Running BOUT++ using BOUTgui (20th August 2015):

BOUT++ is a 3D plasma fluid simulation code which has been developed at York in collaboration with the MFE group at LLNL and the MCS division at ANL. Different models are created for different scenarios with different assumptions being made in each. These models are written using this BOUT++ code. For further information see http://www-users.york.ac.uk/~bd512/bout/ . To run a simulation requires a control file (BOUT.inp) containing all the physical and non-physical variables to be linked to the simulation code that describes the specific model, written using BOUT++. Once the simulation is finished the data files and log files are copied to the folder containing the BOUT.inp file. Data then is analysed separately using a python code based on matplotlib called plotdata which comes as part of the BOUT data routines. In the past all of this was run within a linux terminal requiring several tabs to be opened.

The GUI was developed alongside a command line application with the aim to streamline this process, making everything more connected and the code a lot easier to use. It combines data analysis and data creation all within one application. It is has also been designed with data archiving in mind. Information about each run is stored to aid the user when coming back to old runs and remember what changes were made and the history of each data file. This should make long term research easier, particularly if old data is requested for some reason.

Contained within this document -

1) A quick installation guide
2) An introduction for users new to BOUT
3) A tutorial
4) Further information about using the GUI
5) Troubleshooting

1) Installation:

Quick guide:

1) Download the tar archive from GitHub at https://github.com/joe1510/boutGUI .
2) Un-tar into the chosen folder, e.g. tar xvzf BOUTgui.tar.gz /hwdisks/home/username/BOUTgui
3) When running can ignore all files except BOUTgui.py which is the main
4) Run the GUI by using ./BOUTgui.py when in the BOUTgui folder

BOUTgui was written using python 2.7.5. It uses several modules that do not come as standard with python. These include configparser, QtGui, QtCore, and python libraries written as part of the BOUT code (such as plotdata and collect). When running on the University of York system all these modules are already installed so no action is required by the user.

configparser documentation can be found at https://docs.python.org/2/library/configparser.html. BOUT routines are documented in manuals, these are contained within the main BOUT-dev folder created when BOUT is installed. A general path for this being /hwdisks/home/username/BOUT-dev/manuals.

configparser is a module that can read control files (with extensions such as '.inp' and '.ini') and make them readable. One of the things it can do is that by giving a header and an option it can return a value. It is also able to change config files. This is useful for editing the config files used for simulation.

2) Introduction for people unfamiliar with BOUT:

There is a very detailed manual discussing the workings of BOUT and how to use it so little detail of the actually workings will be included here. BOUT++ is based on a C++ framework and was originally developed to analyse boundary turbulence of plasmas within tokamaks. It was designed to separate the user from the numerical methods of running the simulations to instead focus on the actual physics. The manual can be found within the BOUT-dev folder at ./BOUT-dev/manuals/user_manual.pdf and is written for someone wanting to complete runs and analyse the physics of their simulations rather than develop simulation code to model different physical setups. The GUI has been developed with this same philosophy in mind.

The BOUT simulation code, once installed properly, comes with many example models. These are stored in ./BOUT-dev/examples. Each of these folders contains a few files – importantly the code, usually some sort of log file, a makefile as well as a few other files. Before running each of the examples for the first time they need to be compiled. Simply cd into the example folder that is to be run. Then type 'make' and the code will compiled. This creates a new file without an extension. So if sol1d is.cxx is compiled the simulation code that is actually used in simulations is called sol1d. The other files are required to run simulations but can be ignored by the user.

Within each parent folder there is another folder, often called data. This folder contains the control file that contains all the variables required for the simulation code. Every model has different control files. This folder is also where the data files are kept that are created after a simulation has been run.

So the basic principle of running a simulation is as follows. The simulation code, written in BOUT++, contains all the physical laws of the model that is being used. This is generally a series of equations which govern the density, temperature, and pressure (etc etc) of the plasma. The control file contains values for the variables of those equations. The control file is edited by the user so that the simulation model can be tested, either to understand better how the model works and whether the assumptions made in that model are physical. Or, assuming that the model is physical, to better understand what a physical system set up in the way described by the control file would behave in real life.

Once the control file has been fed into the simulation code the code is run. A temporary folder is created to store the data files that are created by the simulation. These consist of log files which have a quick overview of the run, dmp files, which contain all the actual data of what the value of each variable at each point in space and time is, and finally restart files. Restart files are used so that the theoretical system can be restored exactly to its previous state on the next run. This is useful because usually you want to make changes to a system that is already in steady state (i.e. it is more or less the same in one time as it is in the next). Otherwise it would be impossible to tell exactly how changing one variable changes a system. Also it takes a long time to run a system from an initial state to steady state and very large changes in variables cause the simulation code problems anyway.

It should be noted that the simulations can be run on a user specified number of processors. For each processor that is used there is created a restart, dmp and log file. Every time that a simulation is run, whether with a restart or otherwise, all of these files are overwritten. This means that for each folder containing a control file you end up with an entire set of files all used for that run. It is usual to create new folders each time a variable is changed to keep a record of past data. It should be noted that the GUI creates two extra files within the run folder, usernotes.ini and record.ini. These files contain a small amount of data used by the GUI to store extra information about runs. It is the dmp files that had to be 'collected' in ipython for data analysis and graphing.

To actually make a run happen directly in the command line the following set of commands is used:
mpirun –np 5 nice –n 10 ./sol1d –d ./folderpath restart?
This specifies np, the number of processors, in this case 5. The nice level, which is how to prioritise the run on the servers, ten is default. The relative path to the simulation, in this case sol1d. The relative path to the folder containing the BOUT.inp control file if the ./folderpath part. And finally whether to restart the simulation. This argument is left out if a restart is not required.

When using the GUI none of the above should directly apply but the GUI does all of these things behind the scenes so it helps to understand the basic process of running a simulation to understand how to use the GUI in the correct way.

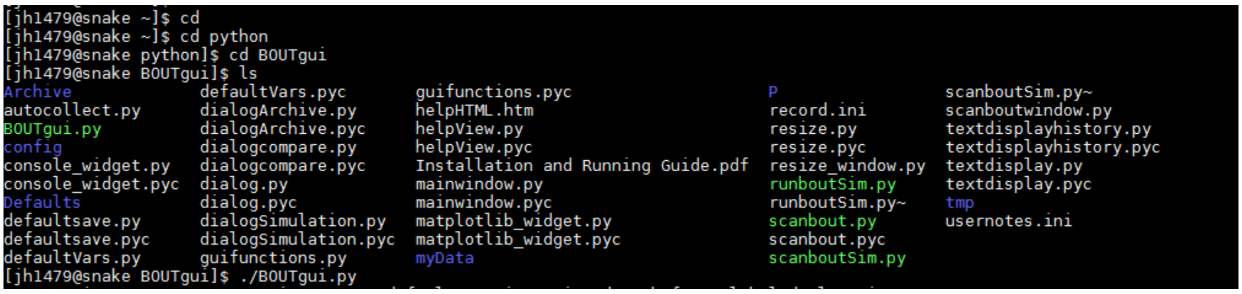
When using the GUI an archive system is used. An archive is specified by clicking on **file->archive location.** Any folders within this archive that contain BOUT.inp files are displayed and in loading one the control file has been specified. Any data/restart files in that folder are used for the next run. Changes can all be made within the change inputs tab. When 'run simulation' is selected there is a prompt to save to a folder of choice. This maybe a new folder or the current folder. Whatever is selected becomes the ./folderpath of the mpirun code. The number of processors and niceness can also be specified here, as well as whether the run should be a restart. All using inputs in the bottom right of the change inputs tab. This leaves only the simulation code to be defined. This is stored within a config file in the BOUTgui folder and the current simulation code is shown at the bottom of the change inputs tab. This can changed by clicking **file -> simulation code.**

How to actually use the features mentioned in this section is described in much more detail below.

