Running BOUT++ using boutGUI (11th August 2015):

Installation:

Currently all the files required to run the simulations are contained within the simulations folder on GitHub at https://github.com/joe1510/GUI-for-SOL1D . Download the most recent .tar that contains all of these. Important file paths are stored within the ./config/config.ini file and can either be changed manually or within the GUI itself. Most of the time the .ini can be ignored.

```
[exe]
path = /hwdisks/home/jh1479/BOUT-dev/examples/bout-sol1d/sol1d

[archive]
path = /hwdisks/home/jh1479/python/Archive/
```

Un – tar the folder wherever you desire and run the boutGUI.py using ipython or otherwise. All other files can be ignored. The GUI has been designed to be as portable as possible so should be able to run anywhere using **any** of the BOUT models provided the above file paths are correct.

Problems with running the GUI:

Please note that if the GUI seems to be working generally fine BUT the actual simulations are getting stuck either immediately or after a few steps then check to if **debug mode** is running in the actual simulation code. This requires a line to be commented out, in sol1d this is:

```
int rhs(BoutReal t) {
    // fprintf(stderr, "\rTime: %e", t);      THIS LINE NEEDS TO BE COMMENTED OUT!!

    mesh->communicate(Ne, NVi, P, Nn);
    if(evolve_nvn) {
```

If you get a ‘segmentation fault’ when trying to run a simulation it is most likely two things causing this problem:

1) The wrong model for input file has been chosen. If the sol1d BOUT.inp file is chosen but the advdiff model is run then the inputs are different and it will crash.
2) The configparser doesn’t like headings with numbers, a commonly used one is [2fluid]. Because 2fluid is common then it has been automated so that if it encounters this heading it is replaced with TWOfluid then turns it back to 2fluid before running. No other issues with control files have been found but if so it would be possible to write functions of the same form as change2fluid() in boutGUI.py and reverse2fluid() in runboutSim.py to change headings automatically.

Running the GUI:

The Load Tab:

Load useforruns.py using ipython or otherwise. This will open up the graphical interface, an example is shown below:

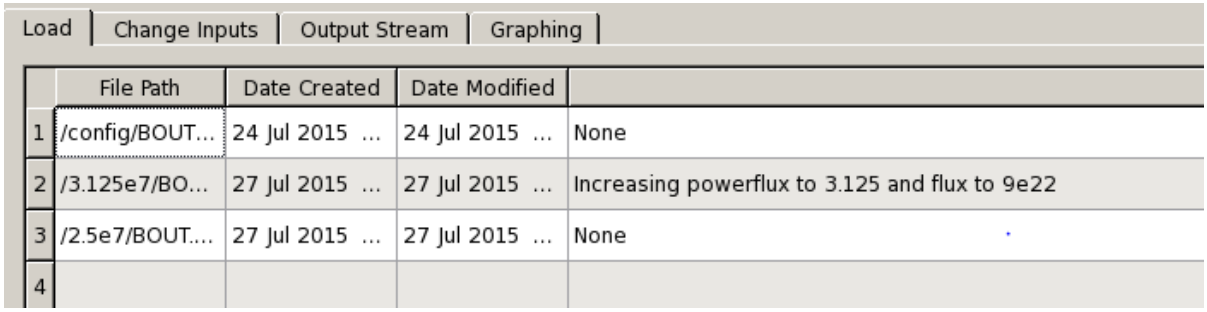


Figure 1 Example load page

This lists all of the BOUT.inp control files contained anywhere within the specified archive along with some corresponding information such as user added comments. Double click to load a file or click the load button. The table can be sorted alphabetically by each category.

At the bottom of the screen is a line to say where the current archive is sourced from. On the first time that the application is run this will say 'None Loaded or bad file path'. Click on file -> archive location and a graphical file browser will appear. This can be used to choose an old archive or create a new one. The archive location can be changed at any time.

Loading a file automatically takes you to the change inputs tab where the inputs are loaded according to the input file.

