The SM Tutorial

by Robert Lupton and Patricia Monger

1 What is SM?

SM is an interactive plotting package for drawing graphs. It does have some capability to handle image data, but mostly works with vectors. The main features of the package are that one can generate a nice looking plot with a minimum number of simple commands, that one can view the plot on the screen and then with a very simple set of commands send the same plot to a hardcopy device, that one can build and save ones own plot subroutines to be invoked with a single user-defined command, that the program keeps a history of ones plot commands, which can be edited and defined as a plot subroutine, to be reused, and that one can define the data to be plotted from within the program, or read it from a simple file.

You might ask, "Why do I need SM?", but I am not going to answer you. If you read through this tutorial, and use the package for a while, and still can't see why you need it, then you probably don't need it.

2 How should I get data into SM?

Plot vectors may be generated in several ways:

a. You may read the vectors from a file using the `read` command The file is expected to be an ASCII file of columns of numbers (separated by spaces, tabs, or commas). You define the file to SM using the `data` command, and associate a column or row of numbers with a SM vector using the `read` command. Example: Say I have a file named test.dat with the following data in it:

```
1       2       3       5.6     10
3       6       8       2.3     11
5       8       2       7.7     12
7       9       4       9.3     13
9       3       1       4.8     14
```

Then the commands to issue to SM to get the data into the program are:

```
data test.dat
read x 1
read y 2
```

(or `read { x 1 y 2 }`). In the last 2 commands I have told SM to read the values in column 1 of the file test.dat, and assign them to a vector named x, and read the values in column 2 of the file and assign them to a vector named y. I could read any of the other columns in as well, of course, and assign them to vectors. And I can name the vectors whatever I like, as long as the name consists of the characters a-z,A-Z,0-9, and _ (underscore). I can also read a row from the file, instead of a column, by saying

```
read row x 1
```

Note that the vector is defined by the read command. But I can redefine it whenever I wish, and change the size. The only point to remember is that when you redefine the vector, the old values are overwritten.

A final point to note about defining vectors from files is that you can skip over lines in the file with the `lines` command. `lines` defines which lines in the file you want to read. A limitation of `lines` is that you may only define one set of lines to read; that is, if you had a 30 line file, and wanted to read lines 3-9 and 15-30, you couldn't (well, you could, but you'd have to make clever use of the method of defining vectors which is discussed in the next subsection, or make lines 10-14 each begin with a #).

b. You may define the vectors within SM using the `set` command. This command has a number of forms:

- If you just want to define the vector with a list of values, the command is

```
set numlist = { 2 3 4 5 6 7 8 9 27 }
```

- you can also define a vector in terms of arithmetic operations on a previously defined vector. For example, having defined numlist as above,

```
set ylist = sqrt(numlist) + numlist/3.1
```

the allowed arithmetic operators are `sin, cos, tan, acos, asin, atan, atan2, abs, int, lg, exp, ln, sqrt, concat, **, +, -, *,` and `/`, where `lg` is $\log_{10}$, `ln` is $\log_e$, `int` takes the integer part, and `concat` concatenates two vectors.

– you can define vectors with implied do-loops:

```
set x = 0,PI,PI/16
set y = 0,10,0.1
```

(PI is a constant defined for you by SM, but you can also define your own scalar variables, as will be described in a later section of this tutorial)

c. You can redefine an *existing* vector element by element with a `do` loop:

```
set y = 1,50
...
do i=0,49 { set y[$i] = $i**2 }
```

(*Note Well* that vector elements are numbered starting from *0*)

d. You may create a vector with the `spline` command. This fits a spline function to a previously defined pair of vectors, and evaluates it at the points given in a third vector, to produce a fourth vector for you.

```
set x = 0, 2 * PI, PI/4
set y = sin(x)
set xx = 0, 2 * PI, PI/32
spline x y xx yy
```

This will fit a spline to the curve y vs x, at the points 0, PI/32, PI/16, 3 * PI/32,... (i.e. the points in the xx vector), and the spline values will be stored in the vector yy.

e. You may define a vector with the graphics cursor using the `cursor` command. If you type the command

```
cursor a b
```

then a cursor will be displayed on the screen, and to the `spline` command. This will take a horizontal slice through the image. If you do not give a filename, the vectors are printed to the terminal.

f. If you want vectors that look properly scientific to play with, you might find that

```
set x=random(100)
```

is just what you want.