3) Tutorial

Starting the GUI

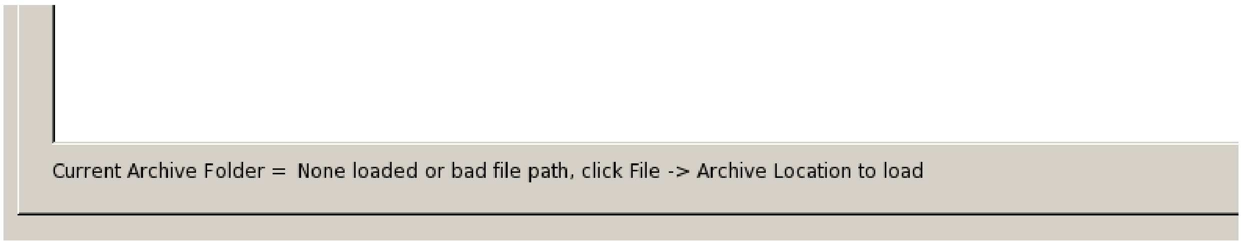
To run the GUI, change directory to the installation folder and then run BOUTgui.py by typing /BOUTgui.py.



```
[jh1479@snake ~]$ cd
[jh1479@snake ~]$ cd python
[jh1479@snake python]$ cd BOUTgui
[jh1479@snake BOUTgui]$ ls
Archive             defaultVars.pyc      guifunctions.pyc                P               scanboutSim.py~
autocollect.py      dialogArchive.py     helpHTML.htm                    record.ini      scanboutwindow.py
BOUTgui.py          dialogArchive.pyc    helpView.py                     resize.py       textdisplayhistory.py
config              dialogcompare.py     helpView.pyc                    resize.pyc      textdisplayhistory.pyc
console_widget.py   dialogcompare.pyc    Installation and Running Guide.pdf  resize_window.py  textdisplay.py
console_widget.pyc  dialog.py            mainwindow.py                   runboutSim.py   textdisplay.pyc
Defaults            dialog.pyc           mainwindow.pyc                  runboutSim.py~  tmp
defaultsave.py      dialogSimulation.py  matplotlib_widget.py            scanbout.py     usernotes.ini
defaultsave.pyc     dialogSimulation.pyc matplotlib_widget.pyc           scanbout.pyc
defaultVars.py      guifunctions.py      myData                          scanboutSim.py
[jh1479@snake BOUTgui]$ ./BOUTgui.py
```

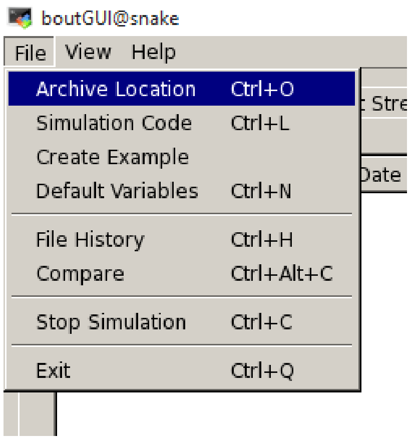
Choosing an Archive

The GUI will have opened with the load tab open. On the first time loading the table showing control files is blank and at the bottom of the screen it says 'None loaded or bad…'

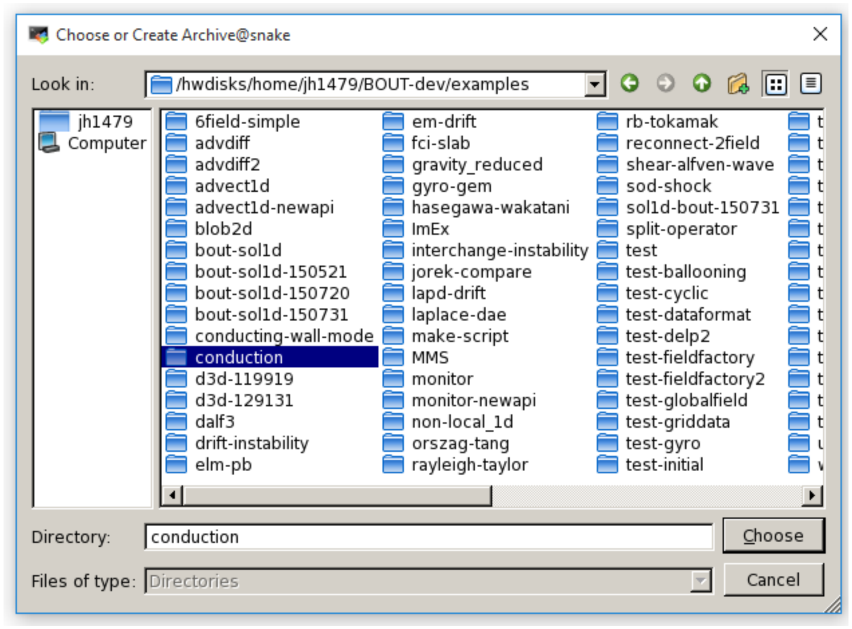


Current Archive Folder = None loaded or bad file path, click File -> Archive Location to load

To load a Current Archive Folder:



This loads **a graphical file explorer allowing you to choose an archive folder**. More details about archiving in section 2 above. The key is to choose a folder which is the parent directory of the folder that contains the control file to run. As an example the 'conduction' model will be used from the BOUT-dev folder. So don't open conduction just highlight and click choose.



Choose or Create Archive@snake

Look in: /hwdisks/home/jh1479/BOUT-dev/examples

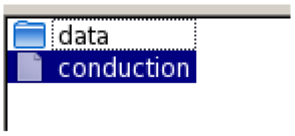
jh1479
Computer

6field-simple	em-drift	rb-tokamak
advdiff	fci-slab	reconnect-2field
advdiff2	gravity_reduced	shear-alfven-wave
advect1d	gyro-gem	sod-shock
advect1d-newapi	hasegawa-wakatani	sol1d-bout-150731
blob2d	ImEx	split-operator
bout-sol1d	interchange-instability	test
bout-sol1d-150521	jorek-compare	test-ballooning
bout-sol1d-150720	lapd-drift	test-cyclic
bout-sol1d-150731	laplace-dae	test-dataformat
conducting-wall-mode	make-script	test-delp2
conduction	MMS	test-fieldfactory
d3d-119919	monitor	test-fieldfactory2
d3d-129131	monitor-newapi	test-globalfield
dalf3	non-local_1d	test-griddata
drift-instability	orszag-tang	test-gyro
elm-pb	rayleigh-taylor	test-initial

Directory: conduction

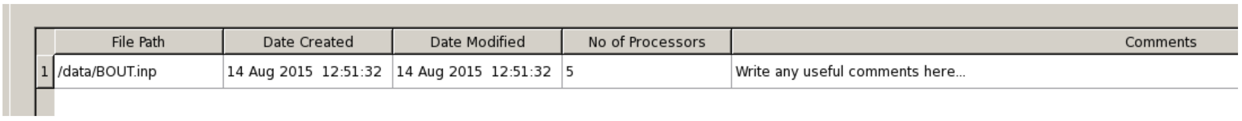
Files of type: Directories

This looks at all folders within the conduction folder for BOUT.inp files. Looking in conduction we see:



Any of the examples that are contained in the ./BOUT-dev/examples folder can be used. They should all have a similar file structure, with a data folder containing a control file (some models it may be names something other than data).

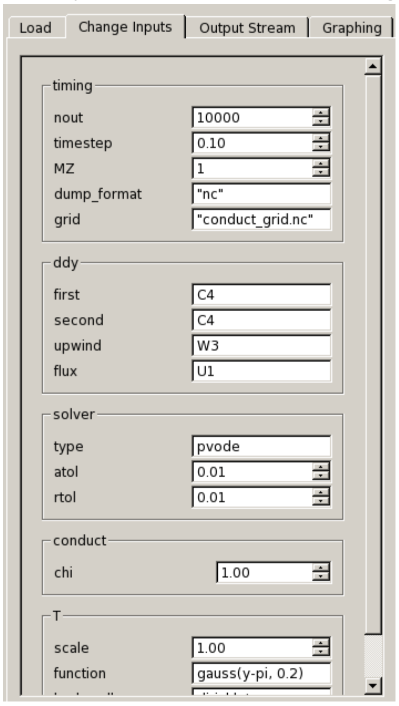
data is the only folder in this case so only this can be searched for BOUT files. It contains an example BOUT.inp folder so is loaded into the table.