The Change Inputs Tab:

Here the values for all the inputs are shown, consistent with the values contained within the BOUT.inp control file that was loaded in the previous tab. Depending on the data type different inputs are created. For true/ false statements a combo box is create. For a float a spin box with two decimal places is create unless the number is very large and exponential notation is used in a line edit. Typing 2.01e7 in a line edit is the same as 20100000 in a double spin box. Integers create a spin box with no decimal places. All other inputs are created with a line edit (such as functions). These values can then be changed as the user requires.

At the bottom of the screen are two lines, both file paths. One is the open control file the other the file path of the simulation that is in use. To change the open control file return to the load tab and to change the simulation code click file -> simulation code.

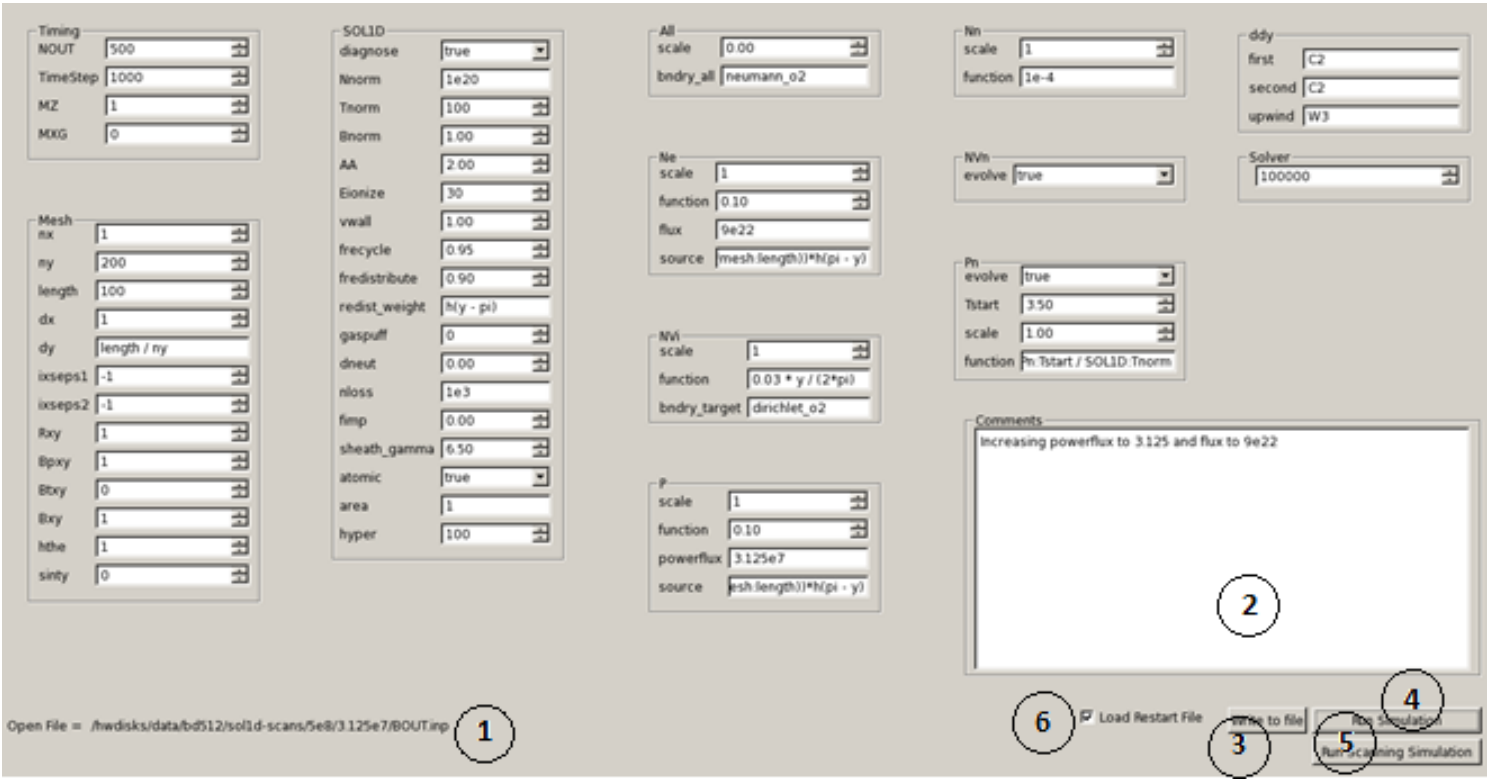


Figure 2 Example change inputs page

1. This shows the loaded config file that is currently being edited.

2. A box in which comments about what changes have been made can be added as the user wishes.

3. **'Write to File'** – this simply saves any changes and does nothing else.

4. **'Run simulation'** – this will load a dialog box requesting a folder path where all the data from the simulation will be stored. The automatic value for this is the same as the path that the data was loaded from so overwriting the old. This is useful for continuing previous data when steady state is trying to be reached but care should be taken if variables are changed as the user generally won't want to overwrite the data in this case, then copies the config to a temporary folder to run the SOL1D simulation from.

5. **'Run scanning simulation'** – similar to the above except the user can run more than one run automatically by giving a start and end value for a parameter and the increment that it is increased each time. Care should be taken here to ensure enough time steps are used to ensure steady state is reached as there are currently no inbuilt checks.

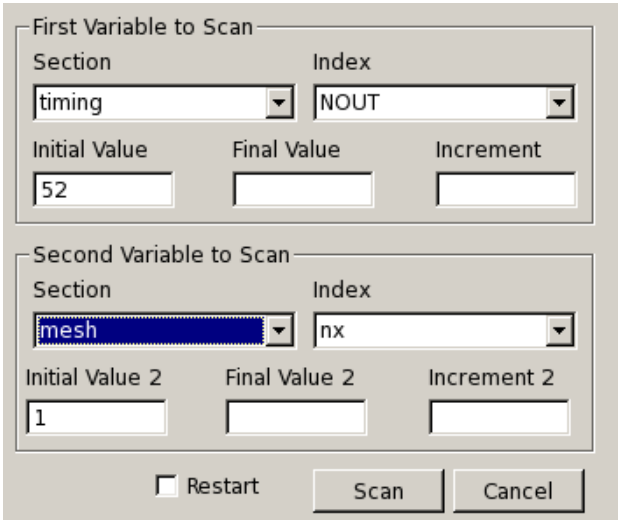
