess` – peak-detection and linear regression info
- `guess [from:to]` – the same, but in the given range
- `history` – the list of all the command issued in this session
- `history [m:n]` – selected commands from the history
- `history_summary` – the summary of command history
- `models` – script that re-constructs all variables, functions and models
- `peaks` – formatted list of parameters of functions in *F*.
- `peaks_err` – the same as peaks + uncertainties
- `prop` *%function_name* – parameters of the function
- `refs` *$variable_name* – references to the variable
- `set` – the list of settings
- `set` *option* – the current value of the option
- `simplified_formula` – simplified formula
- `simplified_gnuplot_formula` – simplified formula, gnuplot style
- `state` – generates a script that can reproduce the current state of the program. The scripts embeds all datasets.
- `title` – dataset title
- `types` – the list of function types
- `variables` – the list of variables
- `version` – version number
- `view` – boundaries of the visualized rectangle

Both `info state` and `info history` can be used to restore the current session.

> **In the GUI**
> *Session* → *Save State* and *Session* → *Save History*.

print

The print command is followed by a comma-separated list of expressions and/or strings:

```
=-> p pi, pi^2, pi^3
3.14159 9.8696 31.0063
=-> with numeric_format='%.15f' print pi
3.141592653589793
=-> p '2+3 =', 2+3
2+3 = 5
```

The other valid arguments are `filename` and `title`. They are useful for listing the same values for multiple datasets, e.g.:

```
=-> @*: print filename, F[0].area, F[0].area.error
```

`print` can also print a list where each line corresponds to one data point, as described in the section *Exporting Data*.

As an exception, `print expression > filename` does not work if the filename is not enclosed in single quotes. That is because the parser interprets > as a part of the expression. Just use quotes (`print 2+3 > 'tmp.dat'`).

debug

Only a few `debug` sub-commands are documented here:

- `der` *mathematic-function* – shows derivatives:

```
=-> debug 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
```

- `df` *x* – compares the symbolic and numerical derivatives of *F* in *x*.
- `lex` *command* – the list of tokens from the Fityk lexer
- `parse` *command* – show the command as stored after parsing
- `expr` *expression* – VM code from the expression
- `rd` – derivatives for all variables
- `%function` – bytecode, if available
- `$variable` – derivatives

Other Commands

- `reset` – reset the session
- `sleep` *sec* – makes the program wait *sec* seconds.
- `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 the `!`) to the `system()` call (i.e. to the operating system).

Starting fityk and cfityk

On startup, the program runs 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 the `--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; columns and data blocks can be specified in the normal way, see *Loading Data*.

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

```
$ fityk -h
Usage: fityk [-h] [-V] [--full-version] [-c <str>] [-g <str>] [-I] [-r] [script or␣
↪data file...]
  -h, --help                show this help message
  -V, --version             output version information and exit
  --full-version            print version with additional info 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)

$ 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
  -n, --no-plot         disable plotting (gnuplot)
  -q, --quit            don't enter interactive shell
```

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

```
wojdyr@ubu:~/foo$ ls *.rdf
dat_a.rdf  dat_r.rdf  out.rdf
wojdyr@ubu:~/foo$ cfityk -q -I "=-> set verbosity=-1, autoplot=0" \
> *.rdf "=-> @*: print min(x if y > 0)"
in @0 dat_a: 1.8875
in @1 dat_r: 1.5105
in @2 out: 1.8305
```