	File Path	Date Created	Date Modified	No of Processors	Comments
1	/data/BOUT.inp	14 Aug 2015 12:51:32	14 Aug 2015 12:51:32	5	Write any useful comments here...

Loading and Changing Input Files

Double click on the row in the table that is to be loaded, in this case there is only one row. This takes you to the next tab – change inputs. Below is an excerpt from change inputs is displayed.



timing

nout	10000
timestep	0.10
MZ	1
dump_format	"nc"
grid	"conduct_grid.nc"

ddy

first	C4
second	C4
upwind	W3
flux	U1

solver

type	pvode
atol	0.01
rtol	0.01

conduct

chi	1.00

T

scale	1.00
function	gauss(y-pi, 0.2)

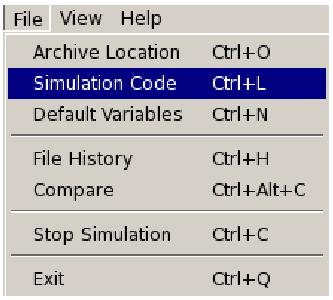
Can highlight row and click load.

The inputs are all in scrollable columns.

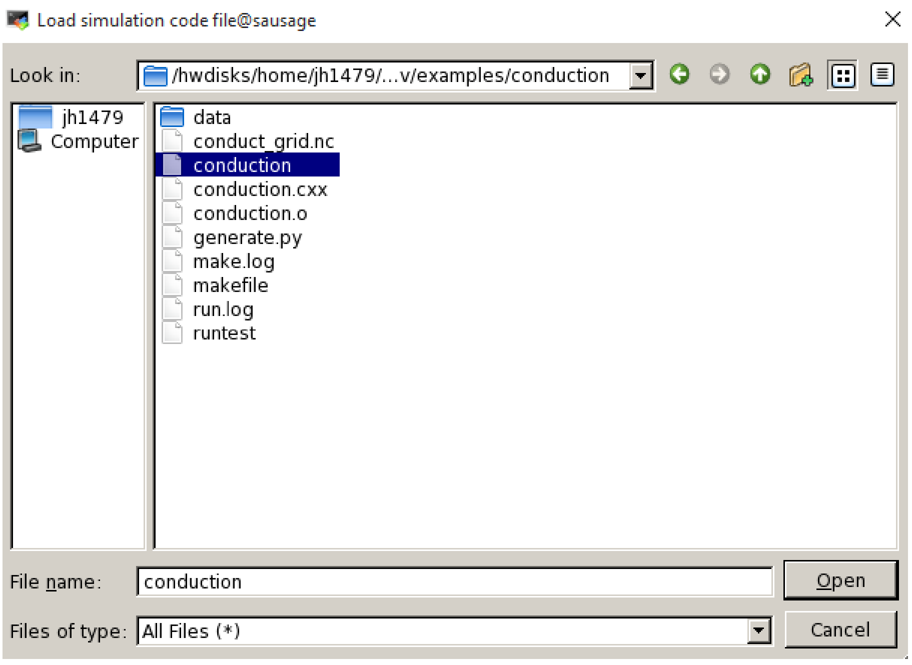
Change the values of any of the files as you wish to run in a simulation. For example increase the timestep to 0.50 by either typing into the box or using the arrows or the mouse wheel. Most variables should display a tool tip when hovering the mouse over them to give extra information.

Choosing Simulation Code

In order to run simulations the file path of the correct simulation code for that model needs to be given by clicking **file -> simulation code,** on first run of the GUI the simulation code file path is 'None'.

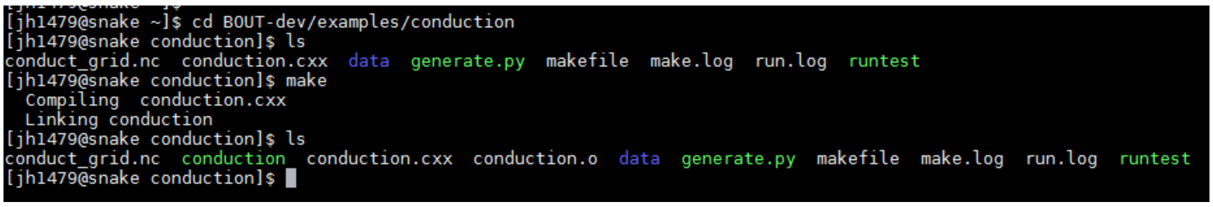


This **brings up a file explorer**. The **control file was for the conduction model so the simulation code needed is 'conduction'**. Check the folder to see if there is a conduction file there. The file path in this case should be /hwdisks/home/username/BOUT-dev/examples/conduction, and the file conduction with no extension.



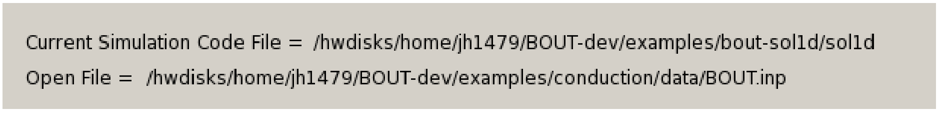
Note there are two files: conduction and conduction.cxx. conduction.cxx is code that has not yet been compiled so should not be used. If conduction (without .cxx) isn't in the folder then the code needs compiling see below.

To compile change directory into the simulation code folder and type 'make'.



```
[jh1479@snake ~]$ cd BOUT-dev/examples/conduction
[jh1479@snake conduction]$ ls
conduct_grid.nc  conduction.cxx  data  generate.py  makefile  make.log  run.log  runtest
[jh1479@snake conduction]$ make
  Compiling  conduction.cxx
  Linking conduction
[jh1479@snake conduction]$ ls
conduct_grid.nc  conduction  conduction.cxx  conduction.o  data  generate.py  makefile  make.log  run.log  runtest
[jh1479@snake conduction]$ 
```

Now the conduction compiled code should be available. The current loaded simulation code is displayed at the bottom of the change inputs tab. Along with the current open file.

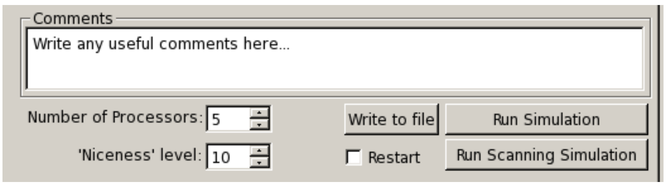


Current Simulation Code File = /hwdisks/home/jh1479/BOUT-dev/examples/bout-sol1d/sol1d
Open File = /hwdisks/home/jh1479/BOUT-dev/examples/conduction/data/BOUT.inp

Simulation code only needs changing on first run of GUI or if the model that is being used is changed.

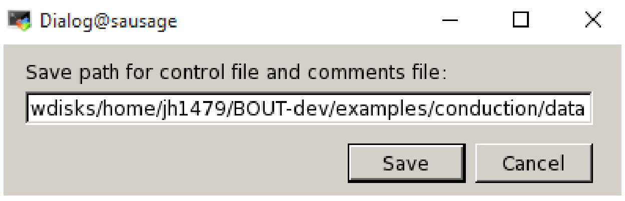
Comments and Run Parameters

Once simulation code has been loaded it is possible to think about runs. All the inputs for this are at the bottom right of the change inputs tab.