   When setting up scanned simulations the naming convention is automatic, as runs happen the folders are named according to their individual attributes. This is as follows – 'name of parameter changed'.'value'.0.individual Number. Individual number is created to make each folder unique. Example: for a simulation increasing NOUT to 502 from 500:

   NOUT.500.0.1, NOUT.501.0.1, NOUT.502.0.1

If however these files exist we get instead:

NOUT.500.0.2, NOUT.501.0.2, NOUT.502.0.2

When it running a scanned simulation the following pop up will appear after a save path for the initial folder is selected:



There is a choice of scanning either one or two variables. There are two boxes of inputs that need filling out. In each there is a combo box where the heading is chosen. Corresponding to the heading in the control file. This then loads the options for that heading in the second combo box. When an option is selected it loads the value of that option to the initial value box. The final value and increment then need to be typed.

The increment can take two forms – percentage and raw. Raw means that the number input is added on each run. So if NOUT is run from 50 to 52 with an increment of 1 then each run will be 50, 50+1, 50+1+1 and then stop. Percentage means that the number input is a multiplier. So it is possible to increase by 20% in each run by inputting 1.2 as the increasing. Choose between increment and raw in the other options box.

If only one variable is filled out then the program should only scan one variable. If two are filled out it currently alternates between adding increment one to initial value one and then increment two to initial value two suitable for powering up a simulation for example where you might want to increase one variable then another to match it and keep a system stable. Soon it will also have a mode where one variable is increase once and then the other will go for its entire range from initial to final value. Then the first variable will have the addition of its increment after and the second variable goes through its entire cycle again. Useful for looking at the impact of changing one variable on the system as a whole.

