Real-time plotter and fitter server manual

Oleksiy Onishchenko

July 12, 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>,...)>
- Class names and other examples of source code are given in `this font`:

2 Basic principles, design philosophy, and governing ideas

2.1 Communication: JSON-RPC 2.0 format, JSONread class

All communications to the server and back are sent via JSON-RPC interface. The file that that handles JSON-RPC format details is ***JSONinterpreter.py***, and it defines the main class `JSONread`, which takes care of checking whether the message complies with JSON-RPC 2.0 format, and very importantly checking whether the message received has meaning inside this software package, so whether the parameters included in the JSON message can be interpreted by the fitter and plotter server. For this purpose, there are class attributes defined, like `method_keys, doClear_message_keys`, etc. This should be helpful to make the program **extensible** if necessary: One simply adds some more of these class attributes as necessary, or alternatively one extends the lists for the attributes with new entries.

Then there's a defined function to handle every legal method given in the JSON-RPC message. It works like this: if the method is "doFit", for example, then the

member function of class `JSONread` is called `__parse_doFit_message()`, for "addData" method we have `__parse_addData_message()` function, so it's always like `__parse_<methodvalue>_message` All these helper functions are meant to be private to the class, and they are called by the `parse_JSON_message()` from the same class. Each of these functions takes as its argument the dictionary that gets parsed from the incoming JSON-RPC message. **Important:** each of these helper functions outputs a Python list of tuples (can be of any nonzero length); each tuple has exactly two entries, the first is a string, the second is in principle any data type. The first entry in the tuple is thus the string name of the function that will be called downstream in the program. The second entry is the data that will be passed to that function. I think that this structure makes for a very clean and uniform interface, and facilitates extensibility.

`parse_JSON_message()` is the only public function from this class. It takes the the JSON previously converted into a string by the utilities of the standard Python `json` package, and first uses `__check_JSON()` to see if this is a valid JSON-RPC 2.0 string, and if it is, then this message is converted into a Python dictionary and the "method" entry is legal in this program, this dictionary is fed into one of the helper functions described in the paragraph above. **Important point:** those helper functions are not called as one would call normal functions with their names, but rather the standard Python `getattr()` function is used very extensively in the program. This allows us to use strings, and all string formatting capabilities, to cleanly and uniformly do in our code whatever we want, without defining multiple copy-pasting multiple variable names at different places in the code. For example, here, once the class variable `method_keys` has been defined for `JSONread` class, the strings that come in via JSON-RPC can be checked against the contents of that variable, and then functions can directly be called based on those strings. Finally, this function returns back from came out of one of those helper functions.

Right now, anytime the function `interpret_message()` is called in ***GUI.py***, we make an instance of `JSONread` class and pass the message to it. Maybe a better idea would be to create one instance of this class and then keep calling the function `parse_JSON_message()` from it, always with the appropriate message, because that's fundamentally the only thing it does.

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

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>,...}: 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

- `''doClear''` This will clear the information from the screen and/or from the server memory.
- `''setConfig''` This will send configuration parameters to the plotter and fitter, which are axis labels, plot label, legend labels

- `''addData''` This send data point by point to the plotter, or alternatively lists of data points at once. Once the plotter receives each data point, it immediately puts it on the screen
- `''doFit''` This will perform the fit to (some of) the data that has been previously sent to the server. Cropping is also done here, because cropping is something that's used only for fitting.
- `''getFitResult''` This tells the plotter which fit to send back to the client.
- `''getConfig''` Not implemented yet, but envisioned to get configurations back to the server

NOTE: Not sure if the following has been implemented correctly already

whenever the method is anything other than "getFitResult" or "getConfig", the server send to the client a JSON-RPC2.0 string of the following form:

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

Whenever the method is "getFitResult", 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 to go with each method. The parameter literal is shown before the colon, the possible values are after the colon.

Case 1: "method" is "doClear"

- "everything" : "" (empty string)

  This clears everything, so data, fits, axis labels, etc. This is also the only function which will make the window autoscale appropriately for the next

plot. Use it between sending completely different sets of data

- "config": <str,"all">

  "all" will clear all given configurations, and if one gives for example "plotTitle" as argument, one gets rid of the title, "axisLabels" gets rid of the labels

- "data": <int,"all">

  Either deletes all data associated with the curve given by int, or associated with all curves

- "plot": <int,"all">

  Either deletes the plot associated with the curve given by int, or associated with all curves. Note that in this case the data are not cleared, only the visual on the screen

- "replot": <int,"all">

  If the clear-plot call has been made before, then this will replot the curve, so bring the visual back on the screen

Case 2: "method" is "setConfig"

- "axisLabels" : <list(<string>,<string>)>

  First one in the list is x-axis label, second one is y-axis label

- "plotTitle" : <string>

- "plotLegend" : <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)

Case 3: "method" is "addData"

- "dataPoint" : <dictionary>.

