MGL script language

for version 2.3.5

A.A. Balakin (`http://mathgl.sourceforge.net/`)

This manual is for MathGL (version 2.3.5), a collection of classes and routines for scientific plotting. Please report any errors in this manual to `mathgl.abalakin@gmail.org`.

Table of Contents

1 MGL scripts

MathGL library supports the simplest scripts for data handling and plotting. These scripts can be used independently (with the help of UDAV, mglconv, mglview programs and others

1.1 MGL definition

MGL script language is rather simple. Each string is a command. First word of string is the name of command. Other words are command arguments. Words are separated from each other by space or tabulation symbol. The upper or lower case of words is important, i.e. variables *a* and *A* are different variables. Symbol '`#`' starts the comment (all characters after # will be ignored). The exception is situation when '`#`' is a part of some string. Also options can be specified after symbol '`;`' (see Section 2.7 [Command options], page 15). Symbol '`:`' starts new command (like new line character) if it is not placed inside a string or inside brackets.

If string contain references to external parameters (substrings '`$0`', '`$1`' ... '`$9`') or definitions (substrings '`$a`', '`$b`' ... '`$z`') then before execution the values of parameter/definition will be substituted instead of reference. It allows to use the same MGL script for different parameters (filenames, paths, condition and so on).

Argument can be a string, a variable (data arrays) or a number (scalars).

- The string is any symbols between ordinary marks '`''`'. Long strings can be concatenated from several lines by '`\`' symbol. I.e. the string '`'a +'\<br>' b'`' will give string '`'a + b'`' (here '`<br>`' is newline). Also you can concatenate strings and numbers using '`,`' with out spaces (for example, '`'max(u)=',u.max,' a.u.'`').
- Usually variable have a name which is arbitrary combination of symbols (except spaces and '`''`') started from a letter. Note, you can start an expression with '`!`' symbol if you want to use complex values. For example, the code `new x 100 'x':copy !b !exp(1i*x)` will create real valued data x and complex data b, which is equal to $exp(I*x)$, where $I^2=-1$. A temporary array can be used as variable too:
  - sub-arrays (like in [subdata], page 58, command) as command argument. For example, `a(1)` or `a(1,:)` or `a(1,:,:)` is second row, `a(:,2)` or `a(:,2,:)` is third column, `a(:,:,0)` is first slice and so on. Also you can extract a part of array from m-th to n-th element by code `a(m:n,:,:)` or just `a(m:n)`.
  - any column combinations defined by formulas, like `a('n*w^2/exp(t)')` if names for data columns was specified (by [idset], page 57, command or in the file at string started with `##`).
  - any expression (without spaces) of existed variables produce temporary variable. For example, '`sqrt(dat(:,5)+1)`' will produce temporary variable with data values equal to `tmp[i,j] = sqrt(dat[i,5,j]+1)`.
  - temporary variable of higher dimensions by help of []. For example, '`[1,2,3]`' will produce a temporary vector of 3 elements {1, 2, 3}; '`[[11,12],[21,22]]`' will produce matrix 2*2 and so on. Here you can join even an arrays of the same dimensions by construction like '`[v1,v2,...,vn]`'.
  - result of code for making new data (see Section 4.6 [Make another data], page 58) inside {}. For example, '`{sum dat 'x'}`' produce temporary variable which contain

result of summation of *dat* along direction 'x'. This is the same array *tmp* as produced by command '`sum tmp dat 'x'`'. You can use nested constructions, like '`{sum {max dat 'z'} 'x'}`'.

Temporary variables can not be used as 1st argument for commands which create (return) the data (like '`new`', '`read`', '`hist`' and so on).

- Special names `nan=#QNAN, inf=INFINITY, rnd=random value, pi=3.1415926..., on=1, off=0, :=-1`, variables with suffixes (see Section 4.9 [Data information], page 63), names defined by [define], page 3, command are treated as number. Also results of formulas with sizes 1x1x1 are treated as number (for example, '`pi/dat.nx`').

Before the first using all variables must be defined with the help of commands, like, [new], page 54, [var], page 56, [list], page 55, [copy], page 54, [read], page 57, [hist], page 59, [sum], page 60, and so on (see sections Section 4.2 [Data constructor], page 54, Section 4.4 [Data filling], page 55, and Section 4.6 [Make another data], page 58).

Command may have several set of possible arguments (for example, `plot ydat` and `plot xdat ydat`). All command arguments for a selected set must be specified. However, some arguments can have default values. These argument are printed in [], like `text ydat ['stl'='']` or `text x y 'txt' ['fnt'='' size=-1]`. At this, the record `[arg1 arg2 arg3 ...]` means `[arg1 [arg2 [arg3 ...]]]`, i.e. you can omit only tailing arguments if you agree with its default values. For example, `text x y 'txt' '' 1` or `text x y 'txt' ''` is correct, but `text x y 'txt' 1` is incorrect (argument `'fnt'` is missed).