3 How do I generate a basic plot for data in a file?

The procedures listed above describe how to get your data in to SM. Then the steps to plotting it are as follows:

- Define the plot device you want to use with the `device` command. The syntax and list of available devices are described in the manual (see section "Device" in *The SM Manual*).
- Declare the data file with the `data` command, see Chapter 2 [Reading Data], page 2, or you could see section "Data" in *The SM Manual*. You might want to use the macro `da` instead, which stops SM worrying about things like `$` or `/` in the filename.
- Read in the data vectors with the `read` command, also described previously and in the manual.
- Define the axis limits with the `limits` command.
- Draw the axes with the `box` command
- Plot the data points using the `connect` command to plot the points as a connected curve, or the `points` command to plot them as points. If you are using `connect` you can also define the line type with the `ltype` command (see section "Ltype" in *The SM Manual*). If you want to plot points, you should first define the point style with the `ptype` command. `ptype` allows you to define the point as a type of polygon, with any number of sides, and 4 basic forms (see section "Ptype" in *The SM Manual*). Ptype also allows you to define your own private symbols, or use strings that you've read from a file.
- If you want to label the axes, use the `xlabel` and `ylabel` commands. SM is able to plot Greek letters, superscripts and subscripts, many sorts of symbols, and a couple of other special fonts. The available fonts are listed in the manual (see section "Fonts" in *The SM Manual*).

For example, you might type:

```
device x11
echo reading test.dat
data test.dat
read { x 1 y 2 z 4 }
limits x y
box
connect x y
ptype 6 3
points x z
xlabel This is the X axis
ylable This is data plotted against X
```

There are a number of other commands that elaborate on this basic set to allow logarithmic axes, labelling curves, putting an ID on the plot, reading positions from the plot with a cursor, manipulating 2-D data, and much more. Some of these commands are described in this tutorial (see Chapter 10 [Common Commands], page 15); the definitive source, however, is in the real manual, where *all* the commands plus their syntax are described.

4 What do I have to do to start up SM?

With any luck, your system manager has set up SM so that you can run it by typing a single command. She should have created a system wide file called an "`.sm`' file'. Just in case she was negligent, or in case you want to overrule some of her choices, you can have in addition your own '`.sm`' in your home directory. This file is used by SM to tell it a number of things. A prototype `.sm` file for a VMS system is reproduced below:

```
device          hirez
edit            $disk:[sm_dir]maps.dat
filecap         $disk:[sm_dir]filecap
file_type       vms_var
fonts           $disk:[sm_dir]fonts.bin
help            $disk:[sm_dir.help]
history         80
history_file    .smhist
graphcap        $disk:[sm_dir]graphcap
macro           $disk:[sm_dir.macro]
macro2          $disk:[lupton.sm]
name            my_friend
printer         qms
temp_dir        sys$scratch:
term            hirez
termcap         $disk:[sm_dir]termcap
```

Now, what is all this?

`device` allows SM to initialize a default plotting device for you. If it finds an entry of this type in your `.sm` file, it will do the call to the `device` command for you.

`edit` is used by SM to find out how to map the key sequences used by the macro and command line editor. The only reason you need to change this line is if you decide you want to define the key sequences differently. For example, if you invoke the macro editor (See Chapter 6 [Macros], page 9 to find out what that means), and want to edit your macro and, say, insert a line, to do that you type `CONTROL-M` (hold down the CTRL key and then press the m key, or simply hit `CARRIAGE RETURN`). Now, if you don't like that particular choice of keys, you can set up your own key definitions in your own edit file, and tell SM to use that instead of the default ones by redefining the `edit` line of the `.sm` file.

`filecap` is a file that tells SM how to read 2-dimensional files. As mentioned in section 11, binary files vary enormously from operating system to operating system, and also depend on the language of the program you wrote to generate them, so we defined a few simple file formats you can use to read binary data into SM, and SM interprets them via the `filecap` file. Read the manual if you want to plot 2-D data.

`file_type`
: is also for 2-D data. This sets the default file type for the binary files. The types are described in the manual (see section "Filecap" in *The SM Manual*). The `filecap` file tells SM how to read data of the given `file_type`

`fonts`
: This tells SM where to find the font definition file. You will almost certainly never change this, but if you have made a new font file you would cause SM to use it instead of the one we supply by changing this line in your `.sm` file.

`help`
: This tells SM where to find the command help files.

`history`
: The number of history commands to remember (see the next entry).

