Stoner Python Package

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

December 19, 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.VSMFile('my_VSM_data.fld')
```

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 example shows the use of one of the sub-classes of the DataFile object to load data from a specific instrument (in this case the VSM).

This is an API change from earlier versions of the Stoner package where a second parameter on the constructor of the DataFile object was used to identify the type of data file. This syntax is now depreciated !).

The possible sub-classes are:

DataFile Tagged Data Interchange Format 1.5 – the default format produced by the LabVIEW measurement rigs

VSMFile The text files produced by the group's Oxford Instruments VSM

BigBlueFile Datafiles produced by VB Code running on Big Blue. 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.

CSVFile Reads a generic comma separated value file. The **CSVFile** load routine 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 data and header rows. **CSVFile** also offers a **save** method to allow data to be saved in a simple deliminated text way (see Section 2.4 for details).

XRDFile Loads a scan file produced by Arkengarthdale - the group's Brucker XRD Machine.

SPCFile Loads a Raman scan file (.spc format) produced by the Rensihaw and Horiba Raman spectrometers. This may also work for other instruments that produce spc files, but has not been extensively tested.

```
>>>import Stoner
>>>d=Stoner.DataFile()
>>>d.load('my_data.txt')
>>> v=Stoner.VSMFIle()
>>>v.load('my_VSM_data.fld')
>>> c=Stoner.CSVFile()
>>> c.load('data.csv',1,0,',',',')
```

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

2.2.1 Data, Column headers and metadata

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 a dictionary that keeps the metadata and also keeps track of the expected type of the metadata (*i.e.*the meta-metadata). These can be accessed like so:

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

2.2.2 Working with columns of data

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.

There is a convenient shortcut for working with cases where the column headers are not the same as the names of any of the attributes of the **DataFile** object:

```
>>>d.Temperature
>>>d.column('Temperature')
```

both return the same data.

2.2.3 Working with complete rows of data

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.

2.2.4 Manipulating the metadata

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:*. For compatibility with normal opython semantics: `>>>d.keys()` is synonymous with `>>>d.dir()`.

We mentioned above that the metadata also keeps a note of the expected type of the data. You can get at the metadata type for a particular key like this:

```
>>>d.metadata.type('User')
```

to get a dictionary of all of the types associated with each key you could do:

```
>>>dict(zip(d.dir(),d.metadata.type(d.dir())))
```

but an easier way would be to use the **typeHintedDict.types** attribute:

```
>>>d.metadata.types
```

.

2.2.5 More on Indexing the data

There are a number o other forms of indexing supported for **DataFile** objects.

```
>>>d[10,0]
>>>d[0:10,0]
>>>d[10,'Temp']
>>>d[0:10,['Voltage','Temp']
```

The first variant just returns the data in the 11th row, first column (remember indexing starts at 0). The second variant returns the first 10 values in the first column. The third variant demonstrates that columns can be indexed by string as well as number, and the last variant demonstrates indexing multiplerows and columns – in this case the first 10 values of the Voltage and Temp columns.

You might think of the data as being a list of records, where each column is a field in the record. Numpy supports this type of structured record view of data and the **DataFile** object provides the *DataFile.records* attribute to d this. This read-only attribute is just providing an alternative view of the same data.

```
>>> d.records
```

2.2.6 Selecting Individual rows and columns of data

Many of the function in the Stoner module index columns by searching the column headings. If one wishes to find the numeric index of a column then the **DataFile.find_col** method can be used:

```
>>> index=d.find_col(1)
>>> index=d.find_col('Temperature')
>>> index=d.find_col('Temp.*')
>>> index=d.find_col(1:10:2)
>>> index=d.find_col(['Temperature',2,'Resistance'])
```

DataFile.find_col takes a number of different forms. If the argument is an integer then it returns (trivially) the same integer, a string argument is first checked to see if it exactly matches one of the column headers in which case the number of the matching column heading is returned. If no exact match is found then a regular expression search is carried out on the column headings. In both cases, only the first match is returned. The final two examples given above both return a list of indices, firstly using a slice construct - in this case the result is trivially the same as the slice itself, and in the last example by passing a list of column headers to look for.

This is the function that is used internally by **DataFile.column**, **DataFile.search** *etc*and for this reason the trivial integer and slice forms are implemented to allow these other functions to work with multiple columns.

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>[,<listof 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 a list of 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. In thr last example, the final parameter can either be a list of columns or a single column. The rules for indexing columns are the same as used for the **DataFile.find_col** method.

2.2.7 Find out more about the data

Another question you might want to ask is, what are all the unique values of data in a given column (or set of columns). The Python numpy package has a function to do this and we have a direct pass through from the DataFile object for this:

```
>>> d.unique('Temp')
>>> d.unique(column,return_index=False, return_inverse=False)
```

The two optional keywords cause the numpy routine to return the indices of the unique and all non-unique values in the array. The column is specified in the same way as the **DataFile.column** method does.

2.2.8 Copying Data

One of the characterisitics of Python that can confuse those used to other programming languages is that assignments and argument passing is by reference and not by value. This can lead to unexcted results as you can end up modifying variables you were not expecting ! To help with creating genuine copies of data Python provides the copy module. Whilst this works with DataFile objects, for convenience, the **DataFile.clone** atribute is provided to make a deep copy of a DataFile object.

This is an attribute not a method, so there are no brackets here !

```
>>> t=d.clone
```

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 Working with Columns of Data

2.3.3 Rearranging Columns of Data

Sometimes it is useful to rearrange columns of data. **DataFile** offers a couple of methods to help with this.

```
>>> d.swap_column(('Resistance','Temperature'))
>>> d.swap_column(('Resistance','Temperature'),headers_too=False)
>>> d.swap_column([(0,1),('Temp','Volt'),(2,'Curr')])
>>> d.reorder([1,3,'Volt','Temp'])
>>> d.reorder([1,3,'Volt','Temp'],header_too=False)
```

The **swap** method takes either a tuple of column names/indices or a list of such tuples and swaps the columns accordingly, whilst the **reorder** method takes a list of column labels/indices and constructs a new data matrix out of those columns in the new order. The *headers_too=False* options, as the name suggests, cause the column headers not be rearranged.

2.3.4 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.columnmn()** 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.

2.3.5 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.

2.3.6 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)
```

2.3.7 Sorting Data

Data can be sorted by one or more columns, specifying the columns as a number or string for single columns or a list or tuple of strings or numbers for multiple columns. Currently only ascending sorts are supported.

```
>>>d.sort('Temp')
>>>d.sort(['Temp','Gate'])
```

2.4 Saving Data

Only saving data in the *TDI* format and as comma or tab deliminated formats is supported.

The CSVFile comma or tab deliminated files discard all metadata about the measurement. You absolutely must not use this as your primary data format – always keep the *TDI* format files as well.

```
>>>d.save()
>>>d.save(filename)
>>>d=Stoner.CSVFile(d)
>>>d.save()
>>>d.save(filename,'\t')
```

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 cause a dialog box to be raised so that the user can supply a filename.

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.

The third is similar but convert the file to *cvs* format while the fourth also specifies that the deliminator is a tab character.

2.5 Plotting Data

Data plotting and visualisation is handled by the PlotFile sub-class of DataFile. The purpose of the methods detailed here is to provide quick and convenient ways to plot data rather than providing publication ready figures.

```
>>> import Stoner.Plot as plot
>>> p=plot.PlotFile(d)
```

The first line imports the **Stoner.Plot** module. Strictly, this is unnecessary as the Plot module's namespace is imported when the Stoner package as a whole is imported. The second line creates an instance of the **PlotFile** class. PlotFile inherits the constructor method of **DataFile** and so all the variations detailed above work with PlotFile. In particular, the form shown in the second line is a easy way to convert a DataFile instance to a PlotFile instance for plotting.