You can provide several variants of arguments for a command by using '`?`' symbol for separating them. The actual argument being used is set by [variant], page 4. At this, the last argument is used if the value of [variant], page 4, is large than the number of provided variants. By default the first argument is used (i.e. as for `variant 0`). For example, the first plot will be drawn by blue (default is the first argument '`b`'), but the plot after `variant 1` will be drawn by red dash (the second is '`r|`'):

```
fplot 'x' 'b'?'r'
variant 1
fplot 'x^3' 'b'?'r|'
```

1.2 Program flow commands

Below I show commands to control program flow, like, conditions, loops, define script arguments and so on. Other commands can be found in chapters Chapter 3 [MathGL core], page 17, and Chapter 4 [Data processing], page 54. Note, that some of program flow commands (like [define], page 3, [ask], page 2, [call], page 3, [for], page 4, [func], page 3) should be placed alone in the string.

`chdir` *'path'* [MGL command]

Changes the current directory to *path*.

`ask` *$N 'question'* [MGL command]

Sets *N*-th script argument to answer which give the user on the *question*. Usually this show dialog with question where user can enter some text as answer. Here *N* is digit (0...9) or alpha (a...z).

`define` *$N smth* [MGL command]
Sets *N*-th script argument to *smth*. Note, that *smth* is used as is (with '' symbols if present). Here *N* is digit (0...9) or alpha (a...z).

`define` *name smth* [MGL command]
Create scalar variable `name` which have the numeric value of `smth`. Later you can use this variable as usual number.

`defchr` *$N smth* [MGL command]
Sets *N*-th script argument to character with value evaluated from *smth*. Here *N* is digit (0...9) or alpha (a...z).

`defnum` *$N smth* [MGL command]
Sets *N*-th script argument to number with value evaluated from *smth*. Here *N* is digit (0...9) or alpha (a...z).

`call` *'funcname'* [*ARG1 ARG2 ... ARG9*] [MGL command]
Executes function *fname* (or script if function is not found). Optional arguments will be passed to functions. See also [func], page 3.

`func` *'funcname'* [*narg=0*] [MGL command]
Define the function *fname* and number of required arguments. The arguments will be placed in script parameters $1, $2, ... $9. Note, script execution is stopped at `func` keyword, similarly to [stop], page 4, command. See also [return], page 3.

`return` [MGL command]
Return from the function. See also [func], page 3.

`load` *'filename'* [MGL command]
Load additional MGL command from external module (DLL or .so), located in file *filename*. This module have to contain array with name `mgl_cmd_extra` of type `mglCommand`, which describe provided commands.

`if` *dat 'cond'* [MGL command]
Starts block which will be executed if *dat* satisfy to *cond*.

`if val` [MGL command]
Starts block which will be executed if `val` is nonzero.

`elseif` *dat 'cond'* [MGL command]
Starts block which will be executed if previous `if` or `elseif` is false and *dat* satisfy to *cond*.

`elseif val` [MGL command]
Starts block which will be executed if previous `if` or `elseif` is false and `val` is nonzero.

`else` [MGL command]
Starts block which will be executed if previous `if` or `elseif` is false.

`endif` [MGL command]
Finishes `if/elseif/else` block.

`for` *$N* `v1 v2` [`dv=1`] [MGL command]
Starts cycle with $N-th argument changing from *v1* to *v2* with the step *dv*. Here *N* is digit (0...9) or alpha (a...z).

`for` *$N dat* [MGL command]
Starts cycle with $N-th argument changing for *dat* values. Here *N* is digit (0...9) or alpha (a...z).

`next` [MGL command]
Finishes `for` cycle.

`once val` [MGL command]
The code between `once on` and `once off` will be executed only once. Useful for large data manipulation in programs like UDAV.

`stop` [MGL command]
Terminate execution.

`variant val` [MGL command]
Set variant of argument(s) separated by '`?`' symbol to be used in further commands.

`rkstep` *eq1;... var1;...* [`dt=1`] [MGL command]
Make one step for ordinary differential equation(s) {var1' = eq1, ... } with time-step *dt*. Here variable(s) '`var1`', ... are the ones, defined in MGL script previously. The Runge-Kutta 4-th order method is used for solution.

1.3 LaTeX package

There is LaTeX package `mgltex` (was made by Diego Sejas Viscarra) which allow one to make figures directly from MGL script located in LaTeX file.

For using this package you need to specify `--shell-escape` option for *latex/pdflatex* or manually run *mglconv* tool with produced MGL scripts for generation of images. Don't forgot to run *latex/pdflatex* second time to insert generated images into the output document. Also you need to run *pdflatex* third time to update converted from EPS images if you are using vector EPS output (default).