6. If the restart box is checked then the simulation will run from the previous timestep provided that restart files exist in the folder in which the config file was loaded from.

Different Models:

The graphical interface was initially designed to work with the BOUT SOL1D model, designed to look at a one dimensional approximation of the scrape off layer. However it has been developed recently so that it can work with any control file given to it. This means that automatic creation of input boxes was required.

A 'group box' or 'frame' is created for each heading on the BOUT.inp file that has been loaded. For each option within that heading either a line edit or a combo box is created. This depends upon that data type of the value of that option. These frames are then sorted automatically and the program attempts to find a good way of displaying them. Generally however the positioning of these will need reconfiguring to get the best view for the user. See the problems section of page one for information about problems encountered with this automation.

Repositioning and resizing options can be found under **view -> Edit Input Positioning** or by using the shortcut, **ctrl + B.** Clicking Apply makes the changes and keeps the screen open for further changes. All the numbers represent pixels and changing them affects the coordinates of the objects appearing on the screen. The values of the positioning are stored within the boutGUI folder in /config/config.ini if it is required that these are to be changed manually.

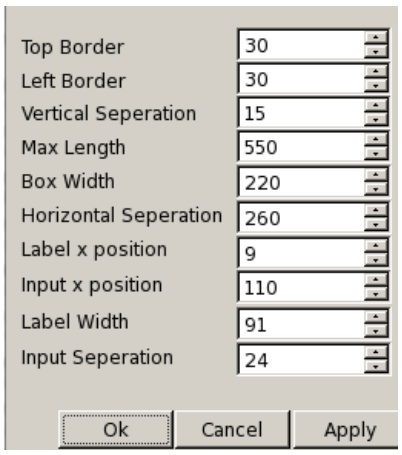


Figure 3 Example of position and sizing options

When a different model is loaded it is up to the user to load the correct simulation code that works with that model. This is done as mentioned previously by clicking **file -> simulation code.** This brings up a graphical file explorer allowing the user to find their simulation code. A segmentation fault when running the code generally means that code and the control file are not compatible with one another, try a different simulation code.

If there is a very long section then it cannot possible fit on the screen. A fix for this may be implemented to allow boxes to take up two columns but this may be very difficult. Because generally only one or two inputs are focussed on at a time when running simulations then for now it will have to do that the boxes are repositioned by the user so that they can see the boxes that are needed for those runs, this may be ugly but will have to do for now.

File History:

It is possible to view the file history of any loaded config file by using file -> history. This will load a list of folders and times when the config file existed in those folders so you know the history of the simulation. This information is stored in the record.ini file within each of the run folders within the selected archive.