`history_file`
: SM keeps of history of the commands you used in your SM session in a file. It reads in the last history file when you start it up again, and you can reuse those commands as you wish (e.g. scroll through them like with the VMS command line editor, extract a group of commands into a macro (see Chapter 6 [Macros], page 9), ...). If you don't want a history, leave this line blank. Otherwise, specify a filename.

`graphcap`
: This is the file SM uses to figure out what magic commands to send to your plot device to cause it to go into graphics mode. We have defined many device types, so hopefully the one you need is already in the default file. If not, you may want a private graphcap.

`macro`
: SM loads a set of default plot macros for you when you start it up. This line gives the location of the default macro file.

`macro2`
: You can load 2 default files if you wish, and this is where you define the second one.

`macro2`
: is the name of a directory where SM expects to find a file name '`default`', in which are contained SM macros. You should load our default one first, for reasons which are explained in the manual. The macro `startup2` in file '`default`' will be executed.

`name`
: This is the name by which SM will address you when you use it.

`printer`
: There is a macro called '`hcopy`' that replays the commands used to generate a plot on your screen and changes the device to a printer to allow you to easily get a hardcopy of your screen plot. This line tells SM what printer you want to use. You can also get hardcopy plots manually (see Chapter 8 [Hardcopy], page 12).

`temp_dir`
: Hardcopy plots are written to a disk file, and then submitted to a print queue and deleted. This tells SM what directory you want it to write the disk files to. They can be large, so if you have disk quota problems, `temp_dir` ought to point to a scratch disk or something

`term`
: SM knows about terminals, and uses that knowledge to allow you to do command line editing. Here is where you specify what kind of terminal you have. Note this is for '`text`' only; the graphics description is given in the `device` line at the start of this file. The available terminal types are described in the `termcap` entry.

`termcap` This file describes terminals.

So to run SM, you should have a file like this in your home directory, with the directory names, etc changed to point to your computer and you, and then just run the program. If all goes well, when you invoke the program, you will wait a while, and then get the following message

```
Hello, <name>, please give me a command
```

where `<name>` is as defined in the `.sm` file `name` line, and you will get a prompt. If this isn't what happens, you need to contact the people who installed SM on your system.

5 How do I define variables, and how can I use them?

Scalar variables are defined with the `define` command. As mentioned above, vectors are defined with the `set` command. A variable may be a number, or a character string. You may use them in any SM command, by preceding the name of the variable with a $. For example:

```
define two_pi 6.283        # or define two_pi $(2*pi)
set x=1,100
do i = 0, 0.99, .01 {
   set x[100*$i] = $i * $two_pi
}
set i=0, 0.99 , .01
set x=$two_pi*i
set y = sin(x)
limits x y
box
define xlab {my signal}
xlabel $xlab
ylabel sine
```

6 What is a plot macro, and how do I make one?

A plot macro is a set of commands that you can execute together by invoking the name of the macro; in effect, it is a plot subroutine. For example, suppose you had a set of plots that you wanted to generate, using the same type of axis box and labels. Rather than laboriously typing the box and label and limits commands for each set of data, you could define a macro as follows:

```
drawbox         # this is a comment
                limits 0 20 0 100
                box
                xlabel xdata
                ylabel ydata
```

where the macro name in this example is `drawbox`. Then, when you access your data, you could do as follows:

```
        data file1.dat
        read { x 1 y 2 }
        drawbox
        connect x y
erase
        data file2.dat
        read x 3
        read y 7
        drawbox
        connect x y
```

(The `read { x 1 y 2 }` is the same as `read x 1 read y 2`, but faster). This is a simple-minded example, and you can immediately see ways to improve the macro I have created to save even more typing. Macros may consist of any SM commands, and may have arguments. You specify the number of arguments in the macro definition, and refer to them by number, preceded by $. In the example I gave above, suppose we wanted to make the axis labels into variables. Then the macro definition would look like this:

```
drawbox 2       # this is also a comment, but not a very useful one
                limits 0 20 0 100
                box
                xlabel $1
                ylabel $2
```

Then to invoke the macro, I type

```
drawbox xdata ydata
```

You can make a macro in 4 ways:

a. You can create it with your favorite editor outside of SM. The rule to remember if you do this is that the name of the macro must be the first thing on a line of the file, and should be followed by SM commands. All the commands must start in a column in

the file other than the first column. To read this macro into SM, use the `macro read` command

```
macro read macro.file
```

will read all the macros in the file `macro.file`.

b. You can define the macro within SM by extracting a set of commands from the history buffer

```
macro mname 1 20
```

will extract lines 1 through 20 from the history buffer, and create the macro `mname` which consists of those 20 lines.

