

Advanced Graphics Programming in Stata

Sergiy Radyakin

<mailto:sradyakin@worldbank.org>

Development Economics Research Group
The World Bank

July 29, 2009

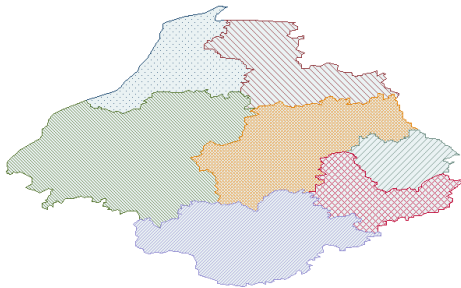
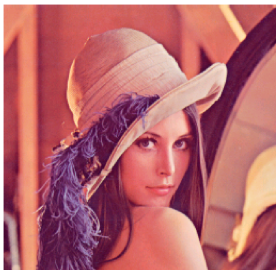
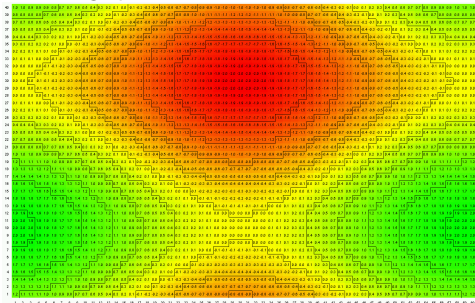
Quote from a statalist message

"...at present, I don't really think that graphics is a strength in Stata. Compared to the myriads of graphs that R can do, Stata can only do simple plots. The main impediment is probably that Stata graphics is not programmable by most users. Could this possibly change in the coming years?"

<http://www.stata.com/statalist/archive/2009-01/msg00872.html>
Fri, Jan 23, 2009 at 11:46 AM

In the next 20 minutes we will learn how to build graphs like these in Stata 9 or later

| | | | | | | | | | | |
|----|------|------|------|------|------|------|------|------|------|------|
| 10 | 1.99 | 1.98 | 1.95 | 1.91 | 1.87 | 1.81 | 1.74 | 1.67 | 1.59 | 1.50 |
| 9 | 1.98 | 1.97 | 1.94 | 1.90 | 1.85 | 1.80 | 1.73 | 1.66 | 1.58 | 1.49 |
| 8 | 1.95 | 1.93 | 1.90 | 1.86 | 1.82 | 1.76 | 1.69 | 1.62 | 1.54 | 1.45 |
| 7 | 1.89 | 1.87 | 1.84 | 1.80 | 1.76 | 1.70 | 1.63 | 1.56 | 1.48 | 1.39 |
| 6 | 1.80 | 1.79 | 1.76 | 1.72 | 1.67 | 1.62 | 1.55 | 1.48 | 1.40 | 1.31 |
| 5 | 1.70 | 1.68 | 1.66 | 1.62 | 1.57 | 1.52 | 1.45 | 1.38 | 1.29 | 1.21 |
| 4 | 1.58 | 1.57 | 1.54 | 1.50 | 1.45 | 1.40 | 1.33 | 1.26 | 1.18 | 1.09 |
| 3 | 1.45 | 1.43 | 1.40 | 1.37 | 1.32 | 1.26 | 1.20 | 1.12 | 1.04 | 0.95 |
| 2 | 1.30 | 1.29 | 1.26 | 1.22 | 1.18 | 1.12 | 1.05 | 0.98 | 0.90 | 0.81 |
| 1 | 1.15 | 1.13 | 1.11 | 1.07 | 1.02 | 0.97 | 0.90 | 0.83 | 0.74 | 0.66 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |



WARNING

- Many features presented here are not documented and thus their behaviour may be unpredictable, especially in different versions of Stata. Proceed at your own risk.
- Illustrations as shown in this presentation have passed through series of conversion from Stata export, through the Beamer package and into the PDF format. This inevitably causes some distortion.
- This is a rather technical presentation: you have to be absolutely comfortable with objects and class programming in Stata.

Stata trivia

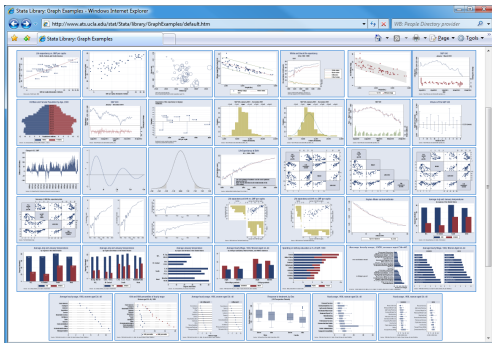
Did you know that (in Stata 9.2) a single command

```
twoway scatter price weight
```

creates 20,002 (twenty thousand and two) classes/objects to show a single graph?

