# Real-time plotter and fitter server manual

Oleksiy Onishchenko

January 14, 2021

## 1 Notation

- File and folder names are given in the following font ***myfavoritefile.py***
- Angled brackets with a datatype inside, so like <integer> mean that the user must provide one instance of the corresponding data type. It is possible that one has to provide lists, in which case the data type of the entries in the list will also be provided in angles brackets, so say a list of two strings would be <list(<string>,<string>)>. If there are several data types possible (which should not happen too much), they will be separated by a comma (so say <integer, string>); do NOT interpret this as a list of two entries, this is integer OR string, one set of single brackets means only one instance. If we don't want to specify the datatype, we will use the word "any", and we will use ... to denote an undefined number; for example, a list of integers of arbitrary length will be given as <list(<integer>,...)>
- Three

## 2 Basic principles, design philosophy, and governing ideas

## 3 Client-server structure

The principal idea of the project is to make this plotter-fitter work as a remote server. It should be possible to send all data and commands remotely and get answers back. Of course there is a GUI layer on top, in order to visualize the plots and fits, read the results off the screen, and also do some manual adjustment of plotting, but a big focus of programming this tool is to make it function as a remote server.

The main server class `TCPIPserver` is located in file ***socketserver.py***. It is defined to work in two modes: either one-way, where it only receives messages from the client and does not send any info back, or in two-way more, where a round of communication consists of message reception and transmission. At some point in the future this should probably be combined into a single server without a real distinction in the functions themselves whether

# 4 How the fitter itself functions

`TCPIPserver` calls class `GeneralFitter1D` with an instance of `Fitmodel` class as the only parameter. Fitting itself is done in function `GeneralFitter1D.doFit()`, meaning that the optimizer from *(*scipy) is called in that function.

# 5 TCP/IP commands

## 5.1 General format of commands

All commands must be sent as character strings in **utf-8** encoding (if that's impossible, one could implement ascii encoding/decoding procedure, but it's not done yet). Each individual command must conform to JSON-RPC2.0 standard. Therefore the strings look as follows:

**JSON-RPC2.0 command format**

`{ "jsonrpc": "2.0", "method": <string>, "params": <dictionary>, "id": <integer> }`

where the dictionary corresponding to "params" is a data structure corresponding to the python dictionary and has the format {<string>:<any>,...}, so it is a list of comma-separated pairs, of arbitrary length, where inside each pair itself the entries are separated by a colon. The first element of each pair is a string (that's known as the *key*), and the second element can be in principle any datatype, including a dictionary itself (that's known as *value*). Note the curly braces around: they must be there.

For those who program in Python and understand the lingo: the value corresponding "params" has the standard form of a Python dictionary, with all keys being strings. For those who do not program in Python and do not understand the lingo: make sure to build this dictionary exactly in the format described above.

## 5.2 Available "method" values and responses from the server

As of now, the implemented methods are

- `''config''` This will send configuration parameters to the plotter and fitter, such as axis labels, plot label, legend labels, as well as what kind of function to plot, what are the plot starting parameters, plot limits, etc.
- `''data''` This send data point by point to the plotter. Once the plotter receives each data point, it immediately puts it on the screen
- `''fitrequest''` This tells the plotter which fit to send back to the client.

whenever the method is "config" or "data", the server send to the client a JSON-RPC2.0 string of the following form:

`{ ''jsonrpc'': ''2.0'', ''result'': ''MessageReceived''}`

Whenever the method is "fitrequest", the server will respond as follows:

`{ ''jsonrpc'': ''2.0'', ''result'': <dictionary>}`

and the result dictionary will contain fit results in the form like

`{ ''frequency'': {''fitvalue'': 100000., ''fiterror'': 3000.}, ... , ''costfunction'': 200. }`

so the result will contain a dictionary of fit parameters with their corresponding fit values and possibly fit errors, and finally it will also contain the minimum cost function that was obtained after optimization. One could also imagine extending this dictionary of results to give more information about the fit, but that should be easy to do given that this is simply extending the dictionary, and one has the tree structure such that on the right inside each colon-separated pair one can have another dictionary (Python lingo: dictionaries can have dictionaries as their values).

### 5.3 Available "params" values

As we have seen, "params" in the request must be sent as dictionaries, so in the form {<string>:<any>,...}. The following table summarizes what can go into these dictionaries

Sending “params” to the server

We list the parameters with an explanation of the possible values (the parameter literal is before the colon, the possible values are after the colon, explanation in parentheses
**Case 1: “method” is “config”**

- “clearData” : “all” (This clears the data from the plot and the plotter memory. It is worthwhile in the future to implement, like, clearing a single curve)
- “setAxisLabels” : <list(<string>,<string>)> (first one in the list is x-axis label, second one is y-axis label)
- “setPlotTitle” : <string>
- “setPlotLegend” : <dictionary(<string>:<string>,...)> (The first string in the dictionary has form “curve1” for example, etc. which just defines the curve to use, the second string will be the label that we want to put in the legend for that curve)

... (more commands in the same pattern)
**Case 2: “method” is “data”** In this case the dictionary that goes with the params will have the form {“xval”:<double>, “yval”:<double>, “errorval”: <double> } (this can be modified for adding points to curves, but the logic of it being a dictionary will stay the same)
**Case 3: “method” is “fitrequest”** In this case the dictionary that goes with params will be {“curve”:<integer>}, where the integer is the number of the curve whose fit we want to retrieve

# 6 Background software and hardware requirements