Real-time plotter and fitter server manual

Oleksiy Onishchenko

October 8, 2020

1 Basic principles, design philosophy, and governing ideas

2 TCP/IP commands

All commands must be sent as character strings in utf-8 encoding for now (if that's impossible, one could implement ascii encoding/decoding procedure). Most of the times, commands must be separated with a semicolon [;]. Different options within a single command must be separated with a comma [,].

Before any commands can be sent, one needs to write down the IP address and the port of the computer where the server is running. If the server is running on the same computer as the client, the IP address will be [127.0.0.1]. The port can basically be chosen at will, as long as its value is a large number, beyond the range of the standard reserved ports. One also has to establish the buffer size for communication, but that does not matter mostly. A common value is [2048], which corresponds to 2048 bytes.

Most of the commands do not require or expect a response from the server. They are in a sense a one-way communication. The box below shows the general setup of sending commands to the plotter, assuming that they are sent from Python. The goal is to make this a general-purpose plotter and fitter server, so since it works with TCP/IP, it doesn't really matter which programming language the commands are sent from, Python is only an example

General approach to sending commands

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
message_string = "my favorite message"
message = my_information.encode("utf-8",errors="ignore")
sendres = s.sendall(message)
s.close()
```

The most important idea here is that one must close the socket communication after sending every message. Sockets are recycled.

One class of commands sends data to the plotter. The result will be that the points are plotted on the main plotter canvas. The following box provides the list of these commands.

Sending data point by point

Sending data without error bars:

```
listdata errorbar_no xx.xxxxx vv.vvvvv vv.vvvvv ...
```

Here, [x], [v] refer to the digits of the numbers representing the x-axis and the values respectively. The x-axis can be anything, for example time. one has to put at most 6 digits after the comma. Each value [vv.vvvvv] will correspond to a curve on the plot. The number of values, and consequently the number of curves, can be whatever. However, you have to always send the same number of curves, once you started sending that number. For example, if you started sending 2 curves, keep sending always two curves until you clear data, otherwise you will generate an error.

Sending data with error bars:

```
listdata errorbar_yes xx.xxxxx vv.vvvvv ee.eeeee vv.vvvvv
ee.eeeee ...
```

Here, [x], [v], [e] refer to the digits of the numbers representing the x-axis, the values, and the corresponding error bars respectively. The x-axis can be anything. Each combination [vv.vvvvv ee.eeeee] will correspond to a curve on the plot. One must always feed values with error bars, in other words one cannot plot one curve with error bars and one curve without, that will generate an error, and one must always send the same number of curves unless one has cleared the plot and starts plotting again.

After one is done with plotting, and possibly fitting, etc., one has to clear the plots and the memory for the next set of data to be sent, plotted, and processed. That can be achieved with the [clear_data] command:

Clearing data plot and fit results

```
config; clear_data all
```

For now, only [all] is implemented, but the idea is to implement the possibility of clearing one particular curve without touching the other ones.

We can also transmit plot parameters, such as axes labels, plot title, legend, via TCP/IP. In addition, we can send the function choice for fitting, starting parameters for

the fit, and the request to do the fit. All of these are **configuration commands**, which can be sent in a string starting with [config].

Configuration command string (config)

At the beginning of the command string we always write:

```
config;
```

From then on, the following commands are defined:

`set_axis_labels myXlabel, myYlabel`
myXlabel and myYlabel and the text that will label the axes

`set_plot_title myPlotTitle`
myPlotTitle will be written above the plot area

`set_fit_function fitfunctionname`
fitfunctionname must exactly match one of the defined fit functions in file mathfunctions/fitmodels.py

`set_curve_number number`
number is the number (string) of one of the curves that has been sent to the plotter. See the box above about sending data point by point, the first point would correspond to number being 0, the second point corresponds to number being 1, etc.

`set_starting_parameters param1name : value1 , param2name : value2, ...`
Not yet implemented fully These parameters are what is given in the parameter dictionary of the fit function, which is in mathfunctions/fitmodels.py The names of the parameters must be written exactly as in that parameter dictionary, otherwise it will produce errors. The fitter will take these parameters as initial values.

`do_fit`
Not yet implemented fully This will perform the fit and display the graph on top of the data points

The commands can be sent in any order, separated by semicolons. So a valid command string would be like
`config; set_axis_labels myX1, myY1; set_plot_title myCoolData; ...`
where ... refers to additional commands. It is also ok to send command strings separately, they are processed as they come in, so for example one can send like
`config; set_axis_labels myX1, myY1; set_plot_title myCoolData; ...`
`config; set_fit_function sinewave; set_curve_number 0; do_fit`
Remember that each of the commands4has to be sent as a separate session of TCP/IP communication. The thing though is that it's better to send the whole command statement as a single string, because otherwise one can run into problems if one for example calls [do_fit] before setting the curve number, and so on.

2.1 A subsection

More text.

3 Background software and hardware requirements