Standard Stata graphics

Stata comes with a number of standard graphs: scatter and line plots, bar and pie charts, etc.



Stata graphics galleries:

<http://www.ats.ucla.edu/stat/Stata/library/GraphExamples/default.htm>

<http://www.survey-design.com.au/Usergraphs.html>

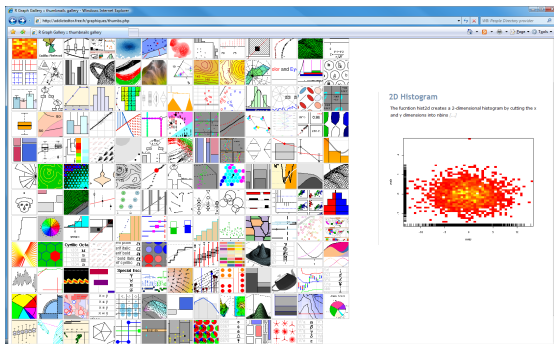
<http://www.stata.com/support/faqs/graphics/gph/statagraphs.html>

There are other types of graphs out there!

There are many more types of charts and graphs out there: shaded charts, heatmaps, contour plots, 3D graphs, etc. Users of R have developed a number of such graphs, see e.g.

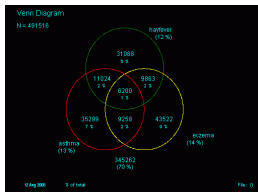
R graphics galleries:

<http://addictedtor.free.fr/graphiques/thumbs.php>

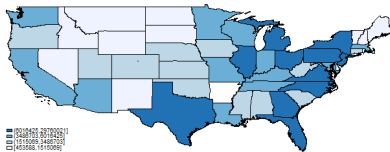


How can we develop custom graphs in Stata?

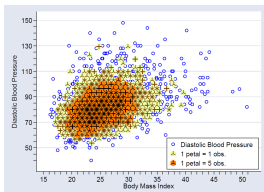
Implementing graphics commands



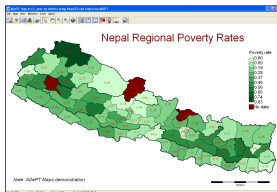
One can use old (pre-Stata 8) documented graphics commands to draw directly, without using the modern Stata graphics engine, like `-venndiag-` by J.M.Lauritsen



Or write a wrapper for a standard command: for example `-tmap-`/`-spmap-` by M.Pisati are in fact wrappers around `-tway area-`. While these commands produce graphics output, they do not add a new graph to Stata's graphical engine.



Another way is to create custom classes required by the graphical engine, for example `-sunflower-`, by W.D.Dupont, W.D.Plummer Jr., T.J.Steichen, N.J.Cox, W.W.Gould, J.S.Pitblado, built-in to Stata 8 and later.



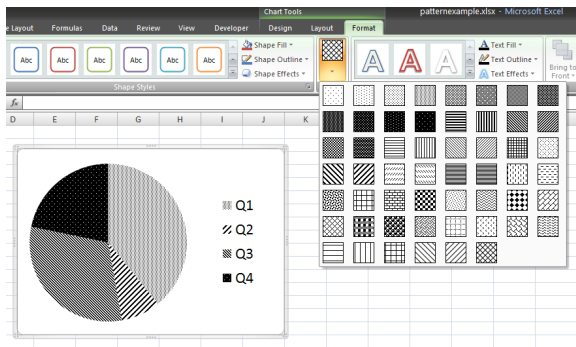
Finally a plugin can be implemented that will communicate with OS libraries directly to create and manage the graph. An early version of the `-amap-` used to work this way before it was moved out of process. These are totally independent from Stata's graphics engine, but are platform-specific

Why develop graphs using Stata's graphics engine?

- Stata graphics engine is implemented as a set of classes and objects with very desirable properties:
 - Stata's cross-platform compatibility, no platform-specific solutions
 - Stata's export facility to produce WMF/EMF/PNG/TIFF/..and any other formats if they are added in the future
 - Stata's *.GPH file format, printing and editing (Stata 10+)
 - Graphs can be made `-graph combine-`able with standard Stata graphs
- The engine provides a mechanism for inheriting standard behavior and properties, as well as overriding them with new, custom ones.
- In particular, programmers do not have to implement features like axes, scales and legend, as long as they are shared with the base (parent) class.
- In the heart of the engine there lies an undocumented command `-gdi-` with its numerous subcommands, which is responsible for communicating with the OS graphics libraries to actually draw something on the screen. It is a successor of the out-of-date command `-gph-` present in Stata 7 and some earlier versions.