c. You can define the macro within SM with the `macro` command

```
macro mname {
```

will start the definition of the macro named `mname`. You then enter SM commands, and terminate the macro definition with a closing `}`.

d. You can define the macro within SM with the `macro edit` command

```
macro edit mname
```

will invoke the macro editor, and you can then enter SM commands to define the macro. The editor is described in detail in the SM manual; the main commands to remember are as follows: In the following descriptions `CONTROL-X` means hold down the CTRL key and then press the `X` key

- the editor is a line editor, not a screen editor. You can advance to the next line with the down arrow on the keyboard, and up to the previous line with the up arrow. Similarly, the right and left arrows advance the cursor one character right and left, respectively.
- within a line, keys will work just like the history editor
- to insert a line above the current line, type `CONTROL-O`
- to insert a line just after the cursor, type `CONTROL-M`
- to erase to the start of the current line, type `CONTROL-U`
- to advance to the end of the current line, type `CONTROL-E`
- when you type into an existing line, this acts in insert mode, not overwrite mode. To overwrite an existing character, position the cursor just after the character you want to overwrite, use the delete key to erase the character, and then type in the new character (or put the cursor on the character and type `CONTROL-D`, or lookup how to set overwrite mode in the real manual)
- to exit the macro editor, and save the changes, type `CONTROL-X`.

7 How do I save macros?

Once you have defined the macro, the command

```
macro write macro1 macro_file.dat
```

will write the macro named macro1 to the file macro_file.dat The `macro write` command remembers the name of the last file it wrote a macro to, and if the filename is the same in the next command, it will append the new macro to the file, otherwise it will delete it first (you can get round this – see section "Macro" in *The SM Manual*). In this way, related macros can be written to the same file.

Another (maybe easier?) way is to use the save command. the command

```
save save_file
```

will save everything to a file – macros, variables and vectors. To get them all back, say

```
restore save_file
```

You can even logout, go to dinner, come back, restart SM, use restore, and be back where you left off.

8 How do I get a hardcopy of a plot?

You can simply define the hardcopy device with the `device` command, then issue the plot commands, and then type

```
hardcopy
```

which sends the plot to the hardcopy device.

Or, in the more common scenario, you have put the plot on the screen, and fiddled with it until you were happy with it, and then want to plot it to a hardcopy device. In this case, you make use of the fact that SM saves your plotting commands in a buffer, and you can manipulate that command list. The command

```
history
```

will print out the list of commands, in reverse chronological order (or chronological order with `history -`). You can then delete all the commands in that buffer that you don't want with the `DELETE` command.

```
DELETE 1 10
```

will delete lines 1 through 10 from the history list. Once you have deleted all the lines from the history list except the ones you used to make the plot on the screen, you can change devices to the hardcopy device using the `device` command, and then type

```
playback
hardcopy
```

This will execute the commands in the history list, and then print the hardcopy plot. In fact, there is a macro `hcopy` defined to do this for you. `hcopy` sets the device to the hardcopy device (as defined in your `.sm` file on the `printer` line), then does a `playback`, then sends the plot to the hardcopy device, and then resets the device type to be whatever it was when you invoked the `hcopy` macro. You don't have to playback all the lines; both `hcopy` and `playback` have optional arguments to specify the range of lines that you want.

You could also define the commands from the history list into a macro, as discussed in section 6, and invoke the macro to execute the plot commands:

```
macro hcplot 1 20
device qms lca0
hcplot
hardcopy
```

This will execute the plot commands from the history buffer lines 1 through 20, and then send the plot to the hardcopy device for printing.

An important point to note about the hardcopy devices is that you have to reissue the `device` command each time you do a `hardcopy` command. This is because the hardcopy plot vectors are

actually written to a file, and this file is closed, sent to the plotter, and deleted when you issue the `hardcopy` command. No new file is opened for you automatically, so you must issue the `device` command to open a new plot file if you want another hardcopy plot, or to redefine the device to a terminal, if that is what you want. You may be able to use the `PAGE` command to start a new page without starting a whole new plot.

9 What about 2-dimensional data?

SM has some capability for handling image data. You can define an image with the `image` command, which is analogous to the `data` command for vectors. As described in the manual, you must first tell SM what sort of image file it is. Binary data is rather tricky to define in a general way, and certainly differs from one operating system to the next, so the few standard types of binary files we have defined will hopefully cover most cases, and if not, you can always write a program to convert your data into one of those types, or try to teach SM about your data format after reading the filecap appendix to the manual (see section "Filecap" in *The SM Manual*).