2.5.1 Plotting 2D data

x-*y* plots are produced by the **PlotFile.plot_xy** method:

```
>>> p.plot_xy(column_x, column_y)
>>> p.plot_xy(column_x, [y1,y2])
>>> p.plot_xy(x,y,'ro')
>>> p.plot_xy(x,[y1,y2],['ro','b-'])
>>> p.plot_xy(x,y,title='My Plot')
>>> p.plot_xy(x,y,figure=2)
>>> p.plot_xy(x,y,plotter=pyplot.semilogy)
```

The examples above demonstrate several use cases of the **plot_xy** method. The first parameter is always the x column that contains the data, the second is the y-data either as a single column or list of columns. The third parameter is the style of the plot (lines, points, colours *etc*) and can either be a list if the y-column data is a list or a single string. Finally additional parameters can be given to specify a title and to control which figure is used for the plot. All matplotlib keyword parameters can be specified as additional

keyword arguments and are passed through to the relevant plotting function. The final example illustrates a convenient way to produce log-linear and log-log plots. By default, **plotxy** uses the **pyplot.plot** function to produce linear scaler plots. In principle a log scale plot could be produced by calculating a new column of data, but a more convenient way is to use the **pyplot.semilogx**, **.semilogy** and **.loglog** functions. These can be passed to the **plot_xy** function by setting the *plotter* keyword argument.

The X and Y axis label will be set from the column headers.

2.5.2 Plotting 3D Data

A number of the measurement rigs will produce data in the form of rows of x, y, z values. Often it is desirable to plot these on a surface plot or 3D plot. The **PlotFile.plot_xyz** method can be used for this.

```
>>> p.plot_xyz(col_x,col_y,col_z)
>>> p.plot_xyz(col_x,col_y,col_z,cmap=matplotlib.cm.jet)
>>> p.plot)xyz(col-x,col-y,col-z,plotter=pyplot.pcolor)
```

By default the plot_xyz will produce a 3D surface plot with the z-axis coded with a rainbow colourmap (specifically, the matplotlib provided *matplotlib.cm.jet* colourmap. This can be overriden with the *cmap* keyword parameter. If a simple 2D surface plot is required, then the *plotter* parameter should be set to a suitable function such as **pyplot.pcolor**.

Like **plot_xy**, a *figure* parameter can be used to control the figure being used and any additional keywords are passed through to the plotting function. The axes labels are set from the corresponding column labels.

Alternatively, if your data is already in the form of a matrix, you can use the **PlotFile.plot_matrix** method:

```
>>> p.plot_matrix()
>>> p.plot_matrix(xvals,yvals,rectang,title="Title",xlabel="X Axis",
            ylabel="Y Axis",zlabel="Z Axis",cmap=matplotlib.cm.jet)
>>> p.plot_matrix(plotter=pyplot.pcolor,figure=False)
```

The first example just uses all the default values, in which case the matrix is assumed to run from the 2nd column in the file to the last and over all of the rows. The x values for each row are found from the contents of the first column, and the y values for each column are found from the column headers interpreted as a floating pint number. The colourmap defaults to the built in 'jet' theme. The x axis label is set to be the column header for the first column, the y axis label is set either from the meta data item "ylabel" or to "Y Data". Likewise the z axis label is set from the corresponding metadata item or defaults to "Z Data". In the second form these parameters are all set explicitly. The *xvals* parameter can be either a column index (integer or sring) or a list, tuple or numpy array. The *yvals* parameter can be either an row number (integer) or list,tuple or numpy array. Other parameters (including *plotter*, *figure etc*) work as for the **PlotFile.plot_xyz** method. The *rectang* parameter is used to select only part of the data array to use as the matrix. It may be 2-tuple in which case it specifies just the origin as (row,column) or a 4-tuple in which case the third and forth elements are the number of rows and columns to include. If *xvals* or *yvals* specify particular column or rows then the origin of the matrix is moved to be one column further over and one

row further down (*i.e.*the matrix is to the right and below the columns and rows used to generate the x and y data values). The final example illustrates how to generate a new 2D surface plot in a new window using default matrix setup.

2.5.3 Getting More Control on the Figure

It is useful to be able to get access to the matplotlib figure that is used for each **PlotFle** instance. The **PlotFile.fig** attribute can do this, thus allowing plots from multiple **PlotFile** instances to be combined in a single figure.

```
>>> p1.plot_xy(0,1,'r-')
>>> p2.plot_xy(0,1,'bo',figure=p1.fig)
```

Likewise the **PlotFile.axes** attribute returns the current axes object of the current figure in use by the **PlotFile** instance.