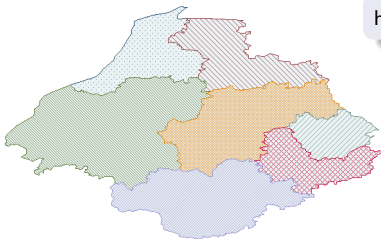
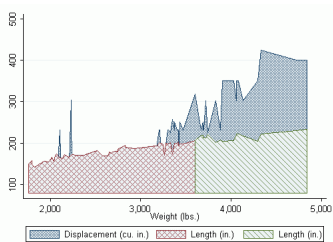
Example problem: pattern fills

- One common request in Stata is creating diagrams with patterned fills for printing (publication) in black-and-white.



- Standard Stata graphics commands do not allow patterned fills.
- Typical solutions suggested were to prepare the data, export it from Stata, and use another graphing package (e.g. Excel) to create the necessary graph.
- Any better solutions?

PAREA - module to generate area graph with pattern fills



PAREA

PAREA by Sergiy Radyakin is a ready to use command, which implements pattern fills, available from SSC since April 2008.

```
findit parea
```

- The command was featured in the "User written Stata graph commands" gallery:

<http://www.survey-design.com.au/Usergraphs.html>

- To understand the Stata graphics engine we will look at one particular type of graphs - area graphs, (other types of graphs bar, pie, etc) can be modified in a similar way.
- Specifically we ask:
 1. How to create patterned fills in Stata?
 2. How to create a new type of graph with this new feature?

How to use the patterns?

In `shadestyle.class` we see the following fragment:

```

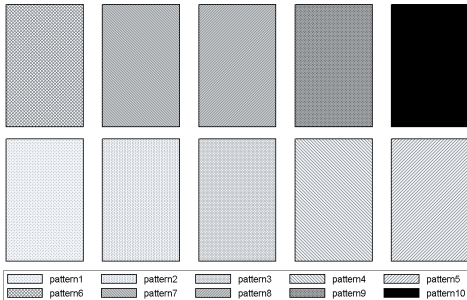
program define setgdi

    capture noisily {
        local holdrgb "`'.color.setting'"
        .color.setintensity100 0 .intensity.val'
        .color.setgdi shade
        .color.setrgb 'holdrgb'
    }

    // .color.setgdi , shade
    gdi shadelevel = 0 .intensity.val'
    if "`'.color.stylename'" == "none" {
        gdi shadepattern = none
    }
    else {
        if "`'.fill.setting'" == "" {
            gdi shadepattern = pattern10
        }
        else {
            gdi shadepattern = `'.fill.setting'
        }
    }

end

```



which suggests that the proper syntax to apply patterns is: `gdi shadepattern=patternname` where *patternname* is anything from the list of patterns: `pattern1`–`pattern10`, `background`, `none`

Now, how do we create our pattern filled twoway area graph? We note that `-twoway-` calls `-graph-`, which is not aware of any particular twoway graphs (implemented as separate programs). So how does Stata know which kinds of twoway graphs are implemented?

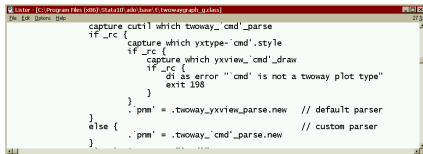
How do we create our pattern filled graph?

It doesn't! It tries. Every time a user requests a `-tway something-` (e.g. `tway para`) Stata follows this route:

```
twoway-->graph-->graph.Graph-->.....-->
-->.twowaygraph_g.new para ...parameters and options...
```

`twowaygraph_g` creates a (parent) object of class `graph_g` and calls own method `parse` to parse the parameters and options. Subsequently it checks presence of the following files (see `twowaygraph_g.class`) :

- `twoway_something_parse.class`
- `yxtype-something.style`
- `yxview_something_draw.ado`



```
capture _rc which twoway_`cmd'_parse
if _rc {
  capture which yxtype-`cmd'.style
  if _rc {
    capture which yxview_`cmd'_draw
    if _rc {
      di as error "'cmd' is not a twoway plot type"
      exit 198
    }
  }
  `prm' = .twoway_yxview_parse.new // default parser
}
else {
  `prm' = .twoway_`cmd'_parse.new // custom parser
}
```

Note how here the presence of the class file is verified not directly by checking the proper file name, but using `-cutil-` command and the class name, (`-cutil-` is same as `-classutil-`).