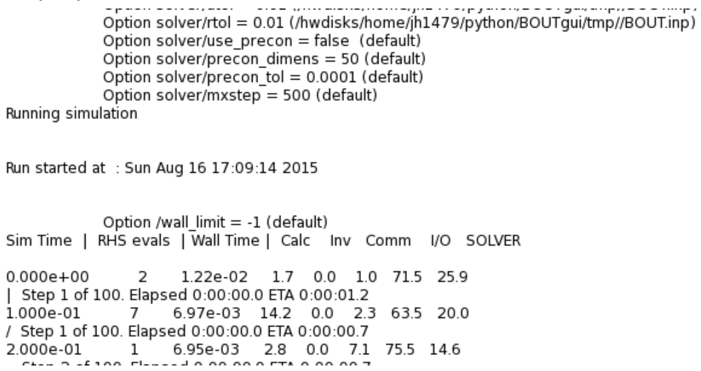
The comments box is for extra information to be stored. Click in and type anything here, deleting the 'Write any useful comments here...' text. Also choose number of processors and nice level. Tick restart box if there are restart files that are to be used, however there are none in a first run so this will be unchecked. See section 2 above if terms such as 'nice' are confusing.

The three buttons are used to determine what to do next with the control file. All three bring up a save dialog box which is used to save the control file with its changes. If a new path is chosen then all data files are also copied across to this new folder. If the same will overwrite.


Dialog@sausage
Save path for control file and comments file:
wdisks/home/jh1479/BOUT-dev/examples/conduction/data
Save
Cancel

That is all that 'Write to File' does, just saves data. The other two bring up the save dialog and then will go into running simulations. 'Run Scanning Simulations' is a little more complicated and is written in greater detail later on. For now 'Run Simulation' will suffice.

Click 'Run Simulation' and once 'Save' has been pressed on the dialog box the GUI moves to the Output Stream tab and displays the output from the BOUT code, this will look something like below. A large amount of information about the run is displayed (can scroll up to see more, for help understanding this refer to BOUT manual) and then a list of how each step is progressing is displayed.



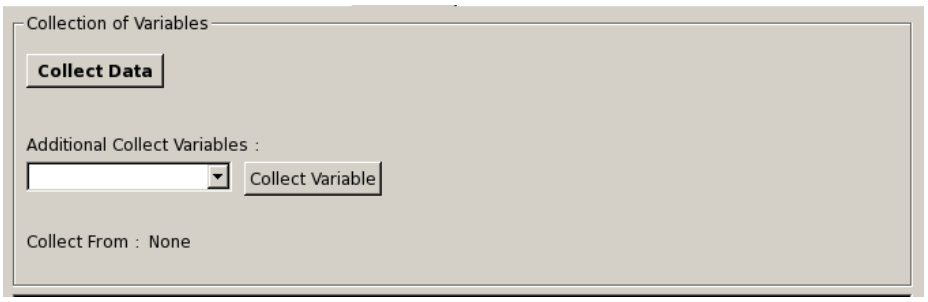
```
                Option solver/rtol = 0.01 (/hwdisks/home/jh1479/python/BOUTgui/tmp//BOUT.inp)
                Option solver/use_precon = false  (default)
                Option solver/precon_dimens = 50 (default)
                Option solver/precon_tol = 0.0001 (default)
                Option solver/mxstep = 500 (default)
Running simulation


Run started at  : Sun Aug 16 17:09:14 2015


                Option /wall_limit = -1 (default)
Sim Time  |  RHS evals  | Wall Time |  Calc    Inv   Comm    I/O   SOLVER

0.000e+00          2      1.22e-02     1.7    0.0    1.0   71.5   25.9
|  Step 1 of 100. Elapsed 0:00:00.0 ETA 0:00:01.2
1.000e-01          7      6.97e-03    14.2    0.0    2.3   63.5   20.0
/  Step 1 of 100. Elapsed 0:00:00.0 ETA 0:00:00.7
2.000e-01          1      6.95e-03     2.8    0.0    7.1   75.5   14.6
```

A simulation can be stopped by clicking 'Stop Simulation' or by using ctrl+c or file -> stop simulation.

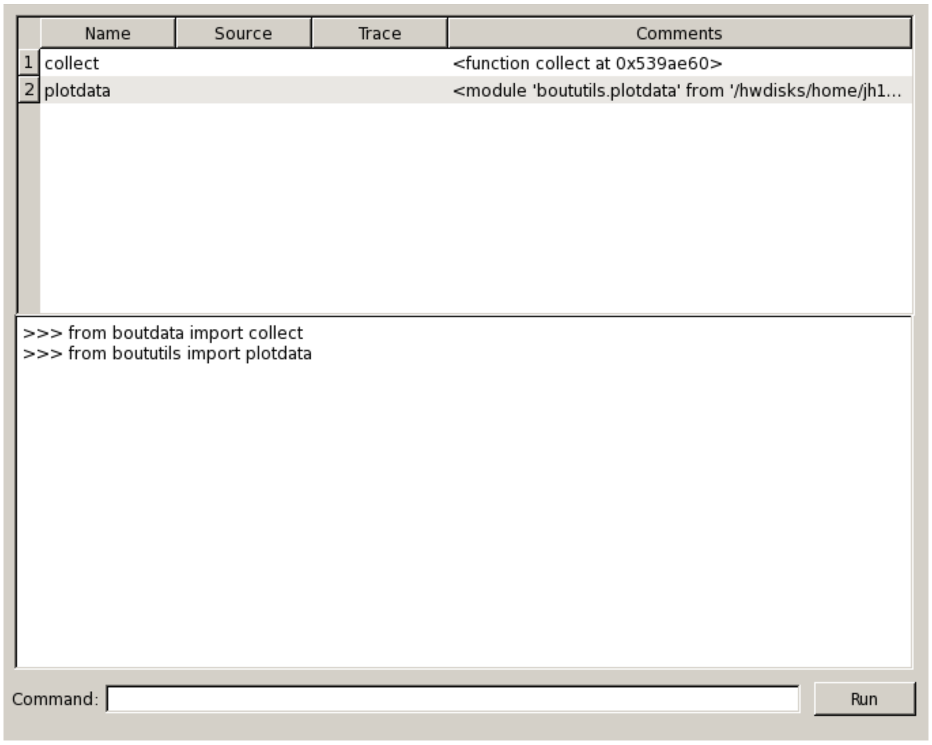
Once finished the GUI moves onto the graphing tab. Please note while a simulation is running it is perfectly OK to load other control files and edit them or analyse the data of them, it does not interrupt the simulation.

<u>Data Analysis – Collection of Data</u>

The first thing to do is to collect data. This imports the data of specific variables from the dmp files. **Click 'Collect Data'**. This runs the automatic data collection routines. All collect related things are in the 'Collection of Variables' box at the top left of the graphing tab.



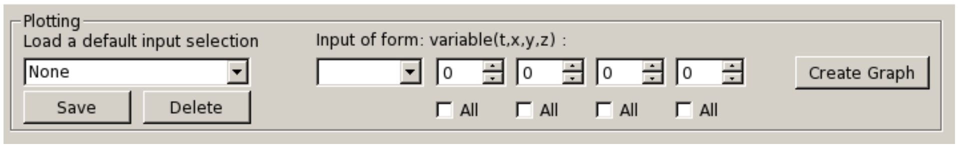
For data analysis it is possible to either use the GUI graphing tools or the command line. The command line behaves just like a python shell. The table above the output display shows the current defined modules and variables, see below.



	Name	Source	Trace	Comments
1	collect			<function collect at 0x539ae60>
2	plotdata			<module 'boututils.plotdata' from '/hwdisks/home/jh1...

```
>>> from boutdata import collect
>>> from boututils import plotdata
```

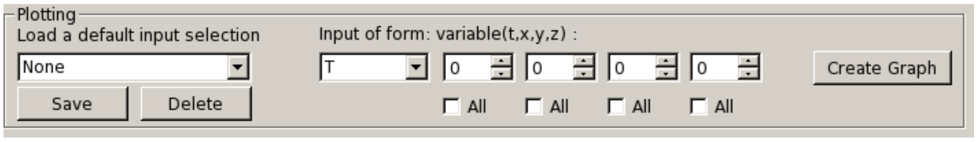
At the top of the screen to the right are the GUI plotting controls.