Once you have read in the image, you can contour it with the `contour` command (first define the contour levels with the `levels` command), you can take a slice through it with the `set x = image(x,y)` command, you can draw it as a surface plot (see section "Surface" in *The SM Manual*), or you can draw a greyscale version of the data (this is a macro. Say `HELP greyscale` for details, or `load demos grey_sincos` for a demonstration).

It is possible that more capabilities will be added someday, but SM is not intended to be an image processing system.

10 What are the other common commands?

- You can leave SM by saying `quit`.
- Perhaps the next most important thing to note in this context is that SM has a command line editor, which allows you to recall previously typed commands (use the up arrow to scroll back through them) and either re-execute them, or edit them. For those of you familiar with VMS, the command line editor is very similar to the one provided with that system.
- The next most important thing is that there is a `CONTROL-C` trap in SM, so if you start a command and regret it, you should be able to abort it by typing `CONTROL-C`.
- If you don't like the axes drawn for you with the `box` command, you can tailor them a bit more with the `axis` command.
- `cursor` invokes the device cursor (for devices that have one). You can then read positions from the screen by positioning the cursor, then typing any key except `e` or `q`. Those latter 2 keys are used to exit the cursor routine.
- `end` or `quit` causes SM to exit.
- `expand` changes the size of the points and characters drawn on the screen, as well as the size of axis tickmarks.
- `format` allows you to specify the format of the numbers that are plotted along the axes.
- `help` is a very important command. You can also specify help on a particular command with `help <command_name>`
- `identification` plots an identification line at the top of the graph, giving the date and some other information, of your choice.
- `label` allows you to plot a label on the graph, at the current location. You can specify different fonts and symbols, as described in the manual (see section "Label" in *The SM Manual*).
- You can change the size of the plot window with the `location` command. But in general, if you want to plot more than one graph on the screen (or page) at once, you will probably use the `window` command (see section "Window" in *The SM Manual*).
- `lweight` allows you to change the line thickness. This will also apply to characters that are plotted.
- By default, if the numbers you plotting on the axis are between 0.0001 and 10000, SM will write these numbers out in decimal format. The `notation` command allows you to specify the range of numbers that you want written out in this way, as opposed to being written out in exponential notation.
- `relocate` relocates the current plot position to wherever you specify when you issue the `relocate` command.
- `ticksize` is used to control the spacing of tickmarks on the axes. Its most common use is to define a logarithmic axis. To do this, the first and third argument to the ticksize command should be negative.

- `window` is used to draw more than one graph on a single screen (or piece of paper). As the name implies, it divides the default plot window into n by m subwindows. You can make the windows touch, if that is what you want to do.
- All the system macros (well, all the interesting ones) are listed in an appendix to the main manual. It's worth skimming through the list sometime.

There are many more commands, which are described at the back of the manual. You will regret not reading about them.

11 What are Common Errors, and What Should I Do?

– If you try to run the program, and it says it can't find graphcap, or font, or edit files, you probably don't have a `.sm` file. If you do have one, it must be in your main directory. If it is there, it must have been edited to look for the files in the correct place, and not be trying to read them from Baltimore. If the directory specifications are correct, the files probably have not got the correct access permissions set, so whoever installed SM should fix that.

– If you try to plot a vector that you read from a file, and it says the vector is not defined, it probably means that the file has some non numeric stuff in it, that you didn't skip over with the `lines` command. If not, we have found problems in some cases with SM trying to read a file written by a VMS Fortran program. If yours is such a file, just use the VMS editor to make a new copy of the file, and it should be ok. The thing to look for is whether the file has Fortran carriage control attributes (do a `dir/full` command on the file, and it will tell you). When you edit it and make a new copy, those attributes will be replaced with normal ones. It does read most Fortran files, and we are not sure how the ones it can't read were written.

– If you make a syntax error, SM will tell you so, and reprint the line with a little arrow indicating the point in the command at which it got confused. One place that a syntax error is apt to arise is if you make use of the ability of SM to accept more than one command on a line. Certain commands cannot be used in this way, because it is ambiguous to the parser what you meant. This is described in the manual (see section "How The Command Interpreter Works" in *The SM Manual*).

12 Where do I go from here?

To the real manual, of course, wherein you will find all the commands described, a list of all currently defined macros and what they do, plus a description of the program structure as well as information about the graphics back end that will enable you to add drivers for other devices. See section “Introduction” in *The SM Manual*.