Creating custom graphic options parser

- After the files are checked, the object of class *twoway_parea_parse* is created by the *twowaygraph_g* class
- Hence in our custom graph we must provide this class to Stata.
- Classes *twoway_something_parse* are responsible for parsing options specific to this graph type, that are not shared with other graphs of the *twoway* family (and hence not handled by the parent).



```
twoway_parea_parse
*! version 1.0.0 11aug2007
version 8
class {
    string pattern="pattern10"
}, inherit(twoway_bar_parse)

// -----
program parse
    .viewtype = "parea"
    .viewclass = "parea_g"
    .super_parse `0'
    local 0, .options'
    _parse combop 0 : 0 , option(pattern)    rightmost
    syntax , [pattern(string) * ]
    .options= `:options'"
    if "`pattern'"!="" {
        .pattern= "pattern'"
    }
end

program log_edits
    args log view i novlabel
    .super_log_edits `0'
    `log'.Arrpush `view'.pattern="`pattern'"
end
```

Creating custom graphic options parser

- This class adds one new property "*pattern*" of type string to the base set of properties that are inherited from the parent class, which I have chosen to be *twoway_bar_parse*. This property is initialized with value "pattern10", which is the default behaviour - solid fill.
- After a new instance of this class is created, it's method `.parse` is called. Note that the parse method calls `.Super.parse '0'` to let the parent object to parse all the options first. Thus everything that the parent (in our case *twoway_bar_parse*) can digest will be removed from the parameters line. All is left for us is to find the parameter for pattern and store it into the declared property.
- In this class you also see the method `log_edits`. The graphs in Stata are not built directly, but rather an internal program (called "log") is created, and then replayed to construct a graph. This allows saving graphs to disk in the [proprietary] *.gph-format.
- In this method I let the parent class do it's work first, then I add (`Arrpush`) to this program line which would set the pattern of the view that is being created.

Creating custom graphic view

The view is the next class that we need to create - *parea_g* class, which inherits from the *yxview*, which implements most useful methods for two-dimensional views.



```

Lister - [C:\DOCUME~1\SRADYA~1\LOCALS~1\Temp\1\_tc\parea_g.class]
File Edit Options Help
// (C) Sergiy Radyakin, Aug. 2007
// Class of twoway graph -- patterned area

class {
    string pattern="pattern10"
} , inherit(yxview)

// -----

program newkey
    if `numkeys' == 0 {
        class exit ""
    }
    syntax [anything(name=keyid)] [ , Position(passthru) ]
    class exit .pareakey_g.new `keyid', pattern("`pattern'") view(`.objkey') `position'
end

```

Note that this class must declare all properties that are going to be used in the log-program commands (in our case we again declare the pattern property and initialize it to solid fill). At the minimum, this is it - all is left is to supply the actual drawing routine.

Here however a newkey method is supplied - this will override the standard method for creating legend keys (because we want the patterns to be in the legend as well). This method doesn't do much, it just creates a new instance of the class *pareakey_g*.

Creating custom legend key

```
lister - [C:\DOCUME~1\SRADYAK~1\LOCALS~1\Temp\1\_tc_pareakey_g.class]
File Edit Options Help
*! version 1.0.1 11aug2007

class {
    string pattern="pattern10"
    string mview=""
}, inherit(yxkey_g)

// -----

program new
    syntax anything(name=keyid), view(string) pattern(string) [ * ]
    .pattern = `"' pattern'"`
    .mview = `"' view'"`
    .Super.new , view(`view') `options'
end

program draw_parea
    local x0=0
    local y0=0
    if `:word count `0''==2 args x1 y1
        else args x0 y0 x1 y1
    .mview'.style.line.setgdi full
    .mview'.style.area.setgdi full
    gdi shadepattern = .pattern'
    gdi shadechange
    gdi rectangle `x0' `y0' `x1' `y1'
end

exit
```

This class implements two methods:

- *new* is the constructor, which is called when an object of this class is created;
- *draw_parea* is responsible for drawing the legend key.

Creating custom graph: drawing

Finally we have to actually draw the graph. When the time is right (when the log-program is replayed) Stata will be calling the following procedure (must be implemented as an *.ado file): `yxview_something_draw` (in our case: `yxview_parea_draw`).