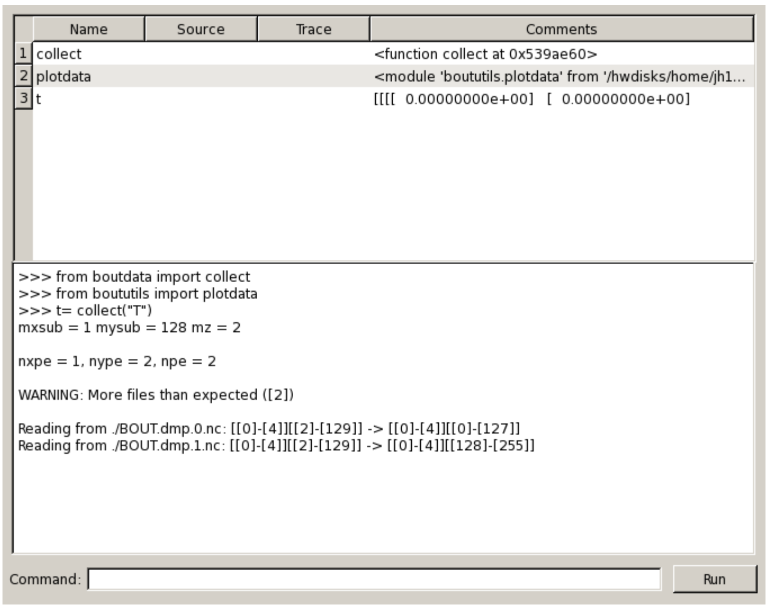
Plotting
Load a default input selection
None
Save
Delete
Input of form: variable(t,x,y,z) :
0
0
0
0
All
All
All
All
Create Graph

In the above screen shots the collect button has been pressed. There is a set of default variables which are collected by the automatic data collection. Any of these which are available are collected automatically. Any collected variables are displayed in the 'inputs variable' combo box or in the data table of the command line. Any variables which can be collected but which aren't on the defaults list are displayed in the 'Additional Collect Variables' combo box on the right and can be manually collected by clicking 'Collect Variable'.

In this case no variables have been collected automatically but T is available. Clicking 'Collect Variable' changes the command line and the GUI controls like this:



Plotting
Load a default input selection
None
Save
Delete
Input of form: variable(t,x,y,z) :
T
0
0
0
0
All
All
All
All
Create Graph

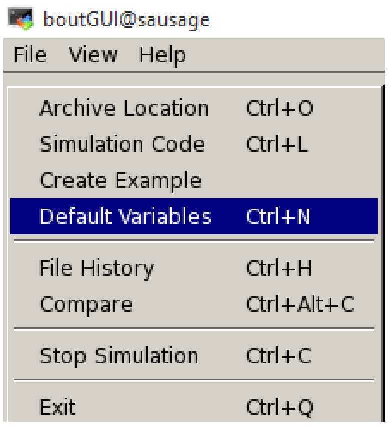


	Name	Source	Trace	Comments
1	collect			<function collect at 0x539ae60>
2	plotdata			<module 'boututils.plotdata' from '/hwdisks/home/jh1...
3	t			[[[0.00000000e+00] [0.00000000e+00]

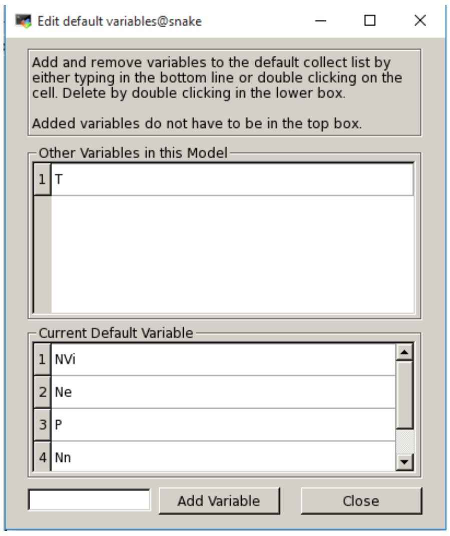
```
>>> from boutdata import collect
>>> from boututils import plotdata
>>> t= collect("T")
mxsub = 1 mysub = 128 mz = 2

nxpe = 1, nype = 2, npe = 2

WARNING: More files than expected ([2])

Reading from ./BOUT.dmp.0.nc: [[0]-[4]][[2]-[129]] -> [[0]-[4]][[0]-[127]]
Reading from ./BOUT.dmp.1.nc: [[0]-[4]][[2]-[129]] -> [[0]-[4]][[128]-[255]]
```

T now exists in the variables table and in the combo box of ‘Inputs variable’.

To add a default to the list of defaults to be collected click **File -> Default Variables**.



This loads a dialog box as shown below. The top box contains all the possible variables available in the current model and the bottom the current defaults. **Type T into the line edit to add it to the list of defaults or double click on the variable in the top box.**



Edit default variables@snake

Add and remove variables to the default collect list by either typing in the bottom line or double clicking on the cell. Delete by double clicking in the lower box.

Added variables do not have to be in the top box.

Other Variables in this Model

1	T

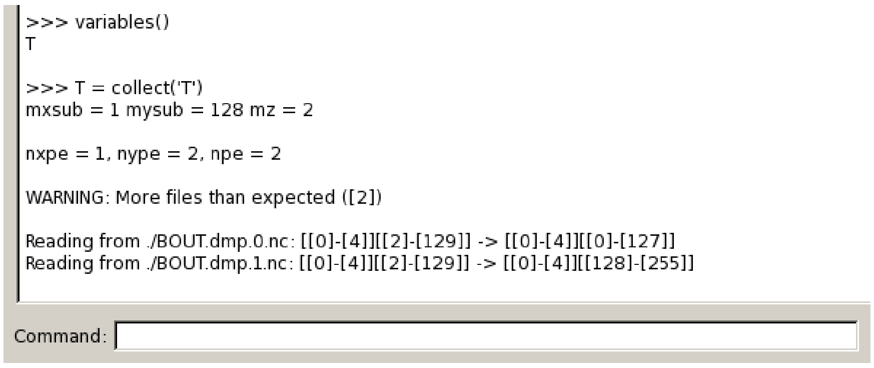
Current Default Variable

1	NVi
2	Ne
3	P
4	Nn

Now that T is in the defaults every time ‘collect’ is clicked the GUI will look to see if T exists in that data set.

Double click a default in the ‘Current Default Variables’ box to delete it.

Data collection can still be done manually, the old way by typing variable = collect('variable') into the command line. For a list of possible variables type variables() into the command line. Note that variables collected this way are only collected for command line plotting and not for GUI plotting.



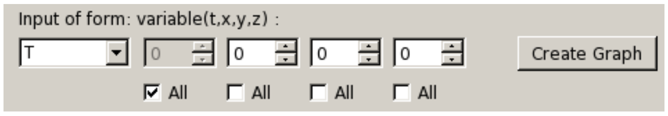
```
>>> variables()
T

>>> T = collect('T')
mxsub = 1 mysub = 128 mz = 2

nxpe = 1, nype = 2, npe = 2

WARNING: More files than expected ([2])

Reading from ./BOUT.dmp.0.nc: [[0]-[4]][[2]-[129]] -> [[0]-[4]][[0]-[127]]
Reading from ./BOUT.dmp.1.nc: [[0]-[4]][[2]-[129]] -> [[0]-[4]][[128]-[255]]
```

Data Analysis- Plotting:

Graph plotting can be achieved either by using the GUI controls at the top of the screen or the command line.