There's a couple of extra methods that just pass through to the pyplot equivalents:

```
>>> p.draw()
>>> p.show()
```

2.6 Curve Fitting

Curve fitting is handled by a sub-class of the DataFile object – AnalyseFile

```
>>> import Stoner.Analysis as Analysis
>>> a=Analysis.AnalyseFile('Data')
>>>  a2=Analysis.AnalyseFile()
>>> a2=d
```

The first line imports the AnaylseFile class. Since the AnalyseFile is a child class of DataFile, everything you can do with a DataFile also works with an AnalyseFile object. The last two lines demonstrate creating a blank AnalyseFile and then copying all of the data, metadata and column headings from an existing dataFile object.

2.6.1 Simply polynomial Fits

Simple least squares fitting of polynomial functions is handled by the **AnalyseFile.polyfit** method:

```
>>> a.polyfit(column_x,column_y,polynomial_order, bounds=lambda x, y:True)
```

This is a simple pass through to the numpy routine of the same name. The x and y columns are specified in the first two arguments using the usual index rules for the Stoner package. The routine will fit multiple columns if *column_y* is a list or slice. The polynomial_order parameter should be a simple integer greater or equal to 1 to define the degree of polynomial to fit. The bounds function follows the same rules as the bounds function in **DataFile.search** to restrict the fitting to a limited range of rows. The mthod returns a list of co-efficients with the highest power first. If *column_y* was a list, then a 2D array of co-efficients is returned.

2.6.2 Simple function fitting

For more general curve fitting operations the **AnalyseFile.cruve_fit** method can be employed. Again, this is a pass through to the numpy routine of the same name.

```
>>> a.curve_fit(func,  xcol, ycol, p0=None, sigma=None,
                                bounds=lambda x, y: True )
```

The first parameter is the fitting function. This should have prototype `y=func(x,p[0],p[1],p[2]...)` where p is a list of fitting parameters. The *p0* parameter contains the initial guesses at the fitting parameters, the default value is 1. *xcol* and *ycol* are the x and y columns to fit. This method cannot handle multiple y columns. *sigma*, if present, provides the weightings for each datapoint and so should also be an array of the same length as the x and y data. Fianlly, the bounds function can be used to restrict the fitting to only a subset of the rows of data.

AnalyseFile.curve_fit returns a list of two arrays `[popt,pcov]` where *popt* is an array of the optimal fitting parameters and *pcov* is a 2D array of the co-variances between the parameters.

2.6.3 Fitting with limits

For cases where one requires more flexibility in fitting data, in particular where the fitting parameters are constrained, the **AnalyzeFile.mpfit** method is provided. This is a pass through to the **mpfit** module.

```
>>> a.mpfit(func,  xcol, ycol, p_info,  func_args=dict(), sigma=None,
              bounds=lambda x, y: True, **mpfit_kargs )
```

In this case, the *func* argument takes a slightly different prototype:
`def func(x,parameters, **func_args)` where *parameters* is a list of the fitting parameters and *func_args* provides a dictionary of fixed *i.e.*non-fitting parameters. *xcol* and *ycol* are the column indices for the x and y data, *bounds* is a bounding function to select only those rows to use for fitting the function, and *sigma* are the weightings for each datapoint. The remaining arguments are a dictionary of keywords to pass through to the **mpfit** routine and *p_info* which is a list of dictionaries which is used to control the parameters in the fit. This described below.

p_info contains one element for each parameter used to fit the data. Each element is a dictionary with the following keys:

value the starting parameter value (but see the START_PARAMS parameter for more information).

fixed a boolean value, whether the parameter is to be held fixed or not. Fixed parameters are not varied by MPFIT, but are passed on to MYFUNCT for evaluation.

limited a two-element boolean array. If the first/second element is set, then the parameter is bounded on the lower/upper side. A parameter can be bounded on both sides. Both LIMITED and LIMITS must be given together.

limits a two-element float array. Gives the parameter limits on the lower and upper sides, respectively. Zero, one or two of these values can be set, depending on the values of LIMITED. Both LIMITED and LIMITS must be given together.

