An SM tutorial for programming and plotting

Gary Mamon

January 2, 2014

1 Introduction

SM (a.k.a. SuperMongo), written by Robert Lupton, is advertised by its author as a graphics language. In fact, SM is much more than a graphics language, it is a high-level scripting programming language! SM has the following advantages:

1. SM is fully vectorial,
2. SM can handle 2D arrays (one at a time),
3. SM allows for vectors of vectors,
4. SM can build vector names from scalars,
5. SM has excellent graphics,
6. SM is widely used in Astronomy.

2 Notation used in this tutorial

In this tutorial, 'SM>' is the SM prompt (which you do not type), `blue courier font` is for typed text (italics for generic items), and green times font is for SM output.

3 Launching and exiting

To launch SM,[1] type:
`sm`

To quit SM:
SM> `quit`

[1] SM works on UNIX, Linux, MacOSX via Terminal or X11 and Windows via Cygwin.

4 Variables and vectors

SM treats variables and vectors differently.

4.1 Variables

Variables are single (scalar) quantities that can be numbers or strings. Their values are assigned with `define` and displayed with `echo`.

```
SM> define x 12.3
SM> echo $x
12.3
```

```
SM> define s abc
SM> echo $s
abc
```

Variables can be combined:

```
SM> define t def
SM> echo $s""$t
abcdef
```

One can build a variable from a variable:

```
SM> define sdef xyz
SM> define u $s$t
SM> echo $u
xyz
```

One can do arithmetic on variables. Enclose in parentheses to evaluate an *expression* and precede by dollar sign to display it

```
SM> echo $(2.3+3.4)
5.7
```

```
SM> define x 2.3
SM> define y 3.4
SM> define z ($x+$y)
SM> echo $z
5.7
```

```
SM> define x 2.3
SM> define y 3.4
SM> echo $($x+$y)
5.7
```

One can list the values of the variables with

```
SM> list define
```

4.2 Vectors

Vectors are like lists or arrays. Their assignment is performed with `set` and their values are displayed with `print`.

```
SM> set x = 23.4
SM> print {x}
x

23.4
```

```
SM> set y = {23.4 34.5}
SM> print {y}
y

23.4
34.5
```

```
set z = 2, 3, .5
SM> print {z}
z

2
2.5
3
```

```
set s = {abc def}
SM> print {s}
s

abc
def
```

A one-element vector can be used in place of a variable. It is a question of philosophy. One can define a variable and set a vector of the same name: the variable and vector are two separate entities. The number of elements of a vector is obtained with `dimen`:

```
SM> echo dimension of x = $(dimen(x)), dimension of y = $(dimen(y))
dimension of x = 1, dimension of y = 2
```

One can list all the vectors and their dimensions with

```
SM> list set
```

SM is vectorized: it can do arithmetic *on equal dimension* vectors as fast as in Fortran or C. On a laptop, it can do arithmetic on vectors of 10^7 elements in a fraction of a second. Here is a typical example:

```
SM> set xx = 1, 3
SM> set yy = xx*xx-0.5
SM> print {xx yy}
 xx   yy

 1    0.5
 2    3.5
 3    8.5
```

If a vector is defined with both numbers and strings, the numbers are stored as strings.

```
SM> set mix = {23.4 abc}
```

Vectors can be concatenated:

```
SM> set u = x concat y concat z
SM> print {x y z u}
    x     y     z     u

 23.4  23.4     2  23.4
       34.5  2.5  23.4
             3    34.5
                  2
                2.5
                  3
```

Vector elements are extracted with square brackets, with the C convention of counting from 0:

```
SM> set y2 = y[1]
SM> print {y2}
y2

34.5
```

```
SM> define y2 (y[1])
SM> echo $y2
34.5
```

```
SM> echo $(y[1])
34.5
```

There are two ways to loop over vectors: with *do loops*:

```
SM> set veclist = {x y z name}
SM> do i = 0, dimen(veclist)-1 {
>> echo dimension of vector $(veclist[$i]) is $(dimen(veclist[$i]))
SM> }
```

and with *foreach loops*:

```
SM> foreach v veclist {
>> echo dimension of vector $v is $(dimen(veclist[$i]))
SM> }
dimension of vector x is 13
dimension of vector y is 13
dimension of vector z is 13
dimension of vector name is 13
```

In both cases, the loop index is a variable. The `foreach` loop is simpler, while the `do` loop is more powerful. To avoid slow code, *one should avoid looping over vector elements* when vectors have over 10^3 or 10^4 elements. In other words, good SM programming (as in other script languages) must be vectorial!