GUI Controls Graphing

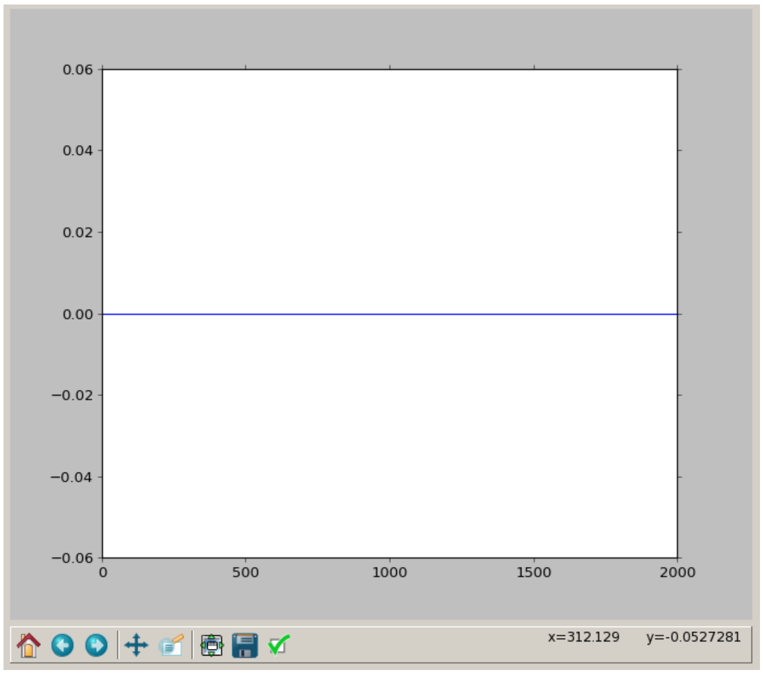
Choose the variable to be plotted in the drop down box, in this model T is the only possibility. Then there are the four boxes, these are the values of the 4-D array (t, x, y, z) to be plotted. Under each is a tick box for all. An example is shown below of the time box being ticked. The 1D graph drawn is of all the values of that T at the point x=0, y=0, z=0 over all time.



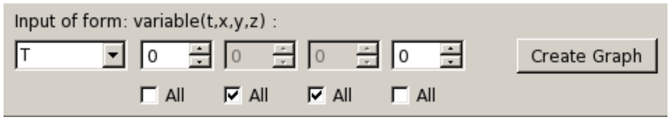
Input of form: variable(t,x,y,z) :

T	0	0	0	0	Create Graph
	✓ All	All	All	All	

The graph looks like this:



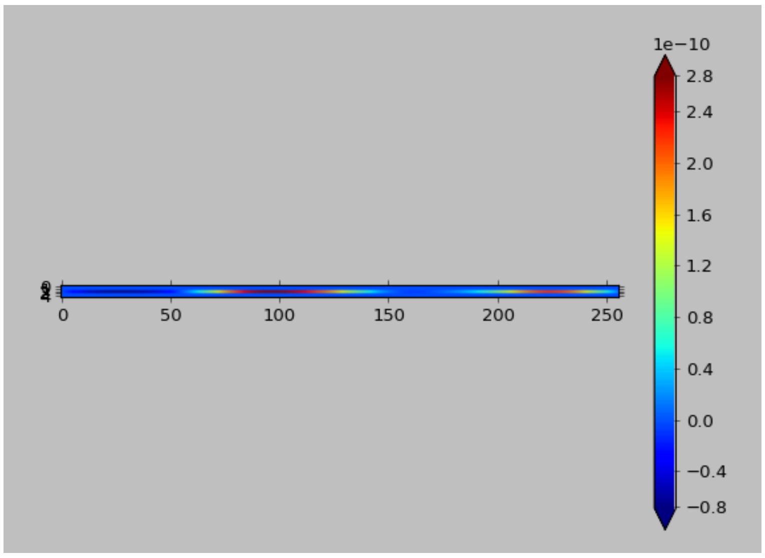
It is also possible to draw 2D graphs by clicking all on two variables. For example:



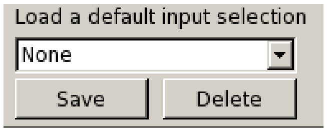
Input of form: variable(t,x,y,z) :

T	0	0	0	0	Create Graph
	All	✓ All	✓ All	All	

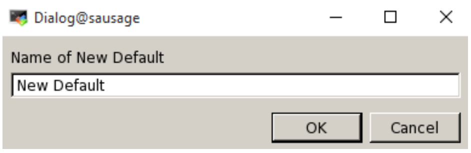
Gives the following two dimensional graph:



The benefit of using the GUI control is that **default plots can be saved**. This is done by clicking on the 'Save Default' button to the right of the graphing controls.



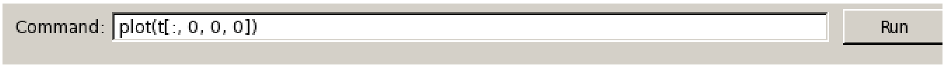
Clicking the button brings up a dialog box asking for the name of the default to save.



To load a default click on the combo box labelled 'Load a default input selection' and chose the name of the default that was previously saved. This should make it quick if the same graphs needs plotting often.

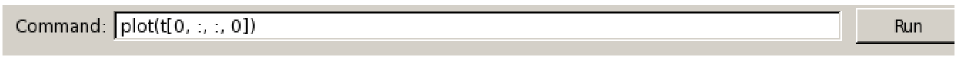
Console Plotting

Console plotting allows more complex tasks to be undertaken. The collected data is stored as arrays, so it is array index notation that is used in the commands. To plot all of a variable a colon is used and sections of data can be done as follows 10:50 would correspond to points between 10 and 50. The command plot is what is used to plot graphs.



```
Command: plot(t[:, 0, 0, 0])
```

Above for the 1D and the below for the 2D case equivalent to the GUI plotting examples.

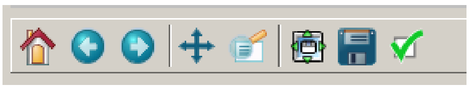


```
Command: plot(t[0, :, :, 0])
```

To look at all the data of a variable type 'print t', or for specific values, 'print t[0,0,0,0]'. **Note that variables are stored as lower case to make it quicker to type.**

Graph Options

The graphs used are imported from matplotlib so detailed information can be found in the relevant documentation. The below buttons at the bottom of the graphs can be used to navigate and zoom on screen.



Also graphs can be saved and the clicking on the tick icon allows axes to be labelled.

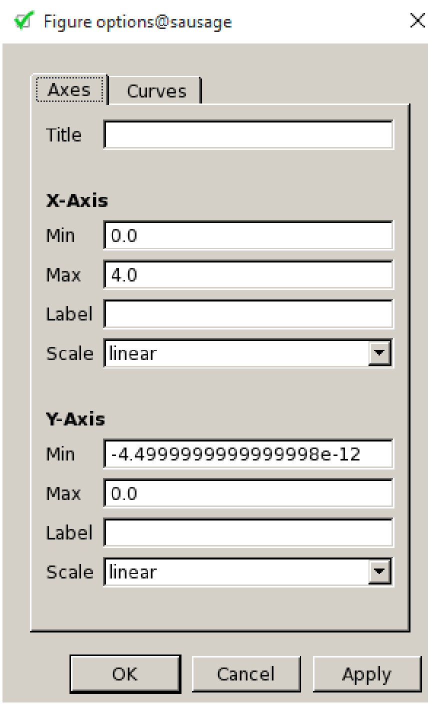


Figure options@sausage

Axes | Curves

Title:

X-Axis
Min: 0.0
Max: 4.0
Label:
Scale: linear

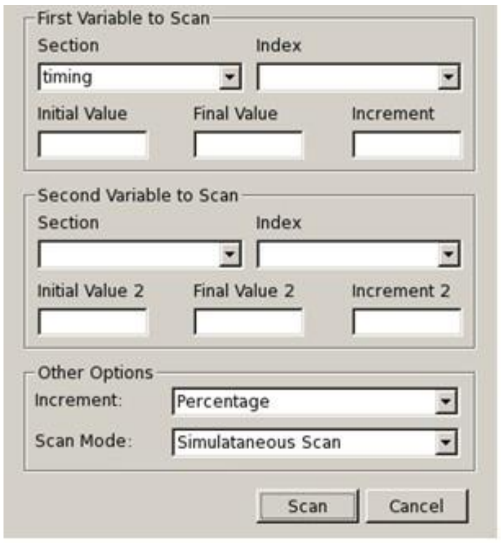
Y-Axis
Min: -4.4999999999999998e-12
Max: 0.0
Label:
Scale: linear

4) Further Information About Using The GUI:

Scanning Simulations

Scanned simulations are designed as a way of automating large numbers of runs where one or two variables are changed by a specified amount each time. Either by a percentage or by a raw amount. There are two different types scanning, full and simultaneous. The details of these are described later.