parname a string, giving the name of the parameter. The fitting code of MPFIT does not use this tag in any way. However, the default iterfunct will print the parameter name if available.

step the step size to be used in calculating the numerical derivatives. If set to zero, then the step size is computed automatically. Ignored when AUTODERIVATIVE=0.

mpside the sidedness of the finite difference when computing numerical derivatives. This field can take four values:

0 one-sided derivative computed automatically

1 one-sided derivative $(f(x+h) - f(x))/h$

-1 one-sided derivative $(f(x) - f(x-h))/h$

2 two-sided derivative $(f(x+h) - f(x-h))/(2*h)$

Where H is the STEP parameter described above. The "automatic" one-sided derivative method will chose a direction for the finite difference which does not violate any constraints. The other methods do not perform this check. The two-sided method is in principle more precise, but requires twice as many function evaluations. Default: 0.

mpmaxstep the maximum change to be made in the parameter value. During the fitting process, the parameter will never be changed by more than this value in one iteration.
A value of 0 indicates no maximum. Default: 0.

tied a string expression which "ties" the parameter to other free or fixed parameters. Any expression involving constants and the parameter array P are permitted.Example: if parameter 2 is always to be twice parameter 1 then use the following: `parinfo(2).tied = '2 * p(1)'`. Since they are totally constrained, tied parameters are considered to be fixed; no errors are computed for them.[NOTE: the PARNAME can't be used in expressions.]

mpprint if set to 1, then the default iterfunct will print the parameter value. If set to 0, the parameter value will not be printed. This tag can be used to selectively print only a few parameter values out of many.
Default: 1 (all parameters printed)

2.7 More AnalyseFile Functions

2.7.1 Basic Data Inspection

```
>>> a.max(column)
>>> a.min(column)
```

2.7.2 Thresholding and Interpolating Data

```
>>> a.threshold(col, threshold, rising=True, falling=False)

>>> a.interpolate(newX,kind='linear' )
```

2.7.3 Smoothing and Differentiating Data

2.7.4 Peak Finding

3 Cookbook

This section gives some short examples to give an idea of things that can be done with the Stoner python module in just a few lines.

3.1 Extract X-Y(Z) from X-Y-Z data

In a number of measurement systems the data is returned as 3 parameters X, Y and Z and one wishes to extract X-Y as a function of constant Z. For example, $I-V$ sweeps as a function of gate voltage V_G. Assuming we have a data file with columns *Current*, *Voltage*,*Gate*:

```
>>> d=DataFile('data.txt')
>>> t=d
>>> for gate in d.unique('Gate'):
>>>     t.data=d.search('Gate',gate)
>>>     t.save('Data Gate='+str(gate)+'.txt')
```

The first line opens the data file containing the $I-V(V_G)$ data. The second creates a temporary copy of the DataFile object - ensuring that we get a copy of all metadata and column headers. The **for** loop iterates over all unique values of the data in the gate column and then inside the for loop, searches for the corresponding $I-V$ data, sets it as the data of the temporary DataFile and then saves it.

3.2 Mapping X-Y-Z data to Z(X,Y) data

In a similar fashion to the previous section, where data has been recorded with fixed values of X and Y *e.g.*I measured for fixed V and V_G, it can be useful to map the data to a matrix.

```
>>> d=DataFile('Data,.txt')
>>> t=d
>>> for gate in d.unique('Gate'):
>>>    t=t+d.search('Gate',gate)[:,d.find_col('Current')]
>>> t.column_headers=['Bias='+str(x) for x in d.unique('Voltage')]
>>> t.add_column(d.unique('Gate'),'Gate Voltage',0)
```

The start of the script follows the previous section, however this time in the for loop the addition operator is used to add a single row to the temporary DataFile *t*. In this case we are using the utility method **DataFile.find_col** to find the index of the column with the current data. After the for loop we set the column headers in *t* and then insert an additional column at the start with the gate voltage values.

The matrix generated by this code is suitable for feeding directly into **PlotFile.plot_matrix()**, however, the same plot could be generated directly from the **PlotFile.plot_xyz()** method too.