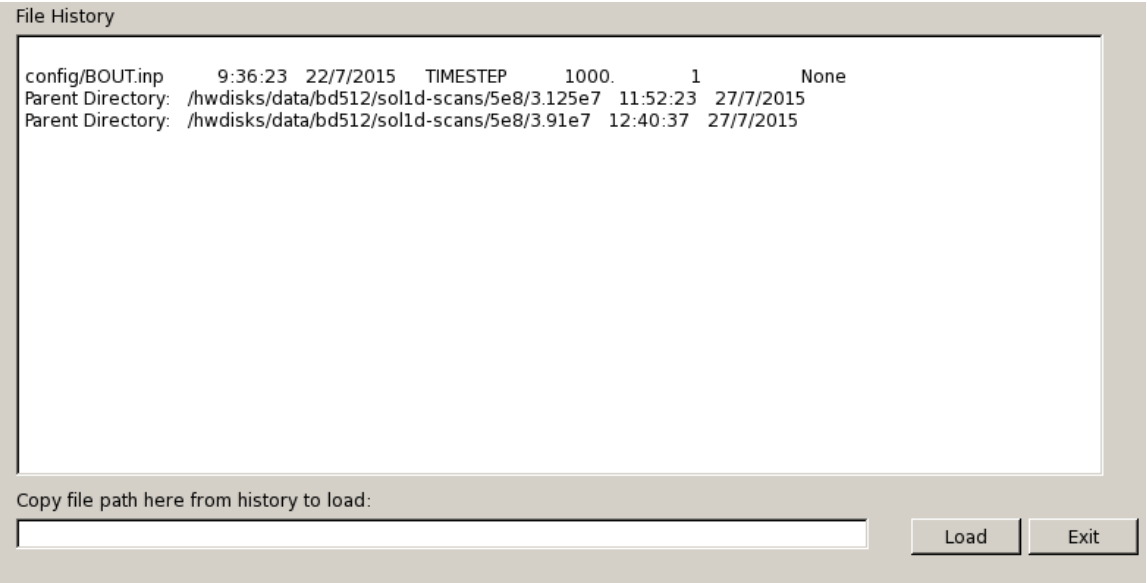


Figure 4 File History example

It is then possible to copy the file path from the parent directory as listed into the text line and click load to load and view/ rerun the old data.

Compare:

Another useful feature is the compare function. Clicking **file -> compare** brings the user back to the initial load tab, with all the same files in it but this time selecting a file brings up a dialog box displaying all the differences between the selected file and the previously loaded files. So if NOUT is reduced to 1500 from 1700 and flux is increased to 1.125e23 we get the following list of changes.

Figure 5 Compare example

Note: the compare will take notice of all little changes to the file regardless of whether they affect the simulation so the additional + in the contents of file 2 can be ignored as addition of white space doesn't effect SOL1D.

The Output Stream Tab:

While a simulation is running this text viewer displays the output from the BOUT simulation. This is identical information to what would have been printed to the screen when running simulations on the console. This information is for interest only to give an idea of how the simulation is progressing generally. It cannot be edited. It is not necessary to keep the output stream tab open while simulations are being run. Other files may be loaded in the background and data analysis of other data may also be undertaken however should the GUI as a whole close then the simulation will crash and the data in the temporary file will not be saved properly.

The stop simulation button acts as a keyboard interrupt so stops the simulation. Ctrl + c can also be used. Currently using a keyboard interrupt deletes the temporary files meaning that the data from that run is lost. This should possibly be changed.

Changing tab while a simulation is running does **not** affect the simulation, neither does loading a new config file and editing it or analysing the data from the run data in that folder (see below).

The Graphing Tab:

Once a simulation has finished it will default to shifting to the graphing tab to enable the user to start analysing the data. It is important to note however that any data set can be loaded and analysed at any time using the load tab. Before any data is analysed it has to be collected by clicking the collect button. An example is shown below in figure 7.