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



First Variable to Scan
Section: timing
Index:
Initial Value: | Final Value: | Increment:

Second Variable to Scan
Section:
Index:
Initial Value 2: | Final Value 2: | Increment 2:

Other Options
Increment: Percentage
Scan Mode: Simulataneous Scan

There is a choice of scanning either one or two parameters - **if only the top variable box is filled in then only that variable will be scanned, if top and bottom are filled in then both will be scanned.**

Scanning means that the program will start at the initial value adding or multiplying by the specified increment until the limit is reached. 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.

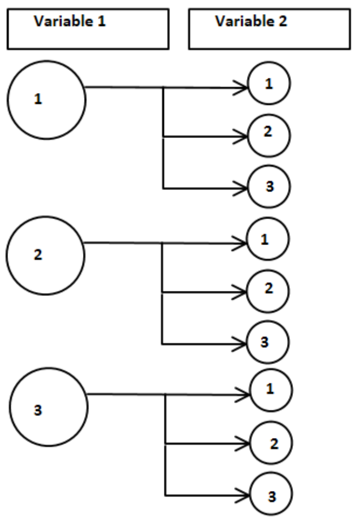
The increment can take two forms – percentage and raw. Raw means that the number input is added on to each run. So if NOUT is run from 50 to 52 with an increment of 1 then each run will be 50, 50+1, 50+1+1 and then stop. Percentage means that the number input is a multiplier. So it is possible to increase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

If only one variable is filled out then scan mode can be ignored as it behaves the same either way.

If two variables are filled out things are a little more complicated. There are two different modes that a double variable scan runs under - simultaneous and full scans.

Simultaneous Scans - these are useful for powering up type simulations where two variables need to either be increased or decreased together with similar or equal magnitude, for example increasing the power of a beam and its flux simultaneously. Two variables are chosen and range and increment size and type specified. The program will then automatically run a series of simulations taking it in turns to increase one variable then the other. For each simulation a new folder is automatically created. The scan only stops when limit 1 is reached so it is important to ensure that limit 2 is set high enough to cover the whole range – if limit 2 is set ridiculously high in this mode this doesn't matter as it will never reach these before limit 1 is reached so in simultaneous scans it is best to set limit 2 several orders of magnitude too high.

Full Scans- these are designed to test how a system behaves over small range of one variable if a second variable is altered. Useful if, for example, testing the effect of reducing or increasing the hyper variable has over a range of densities. Two variables are chosen and range and increment size and type specified. The program will then automatically run a series of simulations, first it runs the case with initial 1 value with no increment. This value is then kept constant while the second variable is increased by the increment 2 over a series of simulations from initial 2 to limit 2. Initial 1 is then increased by increment 1 and variable 2 reset back to initial 2 to run the next series of simulations from initial 2 to limit 2. The size of both limit 1 and limit 2 are important in full scans.



It should be noted that the limit can be lower than the initial value provided that the increment is either a negative if raw or less than one if a percentage. The way the limits work means that program won't necessarily hit that number exactly. For example if a value of 50 is set to increase by 20% for each run but the limit is set to 55 then it will run at 50 then 60 (1.2 x 50) then stop as 60 is higher than the limit, unless the user uses 1.1 then 55 will not be the final result.

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:

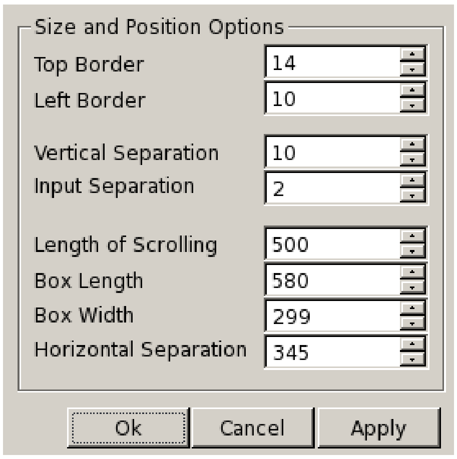
NOUT.500.0.1, NOUT.501.0.1, NOUT.502.0.1

If however these files exist we get instead: NOUT.500.0.2, NOUT.501.0.2, NOUT.502.0.2

Care should be with scanned simulations to ensure enough time steps are used to guarantee steady state is reached as there are currently no inbuilt checks.

Repositioning and Resizing the Inputs

The program automatically creates the inputs of the change inputs tabs depending on the number of options under each heading and the data type of the value of the option in the control file that is loaded. Because it is automatic it cannot be guaranteed that they will be displayed the in the best possible way. If this is the case use **view -> Edit Input Positions** which brings up a dialog box containing lots of different variables which all describes spacing and sizing, the numbers are in pixels.

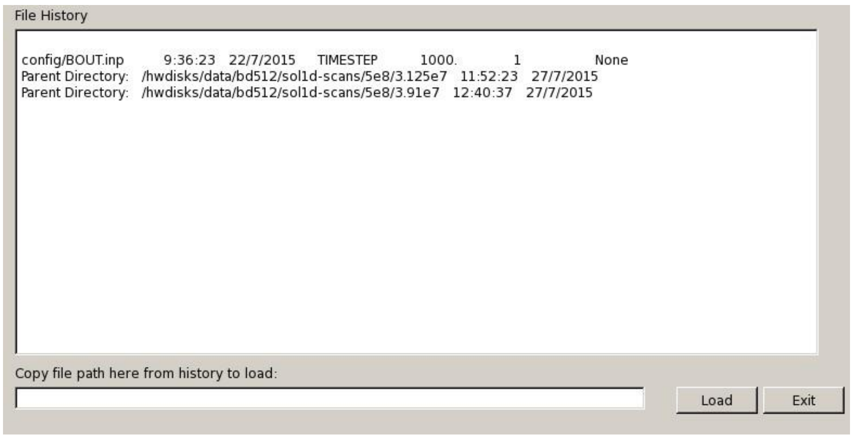


```
Size and Position Options
Top Border              14
Left Border             10

Vertical Separation     10
Input Separation        2

Length of Scrolling     500
Box Length              580
Box Width               299
Horizontal Separation   345
```

If too many columns are created to fit the screen increase <u>length of scrolling</u>

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.