Here it is best to study existing Stata commands to find out what to write in it. In any case drawing is actually performed by calls to an undocumented command `gdi`, which structurally reminds the Windows MetaFile commands or Windows GDI interface.

Command `gdi` is not only undocumented, it is also rarely used. In particular, some of its implemented subcommands are not called anywhere in Stata's classes and *.ado files. So there is plenty of blanks to be filled in.

Drawing with GDI commands

For example, to set the shade we do:

```
gdi shadergb=128 255 100
gdi shadelevel=80
gdi shadepattern=pattern7
gdi shadechange
```

Note that *gdi shadechange* must be called to apply all of the above changes! To draw a line we can:

```
gdi moveto x0 y0
gdi lineto x1 y1
```

Or equivalently

```
gdi line x0 y0 x1 y1
```

Doing more advanced changes

- We have just seen how we can pass one more parameter through the Stata's graphics engine all the way to the drawing procedure. When implementing a custom graph, keep in mind the difference between scalar parameters and vector parameters.
- Scalar parameters are, for example, a number, a string or a color. The memory for their storage can be reserved directly by declaring the corresponding property member in the `twoway_something_parse.class` and `something_g.class`
- Vector parameters are of dimension of data being plotted (e.g. labels for each point on a scatter, etc). These can't be stored in the property members directly, since the required memory is not known at design time. Stata provides a dynamic memory mechanism to store these kind of data particularly for graphics commands. It is based on the concept of **rserset** - a data holder in the Stata's memory outside of the current loaded dataset. Sersets can be saved and loaded, and can also be embedded into other kinds of data (that's how the data is stored with the graph parameters into the `*.gph` file). Nice thing about sersets is that they are documented!

Doing more advanced changes

We add the following line: `.must_create_serset == 1` into the options parser indicating that our graphics command uses additional sersets (in our case to store the values of the cells) and Stata must call a special procedure `log_create_serset` to create them. We then provide this procedure:

```

program log_create_serset
syntax, LOG(name) SERSETNAME(string) [TOUSE(passthru) *]
.log_touse , log(`log') `touse'

if "`sort'" != "" local sortopt sort(`sort')
.log .Arrpush .`sersetName' = .serset.new `varlist' .values_var' .vallab_var' ///
      .channel_r' .channel_g' .channel_b' ///
      [.wtype' .wtxp'] if `touse1' , .omitmethod' ///
      sortopt' `options' nocount

if "`wtype'" != "" {
    .wtvarnum = `word count `varlist'+1'
}

end

program log_edits
args log view i novlabel
.super.log_edits `0'

.log .Arrpush .view'.pattern="`.pattern'"
.log .Arrpush .view'.palette="`.palette'"
.log .Arrpush .view'.grid_color="`.grid_color'"
.log .Arrpush .view'.format="`.format'"

.log .Arrpush .view'.values_var="`.values_var'"
.log .Arrpush .view'.matrix_rows="`.matrix_rows'"
.log .Arrpush .view'.matrix_cols="`.matrix_cols'"
.log .Arrpush .view'.contour="`.contour'"

if "`values_var'" != "" {
    .log .Arrpush .view'.values_var="`.values_var'"
    .log .Arrpush .serset set \ .view'.serset.id'
    .log .Arrpush .view'.vvar=\ :serset varnum `values_var'
}

```

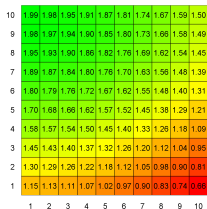
Later while creating the log for graph creation, we pass the index of the values var serset to the `view` object.

Custom graphics command: twoway matrix

`twoway matrix` is a very flexible and powerful command:

- has two modes:
 - plotting values choosing proper values from a color palette, useful for plotting density or intensity of an XY-dependent outcome
 - plotting colors specified directly, useful for digital image processing
- comes with a set of palettes (BW256, Red256, Green256, Blue256, RedGreen256, Yellow256, Acid256).
- R-G-B color components can be specified directly, useful when processing color separately in digital image processing
- supports basic contour option
- formatted values can be displayed in the matrix grid
- rownames/colnames or row/col indices can be displayed on the axes
- currently has a rather inconvenient syntax. it is often simpler to use derived commands, which simplify syntax and handle some preparational work.