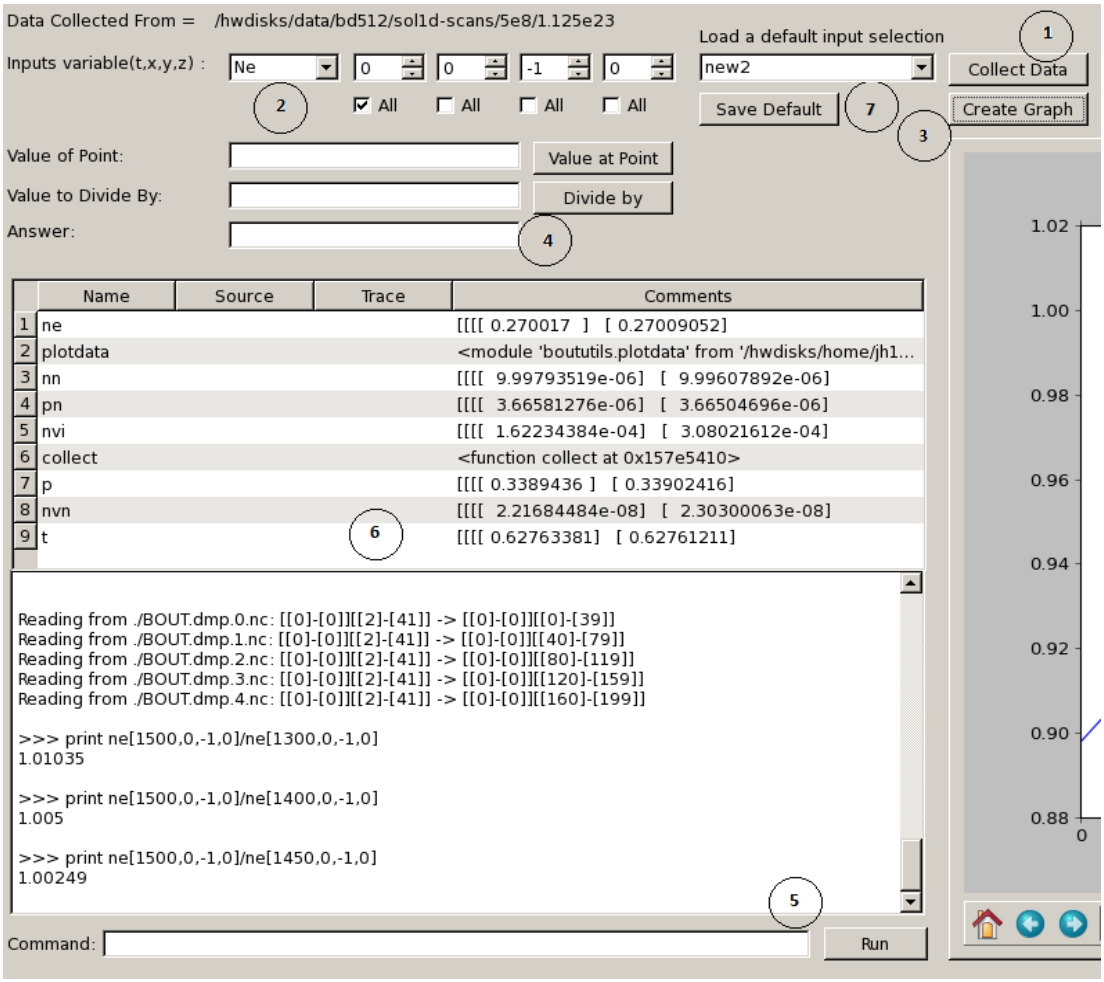


Figure 7 Example of Graphing Tab

1. This is the important collect data button. It must be clicked for the data files that are currently loaded to be analysed. So after a simulation if it is clicked it will 'collect' the data that has just been created from the .dmp files in that folder. If a different folder is loaded in the load tab then the collect path will equal the load path as displayed in the change inputs tab. Collect by default collects all possible variables in the SOL1D model unlike in the command line where each variable has to collected individually.

2. These are user inputs for creating graphs using the graphical side of the GUI. If all of one variable is to be used then the checkbox under that box should be clicked. Here is an example comparing how to plot data in the GUI compared to in the command line.

Figure 6 Example of how to input variables for a graph

   This is equivalent to writing plotdata(ne[:,0,-1,0]).
3. Once the correct variables are selected click 'create graph' to plot a graph in the plotting area to the right.

4. The 'value at point' button and 'divide by' are debatably a little redundant however they can be a quick way to find the value of the data at a specific point. Note that the 'all' boxes are ignored so figure 6 would give the same output as typing ne[0,0,-1,0] into the command line.

5. The graphical input has its uses for quick checks of certain graphs but when analysis becomes more involved and complex it is easier to use this command line like shell. For example it is not possible in the GUI to plot ne[10:73, 0, -1, 0] as the input is either all points or a single point. For these cases the command line is the obvious choice. This command line exhibits all the usual properties of typing into ipython as running simulations in the command line. Modules can be imported, variables assigned etc etc. There is no need to worry about importing collect and plotdata however as this is taken care of by the collect data button. It is possible to hardwire some shorter commands into the command line. Currently only 'plot' is used such that: plot(ne[:,0,-1,0]) == plotdata(ne[:,0,-1,0]). If the user wants to add their own commands they can find the function 'runCommand' in the 'boutGUI.py' file and write the shortcut there as has been done for plot below.

```
def runCommand(self, cmd):
    # Output the command
    self.write(">>> " + cmd)

    glob = globals()
    glob['plot'] = self.DataPlot.plotdata
    # Evaluate the command, catching any exceptions
    # Local scope is set to self.data to allow access to user data
    self.runSandboxed(self._runExec, args=(cmd, glob, self.data))
    self.updateDataTable()
```

