Stoner Python Package

C.S. Allen, M. Newman and G. Burnell

January 12, 2011

Contents

1 Introduction

This manual provides a user guide and reference for the Stoner python pacakage. The Stoner python package provides a set of python classes and functions for reading, manipulating and plotting data acquired with the lab equipment in the Condensed Matter Physics Group at the University of Leeds.

1.1 Getting the Stoner Package

The source code for the Stoner python module is kept in CVS revision control on the stonerlab server. A stable release of the code is available for copying and use in `\\stonerlab\data\software\python\stable\`. The development code can be obtained by checking out the PythonCode module with a CVSROOT of `:ext:cvs@stonerlab.leeds.ac.uk:/home/cvs/`. Appropriate ssh keys for the cvs user account are kept in `\\stonerlab\data\software\CVS\`.

The Stoner Package currently depends on a number of other modules. These are installed on the lab machines that have Python installed. Primarily these are Numpy, SciPy and Matplotlib. Windows installable versions are kept in `\\stonerlab\\data\software\Python\`. The easiest way to get a Python installation with all the necessary dependencies for the Stoner Package is to install the *Enthought Python Distribution*. Windows install file are kept in `\\stonerlab\data\software\python`

1.2 Using the Stoner Package

The easiest way to use the Stoner Package is to add the path to the directory containing Stoner.py to your PYTHONPATH environment variable. This can be done on Macs and Linux by doing:

```
cd <path to PythonCode directory>/src
export PYTHONPATH=‘pwd‘:$PYTHONPATH
```

On a windows machine the easiest way is to create a permanent entry to the folder in the system environment variables. Go to Control Panel -¿ System -¿ Advanced Tab -¿ click on Environment button and then add or edit an entry to the system variable PYTHONPATH.

One this has been done, the Stoner module may be loaded from python command line:

```
>>> import Stoner
```

or

```
>>> from Stoner import *
```

2 Users' Guide

The Users'Guide provides a brief overview of the functions contained within the Stoner module and so basic examples of how the module can be used.

The Stoner module provides several Python classes that can be used to manipulate experimental data. The main class that provides the basic functionality is the DataFile class. This handles loading data, finding and manipulating meta data, selecting rows or columns of data, adding or removing data, and saving data.

The PlotFile class is a descendent of DataFile, meaning it shares all the same functionality as DataFile, but in addition has methods to present data graphically. The AnalyseFile class is another descendent of DataFile, but provides extra methods to fit curves, smooth and differentiate data, find peaks and carry out other simple analysis operations.

2.1 Loading a data file

The first step in using the Stoner module is to load some data from a measurement.

```
>>>import Stoner
>>>d=Stoner.DataFile('my_data.txt')
>>>d=Stoner.DataFile('my_VSM_data.fld','VSM')
```

In this example we have loaded data from my_data.txt which should be in the current directory – here we are assuming that my_data.txt contains data in the *TDI Format 1.5* which is produced by the LabVIEW rigs. Assuming that the file successfully loads, *d*, is an instance of the DataFile object. Here the DataFile constructor has been used to both create the instance and load the data in one go. The second version adds an extra parameter that specifying how to interpret the file data. Currently allowed values are *TDI*, *VSM*, *BigBlue*, and *csv*. It is also possible to do these steps separately, or indeed to load new data into an existing instance of DataFile.

```
>>>import Stoner
>>>d=Stoner.DataFile()
>>>d.load('my_data.txt')
>>>d.load('my_VSM_data.fld','VSM')
```

The *BigBlue* version of the DataFile.load and DataFile constructors takes two additional parameters that specify the row on which the column headers will be found and the row on which the data starts. The *csv* takes four additional parameters to the constructor and load methods. In addition to the two extra arguments used for the *BigBlue* variant, a further two parameters specify the deliminators for the header and data rows.

The load method, like many of the DataFile methods returns a copy of the Datafile object ***as well as*** *modifying the object itself. The advantage of this is that it is then possible to chain several methods into one command*

2.2 Examining Some Data

Having loaded some data, the next stage might be to take a look at it. Internally, data is represented as a 2D numpy array of floating point numbers, along with a list of column headers and two dictionaries that hold metadata and type information about metadata (metametadata perhaps !). These can be accessed like so:

```
>>>d.data
>>>d.column_headers
>>>d.metadata
>>>d.typehint
```

This is all very well, but often you want to examine a particular column of data or a particular row:

```
>>>d.column(0)
>>>d.column('Temperature')
>>>d.column(['Temperature',0])
```

In the first example, the first column of numeric data will be returned. In the second example, the column headers will first be checked for one labeled exactly *Temperature* and then if no column is found, the column headers will be searched using *Temperature* as a regular expression. This would then match *Temperature (K)* or *Sample Temperature*. The third example results in a 2 dimensional numpy array containing two columns in the order that they appear in the list (*i.e.*not the order that they are in the data file). For completeness, the **DataFile.column** method also allows one to pass slices to select columns and should do the expected thing.

Rows don't have labels, so are accessed directly by number:

```
>>>d[1]
>>>d[1:4]
```

The second example uses a slice to pull out more than one row. This syntax also supports the full slice syntax which allows one to, for example, decimate the rows, or directly pull out the last fews rows in the file.

What happens if you use a string and not a number in the above examples ?

```
>>>d['User']
```

in this case, it is assumed that you meant the metadata with key *User*. To get a list of possible keys in the metadata, you can do:

```
>>>d.dir()
>>>d.dir('Option\:.*')
```

In the first case, all of the keys will be returned in a list. In the second, only keys matching the pattern will be returned – all keys containing *Option:*.