  This will add a single data point (a point to a single curve). The dictionary that goes with the dataPoint have the form

  ```
  { ''curveNumber'':<integer>, ''xval'':<float,int>,
  ''yval'':<float,int>, ''yerr'':<float,int>,
  ''xerr'':<float,int> },
  ```

  and in this case "xerr" and "yerr" are optional. This will add a single data point to the curve that has been specified.

- "pointList" : <list(<dictionary>,...)>.

  This will add multiple data points in a single communication run. It's as many data points as the dictionaries in the list, and each dictionary has exactly the same format as in the item above. The list is specified by using these symbols before and after:

  [ ]

  so it's formatted as a standard Python list.

  NOTE: this list is parsed directly in the JSONinterpreter.py, so the main program gets directly a list of commands corresponding to "dataPoint".

Case 4: "method" is "doFit"

- "fitFunction": <str>

  Name of the fitting function to be used in case "performFitting" is called afterwards. This name must be defined in ***fitmodels.py***. The availble fit functions are listed, together with the parameters, in 5.4

- "curveNumber": <int>

  The curve number that will be processed in this particular call iteration of doFit routines.

- "startingParameters" : <dictionary>

  This dictionary must come in the form

  `{ ''frequency'':<float>, ''center'':<float>,...}`, so these will be the names of the variables are they are defined in the fit model. This depends of course on what one wants to fit, Gaussian, sinusoidal, Lorentzian, etc.

  Whatever parameters from the fit model are not specified will be handled by the automatic parameter estimation routine (which, OK, could be good or bad). This parameter is in principle optional, so one can not specify it at all, in which case all starting parameters will be handled by the automatic initial parameter estimation routine.

- "startingParametersLimits" : <dictionary>

  This is optional, this will tell the fitter the limits of parameter search. If one doesn't provide this, they will be determined by the automatic estimation routines. The dictionary must come in a form very similar to startingParameters, but the value is a 2-entry list, so it will be like this: `{ ''frequency'':<list>(<float>,<float>), ''center'':list(<float>,<float>),...}` . It is not required to provide the limits for each parameter, however if one does provide limits, one

must provide both lower (on the left) and upper (on the right) limits. One **cannot** simply leave one limit empty.

- "cropLimits" : <list(<float or int, "-inf">,<float or int,"inf">)>

  This will crop the data before sending it to the fitter. This item is optional, if not provided, all data for the given curve will be used. If one wants a one-sided limit, then the other side must be specified as "-inf" or "inf"

- "fitMethod" : <str>

  This is the name of the fit method from the scipy.optimize library. Currently we have "minimize", "least_squares", "basinhopping", "differential_evolution", "shgo", "dual_annealing". Brute force is not implemented. If fit method is not provided, it will default to "least_squares".

  In case "fitFunction" is "curvepeak", then this option must be either "findmax" or "findmin".

- "fitterOptions": <dict>

  The dictionary is passed directly to the scipy.optimize method. The function will not check if the given keyword arguments make sense for the optimization algorithm, that is up to the user.

- "monteCarloRuns" : <dict>

  The key-value pairs in this dictionary have to be of the form "string":<int>, where the string is exactly the name of the parameter for which the tried have to be done between the min and max parameter bounds values, and int refers to how many random points have to be taken.

  Warning: Monte Carlo procedures are not yet implemented very well. One has to check this, so that they work when necessary, and don't crash the program when they are not implemented for some options

- "performFitting": <str>

  This parameter asks the program to do the fit. The value must **always** be an empty string, "", and it is ignored. It is only there for consistency with all other parameters

Important note: Any string on the left of the colon (so the dictionary keys "clearData", "axisLabels", "plotTitle", etc, must consist of a word starting with a lowercase letter, followed by one or more words starting with an uppercase letter, without spaces. Inside, the program does string parsing by detecting the locations of the capital letters, and then it calls the corresponding functions, which have exactly the same name but with no capital letter and with underscore separators between the words. So the function called when parameter "clearData" is given

will be “first_second()”. See Section 2 for a more detailed explanation of the structure.

Case 5: “method” is “getFitResult”

- “curveNumber” : <integer>

  This will send back a JSON-RPC-formatted response, with the result being a dictionary with keys being the fit parameters and values being the fitted values.

  NOTE: This does not yet send the fit confidence intervals, and also it is not quite sure how the fit errors are treated. That has to be yet taken care of.

Case 6: “method” is “getConfig”

- None

This option is not implemented yet

5.4 Available fit functions and names of fit parameters

6 Background software and hardware requirements