4.3 Summary of variables and vectors

Variables	Vectors
`define s 12.3`	`set v = 1, 2, 0.5`
`echo $s`	`print {v}`
12.3	v
	1
	1.5
	2
`define s delete`	`delete v`

5 Data handling

5.1 Reading data

The following is a basic example of reading data, assumed to be in an equal number of columns, not necessarily formatted to be aligned, and separated by spaces, tabs (i.e. `.tsv` files), or commas (i.e. `.csv` files).

```
SM> cd ~/CAT
SM> data catalog.dat
SM> lines 2 0 # skip one header line (read from 2nd to end (0))
SM> read {id 1.i x 2 y 3 name 4.s}
SM> print {id x y name}
 id     x      y      name

 1      29.4   17.3   Sombrero
 2      12.3   54.6   NGC33379
 (...)
```

SM will automatically skip lines beginning with `#`. By default, SM read a column in floating point (double precision if SM has been properly compiled). Otherwise append `i` or `s` to specify integer and string columns, respectively (see example above).

5.2 Filtering data

One does not always wish to work with an entire data set, but on a subsample instead. SM is particularly powerful to filter data. For example to pick up the values of `x`, `y` and `name` for even values of `id` and positive values of `x`, one can write:

```
SM> foreach vec (x y name) {
>> set _$vec = $vec if (id % 2 == 0 && x > 0)
>> }
SM> print {_x _y _name}
 _x     _y     _name

 12.3   56.4   NGC3379
 (...)
```

Here, we used the `%` modulo operator.

The `foreach` index `vec` is a variable that takes, in succession, the values 'x', 'y', and 'name'. The new (filtered) vectors `_x`, `_y` and `_name` will have equal dimensions, smaller or equal than those of `x`, `y` and `name`.

If one has to do such filtering on several occasions, one can build a vector of vectors:

```
SM> set veclist = {x y name}
SM> foreach vec veclist {
>> set _$vec = $vec if (id % 2 == 0 && x > 0)
>> }
```

with the same effect as before. One can even choose the same name for the vector of vectors and for the index, as SM does not confuse variables and vectors.

5.3 Building a vector using a condition

Another very powerful construct in SM is that it can build vectors on conditions:

```
SM> set w = x > y ?  0 :  y-x
SM> print {x y w}
 x      y      w

 29.4   17.3   0
 12.3   54.6   42.3
 (...)
```

The clause is like in C (the expression before the question mark is the question, that after the question mark is the value if true, that after the colon is the value if false), but works on entire vectors. The different vectors in the question, true-, and false-values *must be of equal dimension*. The output vector is of the *same dimension*.

However, please note that the assignment to `w` is internally performed for all cases. Therefore, if there is a floating exception, you need to work around it. For example, suppose you have 2 non-string vectors `x` and `y` of same dimension and you wish to create a vector `ratio`=`y`/`x`, except that when `x`=0, `ratio`=–1. Then, instead of

```
SM> set ratio = y/x
```

or

```
SM> set ratio = x != 0 ?  y/x :  -1
```

both of which generate error messages when they encounter elements of `x` that are zero, one can use a temporary dummy denominator, `x_tmp`:

```
SM> set x_tmp = x == 0 ?  1 :  x
SM> set ratio = x == 0 ?  -1 :  y/x_tmp
```

6 Plotting

SM was originally designed as a plotting package, so its plotting facilities are pretty good.

6.1 Plotting device

There are 2 basic plotting device families: screen and file. To set up plots for the screen, type:

```
SM> device x11
```

and all plotting commands will appear on the X11 screen.

To set up plots in a file, say `myfile.eps`, type:

```
SM> device postfile myfile.eps
(...)
SM> hardcopy
```

6.2 Basic plotting commands

The typical plotting commands involve setting the plot box limits, setting the line type or point type, setting the width and color of the graphics, plotting as points, line or histogram. Suppose you wish to plot sqrt(x) vs. x for x from 0 to 20 (once the device has been set):

```
SM> set x = 0, 20
SM> set sqrtofx = sqrt(x)
SM> limits x sqrtofx
SM> box
SM> xlabel x
SM> ylabel \sqrt{x}
SM> points x sqrtofx
```

If you wish connected points instead of symbols, replace `points` by `connect`.

If you have error bars in a vector `esqrtofx`, they can be plotted with
`SM> error_y x sqrtofx esqrtofx`.

6.3 Plot settings

The basic plotting symbol is the polygon. The point type is set with `ptype`, which takes 2 arguments: the first is the number of sides of the polygon and the second is 0 for open polygons and 3 for filled ones. If the second argument is 1, the polygon is replaced by a cross with as many branches as the first argument. So to plot an open triangle, a filled circle (20-gon) or a cross, use respectively `ptype 3 0`, `ptype 20 3`, or `ptype 4 1` (the default). Symbols can be expanded, say by a factor 2, with `expand 2`. They can be rotated by, say 45 degrees, with `angle 45`.