Sometimes you may want to iterate over all of the rows or columns in a data set. This can be done quite easily:

```
>>>for row in d.rows():
......print row
......
>>>for column in d.columns():
......print column
......
```

The first example could also have been written more compactly as:

```
>>>for row in d:
......print row
......
```

In many cases you do not know which rows in the data file are of interest - in this case you want to search the data.

```
>>>d.search('Temperature',4.2)
>>>d.search('Temperature',4.2,['Temperature','Resistance'])
>>>d.search('Temperature',lambda x,y: x>10 and x<100)
>>>d.search('Temperature',lambda x,y: x>10 and
               x<1000 and y[1]<1000,['Temperature','Resistance'])
```

The general form is
`DataFile.search(<search column>,<search term>[,<list of return columns>])`

The first example will return all the rows where the value of the *Tenperature* column is 4.2. The second example is the same, but only returns the values from the *Temperature*, and *Resistance* columns. The rules for selecting the columns are the same as for the DataFile.column method above – strings are matched against column headers and integers select column by number.

The third and fourth examples above demonstrate the use of a function as the search value. This allows quite complex search criteria to be used. The function passed to the search routine should take two parameters – a floating point number and a numpy array of floating point numbers and should return either *ture* or *False*. The function is evaluated for each row in the data file and is passed the value corresponding to the search column as the first parameter while the second parameter contains all of the values in the row to be returned. If the search function returns True, then the row is returned, otherwise it isn't.

2.3 Modifying Data

2.3.1 Appending data

The simplest way to modify some data might be to append some columns or rows. The Stoner mpodule redefines two standard operators, `+` and `&` to have special meanings:

```
>>>a=Stoner.DataFile('some_new_data.txt')
>>>add_rows=d+a
>>>add_columns=d&a
```

In these example, *a* is a second DataFile object that contains some data. In the first example, a new DataFile object is created where the contents of *a* are added as new rows after the data in *d*. Any metadata that is in *a* and not in *d* are added to the metadata as well. There is a requirement, however, that the column headers of *d* and *a* are the same – *i.e.*that the two DataFile objects appear to represent similar data.

In the second example, the data in *a* is added as new columns after the data from *d*. In this case, there is a requirement that the two DataFile objects have the same number of rows.

These operators are not limited just to DataFile objects, you can also add numpy arrays to the DataFile object to append additional data.

```
>>>import numpy as np
>>>x=np.array([1,2,3])
>>>new_data=d+x
>>>y=np.array([1,2,3],[11,12,13],[21,22,23],[31,32,33]])
>>>new_data=d+y
>>>column=d.column[0]
>>>new_data=d&column
```

In the first example above, we add a single row of data to *d*. This assumes that the number of elements in the array matches the number of columns in the data file. The second example is similar but this time appends a 2 dimensional numpy array to the data. The third example appends a numpy array as a column to *d*. In this case the requirement is that the numpy array has the same or fewer rows of data as *d*.

2.3.2 Inserting Columns of Data

The append columns operator `&` will only add columns to the end of a dataset. If you want to add a column of data in the middle of the data set then you should use the **add_column** method.

```
>>>d.add_column(numpy.array(range(100)),'Column Header')
>>>d.add_column(numpy.array(range(100)),'Column Header',Index)
>>>d.add_column(lambda x: x[0]-x[1],'Column Header',func_args=None)
```

The first example simply adds a column of data to the end of the dataset and sets the new column headers. The second variant inserts the new column before column *Index*. *Index* follows the same rules as for the **DataFile.columnn()** method. In the third example, the new column data is generated by applying the specified function. The function is passed s dingle row as a 1D numpy array and any of the keyword, argument pairs passed in a dictionary to the optional *func_args* argument.

The **DataFile.add_column** method returns a copy of the DataFile object itself as well as modifying the object. This is to allow the metod to be chained up with other methods for more compact code writing.

2.3.3 Deleting Rows of Data

Removing complete rows of data is achieved using the **DataFile.del_row** method.

```
>>>d.del_rows(10)
>>>d.del_rows('X Col',value)
>>>d.del_rows('X Col',lambda x,y:x>300)
```

The first variant will delete row 10 from the data set (where the first row will be row 0). You can also supply a list or slice to **DataFile.del_rows** to delete multiple rows.

If you do not know in advance which row to delete, then the second and third variants provide more advanced options. The second variant searches for and deletes all rows in which the specified column contains *value*. The third variant selects which ros to delete by calling a user supplied function for each row. The user supplied function is the same in form and definitition as that used for the **DataFile.search** method.

2.3.4 Deleting Columns of Data

Deleting whole columns of data can be done by referring to a column by index or column header - the indexing rules are the same as used for the **DataFile.column** method.

```
>>>d.del_column('Temperature')
>>>d.del_column(1)
```

2.4 Saving Data

Only saving data in the *TDI* format is supported.

```
>>>d.save()
>>>d.save(filename)
```

In the first case, the filename used tosave the data is determined from the filename attribute of the DataFile object. This will have been set when the filewas loaded from disc.

If the filename attribute has not been set *e.g.*if the DataFile object was created from scratch, then the DataFile.save method will fail. This will be changed to cause a dialog box to be raised.

In the second variant, the supplied filename is used and the filename attribute is changed to match this *i.e.*`d.filename` will always return the last filename used for a load or save operation.

2.5 Plotting Data

2.6 Curve Fitting