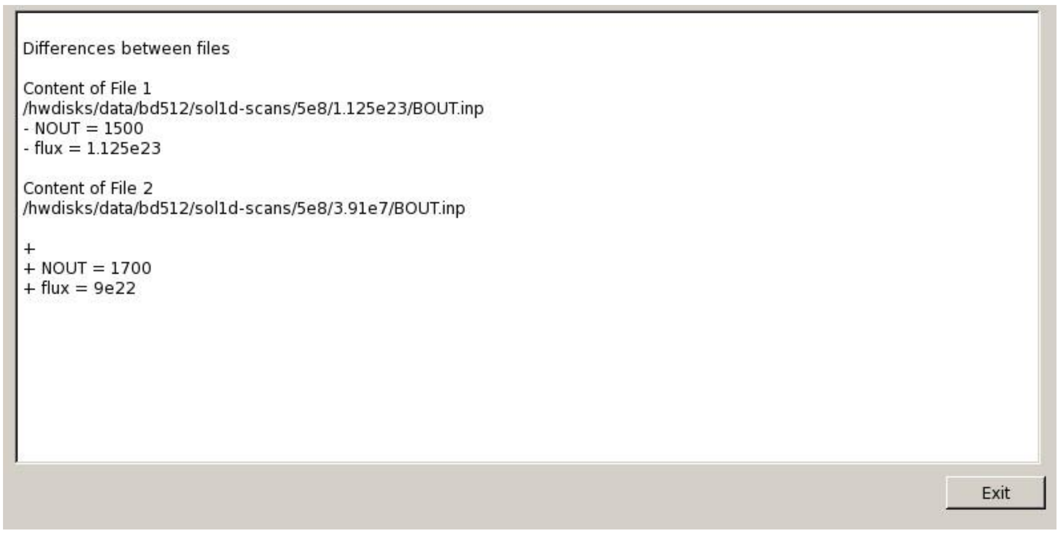


```
File History

config/BOUT.inp       9:36:23   22/7/2015    TIMESTEP       1000.          1               None
Parent Directory:   /hwdisks/data/bd512/sol1d-scans/5e8/3.125e7   11:52:23   27/7/2015
Parent Directory:   /hwdisks/data/bd512/sol1d-scans/5e8/3.91e7   12:40:37   27/7/2015

Copy file path here from history to load:
```

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.



```
Differences between files

Content of File 1
/hwdisks/data/bd512/sol1d-scans/5e8/1.125e23/BOUT.inp
- NOUT = 1500
- flux = 1.125e23

Content of File 2
/hwdisks/data/bd512/sol1d-scans/5e8/3.91e7/BOUT.inp

+
+ NOUT = 1700
+ flux = 9e22
```

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.

Command Line Commands

Commands such as 'variable' and 'plot' in the command line are additional to the previous commands that were available in the BOUT python routines. These have to be defined within the BOUTgui.py file. If more functions are wanted they can be written and the imported into the command line by editing the runCommand function. This requires a line similar to that used for plot to be written.

```
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()
```

This can be found at line 1206 of BOUTgui.

5) Troubleshooting

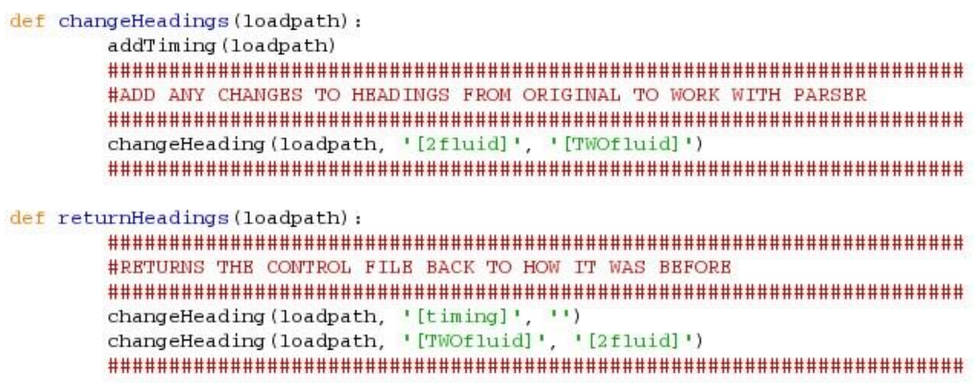
- **Segmentation faults** when trying to run a simulation, or the output prints a few a lines and then moves to graphing tab.

(what is printed)

```
Loading data utilities


No plotpolslice command
Loading BOUT++ data routines
No mlab
No mlab available


No anim


No View3D
```

This could be either:

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 there are any encountered they can easily be automated to be changed also by using the **changeHeadings() and returnHeadings() functions**. These functions can be found in the guifunctions.py file. All that needs to be added is a line like the one to change 2fluid to TWOfluid and a line to reverse that change when running a simulation, see below.



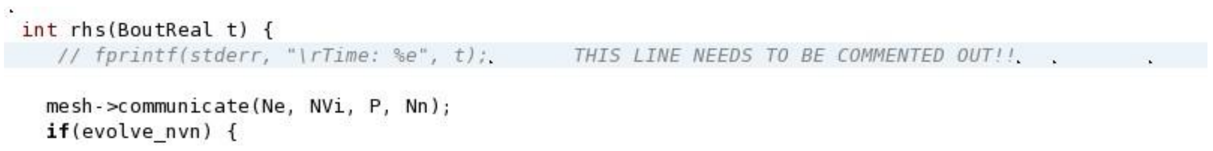
```
def changeHeadings(loadpath):
        addTiming(loadpath)
        ######################################################################
        #ADD ANY CHANGES TO HEADINGS FROM ORIGINAL TO WORK WITH PARSER
        ######################################################################
        changeHeading(loadpath, '[2fluid]', '[TWOfluid]')
        ######################################################################

def returnHeadings(loadpath):
        ######################################################################
        #RETURNS THE CONTROL FILE BACK TO HOW IT WAS BEFORE
        ######################################################################
        changeHeading(loadpath, '[timing]', '')
        changeHeading(loadpath, '[TWOfluid]', '[2fluid]')
        ######################################################################
```

changeHeading() will look for any line containing the first string and replace it with the second. Headings are found within the config file. **They MUST be surrounded by square brackets.**

Hopefully nearly all cases of this have been caught by TWOfluid however.

- If the GUI appears to be working generally fine BUT the actual **simulations are getting stuck either immediately or after a few steps.**

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) {
```

- The inputs freeze up and keyboard input stops working.

When this happens, combo boxes and spin boxes can still be changed using the mouse but not the keyboard. This seems to be a fault with qt applications generally and no easy fix seems possible. **Unfortunately the only way to resolve this is to restart the program.** This doesn't tend to occur until after a simulation has finished so shouldn't cause too much of a problem. If it does happen during a simulation that simulation will continue anyway even if there is the freezing problem.

- When using collect the GUI crashes completely and exits itself.

One problem with the automatic collect routine was that it would send a second call to collect before the previous call had been finished. This would result in a segmentation fault because of bad threading. To get around this a sleep line was added to the code which makes the code wait for 0.1

of a second between each call. *This made the code stable for all runs that were tested, even large data files*. However it may still be possible that if **very** large data files are used then it may take longer to collect from them and cause a crash. Try to increase the time to sleep enter open BOUTgui.py in a text editor and move to workercollect. Change all the calls to sleep to a larger time. Line 350 onwards.

```
# temp was something I needed for SOL1D so it was automated, possibly could be deleted
try:
  if 'P' and 'Ne' in varLst:
      window.commandEntered('temp = 0.5*p/ne')
      sleep(0.1)
except NameError:
    pass

# collects for the GUI graphing
from boutdata import collect
global varDic, plotLst, notLoadedLst
varDic = {}
plotLst = []
notLoadedLst = []


# does the same as before but appends all data to a dictionary of variables
for var in varLst:
        # adds to combo for graphing and collects if var is a default
        if var in defaultLst:
            plotLst.append(var)
            window.variableCombo.addItem(var)
            a = collect(str(var))
            varDic[var] = a

        # again cerates the temp variable which could potentially be removed
        elif 'P' and 'Ne' in varLst and 'temp' not in plotLst:
            try:
                var = 'temp'
                plotLst.append(var)
                window.variableCombo.addItem(var)
                a = collect('P')
                b = collect('Ne')
                varDic[var] = 0.5*a/b
            except NameError:
                pass

        else:
          # all varaibles that aren't defaults are appended to the combobox for optional collection
            window.extraVarsCombo.addItem(var)
            notLoadedLst.append(var)
            self.insertTableRow(var)
            sleep(0.01)
```