Creating derived commands for specialized purposes



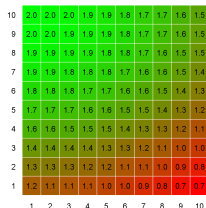
`plotmatrix` is intended to plot matrices:

```
plotmatrix M [, label(string) format(string)
contour(int) grid_color(R G B)]
```

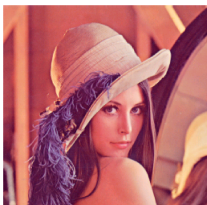
Here:

- M is a required matrix name
- *label* is either "indices" or "names"
- if *format* is specified formatted values are displayed in cells
- if *contour* is specified, its integer value [1-255] is taken as a threshold to separate colors range into color bins: $0(\text{contour})255$.
- if *grid_color* is specified, grid of this color is plotted over the color cells

Other options may be specified to further control the graphics parameters - these will be passed to the underlying graphics command.



Creating derived commands for specialized purposes



`pictureppm` is intended to plot simple (color) graphics imported from *Portable Pixel Map* (PPM) format:

```
pictureppm using "filename.ppm"
```

Other options may be specified to further control the graphics parameters - these will be passed to the underlying graphics command.

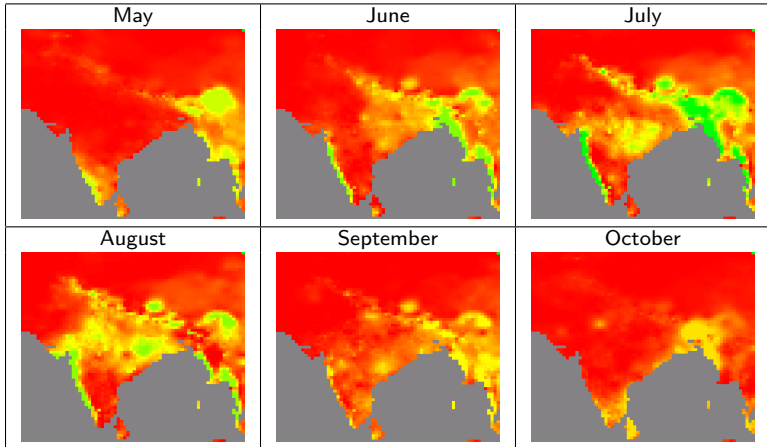
- currently set fixed to 256x256 graphics size, but this can be easily modified
- PPM format is a lossless raster graphics format, where R-G-B color components are stored separately for each pixel in ASCII encoding
- JPG, PNG, GIF, BMP, and other images can be easily converted to PPM with a free image viewer/converter IrfanView <http://www.irfanview.com/>

About the image

By popular demand, here is some more information about the image: **Lenna (Lena)** is image 4.2.04 in the University of Southern California SIPI Image database and is available for research purposes. The image is an equivalent of *auto.dta* dataset in the image-processing/compression communities. Original image is copyrighted by Playboy. Some more information is here: <http://en.wikipedia.org/wiki/Lenna>

Creating more exciting graphics: meteorological data

Monsoon progression in India, 2006, monthly precipitation



Terrestrial Precipitation: 1900-2006 Gridded Monthly Time Series(v1.01)

Center for Climatic Research, Department of Geography, University of Delaware

Drawing with GDI commands

| | | | |
|------------------------|--------------------|--------------------------|-------------------------|
| init | record | textrgb | xalpha, yalpha |
| end | maybedraw | textsize | xbeta, ybeta |
| update | pen | gm_textsize | gbeta |
| record | linergb | textangle | xtransform, ytransform |
| maybedraw | gm_linewidth | textalign | natscale |
| resetregs | linedash | textvalign | tsnatscale |
| topwindow | penchange | textfont | xbounds, ybounds |
| xcur, ycur | shadergb | textchange | |
| xtransform, ytransform | shadelevel | symbol | _____ |
| xreverse, yreverse | shadepattern | symbolsize | |
| xmetric, ymetric | shadechange | gm_symbolsize | Subcommands |
| xsize, ysize | rectangle | pointcloud | mentioned here two |
| newjitterseed | pieslice | scatter | times work in two |
| jitterseed | point | scattervalue | directions - to set and |
| xalpha, yalpha | cpoint | scatterlabel | to get a parameter. |
| xbeta, ybeta | ctext | scatterline | |
| gbeta | polybegin, polyend | scatterweight | _____ |
| | moveto | | |
| | lineto | scatterline_connect_type | Stata 10 added more |
| | line | scatter- | subcommands, e.g. to |
| | gm_rmoveto | line_connect_missing | communicate with the |
| | gm_rlineto | gm_jitter | graph editor. |
| | | jitterseed | |