For line types, the default is solid, also achieved with `ltype 0`. Dotted and dashed lines can be achieved with `ltype 1` and `ltype 2`, respectively. Line widths are unity by definition, but a quadruple line width is obtained with `lweight 4`.

The plot color can be set to say, red, with `ctype red`. The default is `ctype default`, which sets the color to white on black screens like the default X11 device and to black on devices that point to files. The colors can be abbreviated with numbers: 0 for default, 3 for red, 4 for green, 5 for blue, etc.

All these settings are permanent, i.e. they are valid until they are set to new values.

6.4 Logarithmic axes

Specifying logarithmic axes must be done before the `box` command. For semi-log plots, type:
`SM> ticksize 0 0 -1 0`
and for log-log plots, type:
`SM> ticksize -1 0 -1 0`
If the y error bars are linear, one should use `logerry` instead of `error_y`.

7 Macros and macro files

SM works both interactively, and in macro files. Whereas the interactive interpreter allows for command editing, and has a history (that can be traced back using up arrows or recalled using a special character), it is best to save one's work into files of SM macros. By convention, these SM macro files have suffix `.sm`. Below is an example of an SM macro file.

```
square 1 # square of a vector
    set $0 = $1*$1
```

The first line, with no indentation specifies the name of the macro and its number of arguments. All other lines must be preceded by a TAB. Argument `n` is referenced by `$n`, while `$0` is the returning value (vector really). Comments follow the `#` signs. A comment on the first line will be attached to this macro definition. If this macro `square` is in file `macros.sm`, one reads in this file (once) and executes it as follows:

```
SM> set x = 1, 3
SM> macro read macros.sm
SM> set y = square(x)
SM> print {x y}
 x  y

 1  1
 2  4
 3  9
```

The macro above is like a *function* in Fortran, returning a single entity. One can also define macros to behave like Fortran *subroutines*, returning several entities as arguments:

```
rect2polar 4 # convert cartesian to polar coordinates
    set $3 = sqrt($1*$1+$2*$2)
    set $4 = atand($2/$1) # angle in degrees
```

It is called as follows:

```
SM> set x = {1.2 3.4 0}
SM> set y = {0 3.4 1}
SM> rect2polar x y rho theta
SM> print {x y rho theta}
  x    y    rho   theta

1.2    0    1.2       0
3.4  3.4  4.808      45
  2    1  2.236   26.57
```

8 Global and local variables

In SM macros, all variables and vectors are *global* by default. This means that their values are remembered outside of the macro. This means also that a variable (vector) in a macro will overwrite the corresponding variable (vector) with the same name outside the macro. Moreover,

loop indices are erased once the macro is out of the loop, which erases the variable of the same name outside the macro. These overwriting and erasing of global variables and vectors is a common source of programming error.

To avoid these errors, it is good practice to specify for each vector and variable that it is *local* to the macro:

```
define var local
define var 23.4
set vec local
set vec = {23.4 35.6}
```

```
local define var 23.4

local set vec = {23.4 35.6}
```

Or more globally:

```
mymacro 3
    define var local
    foreach var (xx yy zz vec) { define $var local }
    foreach vec (x y z name id) { set $vec local }
(...)
```

9 Getting help

The basic SM commands (and many useless ones too) are listed with the command `help`. More useful is to query a specific command, for example a standard SM command:

```
SM> help relocate
Syntax: RELOCATE X Y
or RELOCATE ( X Y )

Set the current location to (x,y) without drawing a line. The first
form gives (x,y) in user coordinates, the second in screen coordinates
(0-32767).
```

One can also get help on one's own commands:

```
SM> help rect2polar
Macro:
4 arguments

set $3 = sqrt($1*$1+$2*$2)
set $4 = atand($2/$1)
```

10 Gary's shortcuts

To automatically access Gary's 750+ SM macros from an IAP Linux machine, you must place the following in your .sm file in your home directory:

```
macro2     /nethome/gam/SM/
```

Among, these are several very useful shortcuts:
`h command` → `help command`
`a word` → `apropos word`
`e variable` → `echo $variable`
`e (expr)` → `echo $(expr)`
`dim vector` → `echo $(dimension(vector))`
`dim vector 1` → `echo dimension(vector) = $(dimension(vector))`
`mr macro` → `macro read macro.sm`
`p vector1 (...) vectorn` → `print {vector1 (...) vectorn}`
`initplot fileprefix` (0 for X11 device) → initialize plot (label sizes & line widths to publication quality)
`endplot` → save and view plot and reinitialize
`plot2 vector1 vector2` → scatter plot of *vector2* vs. *vector1* with automatic axes and labels
`plot2 vector1 vector2 1 1` → same with log axis ticks (automatic axes on positive values)