   Please not that to get an output to show please use print. So '**print** ne[0,0,-1,0]'.

6. The table here shows the current memory of the python command line i.e. which variables

7. These are default input selections. A default is created by selecting a variable setup that is commonly used and then clicking ‘save default’. A prompt to name the default then appears. Defaults are loaded by being selected from the dropdown box. Defaults help make it really quick to view some common graphs so are useful to help find steady state quickly.

The Graphing Area:

This area to the right command line is where graphs are displayed. They are standard matplotlib graphs so have all the usual attributes such as zoom and save and should be of publishable standard, see matplotlib documentation for more information. Clicking on the green allows the user to edit some of the settings of the graph area and for 1D graphs to add titles and x labels. If a 2D graph is input then it will be plotted with a colour bar to help indicate what each colour means. 3D plots are not possible. This all uses the same plotdata code as was in place for previous plotting in the console through ipython. See examples below.

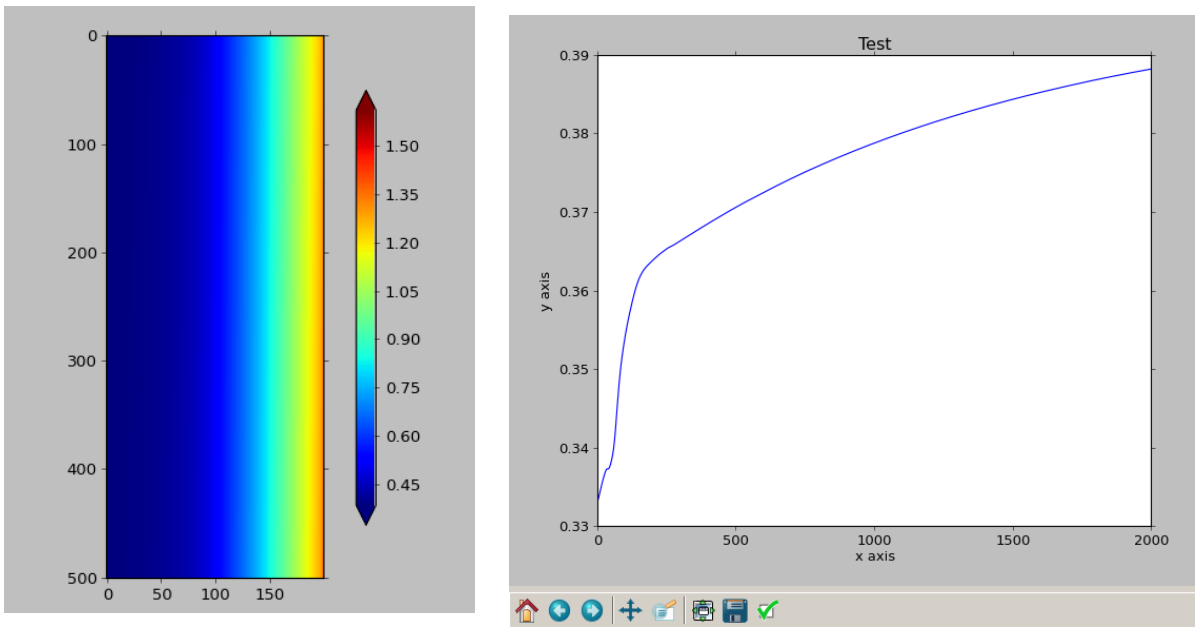


Figure 6 Example of 1d and 2d plot