The package may have following options: `draft`, `final` — the same as in the *graphicx* package; `on`, `off` — to activate/deactivate the creation of scripts and graphics; `comments`, `nocomments` — to make visible/invisible commentaries contained inside `mglcomment` environments; `jpg`, `jpeg`, `png` — to export graphics as JPEG/PNG images; `eps`, `epsz` — to export to uncompressed/compressed EPS format as primitives; `bps`, `bpsz` — to export to uncompressed/compressed EPS format as bitmap (doesn't work with *pdflatex*); `pdf` — to export to 3D PDF; `tex` — to export to *LaTeX/tikz* document.

The package defines the following environments:

'`mgl`' It writes its contents to a general script which has the same name as the LaTeX document, but its extension is *.mgl*. The code in this environment is compiled and the image produced is included. It takes exactly the same optional arguments as the `\includegraphics` command, plus an additional argument *imgext*, which specifies the extension to save the image.

An example of usage of 'mgl' environment would be:

```
\begin{mglfunc}{prepare2d}
  new a 50 40 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
  new b 50 40 '0.6*cos(pi*(x+1))*cos(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
\end{mglfunc}

\begin{figure}[!ht]
  \centering
  \begin{mgl}[width=0.85\textwidth,height=7.5cm]
    fog 0.5
    call 'prepare2d'
    subplot 2 2 0 : title 'Surf plot (default)' : rotate 50 60 : light on : box :

    subplot 2 2 1 : title '"\#" style; meshnum 10' : rotate 50 60 : box
    surf a '#'; meshnum 10

    subplot 2 2 2 : title 'Mesh plot' : rotate 50 60 : box
    mesh a

    new x 50 40 '0.8*sin(pi*x)*sin(pi*(y+1)/2)'
    new y 50 40 '0.8*cos(pi*x)*sin(pi*(y+1)/2)'
    new z 50 40 '0.8*cos(pi*(y+1)/2)'
    subplot 2 2 3 : title 'parametric form' : rotate 50 60 : box
    surf x y z 'BbwrR'
  \end{mgl}
\end{figure}
```

'mgladdon'
: It adds its contents to the general script, without producing any image.

'mglcode'
: Is exactly the same as 'mgl', but it writes its contents verbatim to its own file, whose name is specified as a mandatory argument.

'mglscript'
: Is exactly the same as 'mglcode', but it doesn't produce any image, nor accepts optional arguments. It is useful, for example, to create a MGL script, which can later be post processed by another package like "listings".

'mglblock'
: It writes its contents verbatim to a file, specified as a mandatory argument, and to the LaTeX document, and numerates each line of code.

'mglverbatim'
: Exactly the same as 'mglblock', but it doesn't write to a file. This environment doesn't have arguments.

'mglfunc'
: Is used to define MGL functions. It takes one mandatory argument, which is the name of the function, plus one additional argument, which specifies the number of arguments of the function. The environment needs to contain only the body of the function, since the first and last lines are appended automatically, and

the resulting code is written at the end of the general script, after the [stop], page 4, command, which is also written automatically. The warning is produced if 2 or more function with the same name is defined.

‘`mglcomment`’
: Is used to contain multiline commentaries. This commentaries will be visible/invisible in the output document, depending on the use of the package options `comments` and `nocomments` (see above), or the `\mglcomments` and `\mglnocomments` commands (see bellow).

‘`mglsetup`’
: If many scripts with the same code are to be written, the repetitive code can be written inside this environment only once, then this code will be used automatically every time the ‘`\mglplot`’ command is used (see below). It takes one optional argument, which is a name to be associated to the corresponding contents of the environment; this name can be passed to the ‘`\mglplot`’ command to use the corresponding block of code automatically (see below).

The package also defines the following commands:

‘`\mglplot`’
: It takes one mandatory argument, which is MGL instructions separated by the symbol ‘`:`’ this argument can be more than one line long. It takes the same optional arguments as the ‘`mgl`’ environment, plus an additional argument *setup*, which indicates the name associated to a block of code inside a ‘`mglsetup`’ environment. The code inside the mandatory argument will be appended to the block of code specified, and the resulting code will be written to the general script.

  An example of usage of ‘`\mglplot`’ command would be:

```
\begin{mglsetup}
    box '@{W9}' : axis
\end{mglsetup}
\begin{mglsetup}[2d]
  box : axis
  grid 'xy' ';k'
\end{mglsetup}
\begin{mglsetup}[3d]
  rotate 50 60
  box : axis : grid 'xyz' ';k'
\end{mglsetup}
\begin{figure}[!ht]
  \centering
  \mglplot[scale=0.5]{new a 200 'sin(pi*x)' : plot a '2B'}
\end{figure}
\begin{figure}[!ht]
  \centering
  \mglplot[scale=0.5,setup=2d]{
    fplot 'sin(pi*x)' '2B' :
    fplot 'cos(pi*x^2)' '2R'
```

```
  }
\end{figure}
\begin{figure}[!ht]
  \centering
  \mglplot[setup=3d]{fsurf 'sin(pi*x)+cos(pi*y)'}
\end{figure}
```

'`\mglgraphics`'
: This command takes the same optional arguments as the '`mgl`' environment, and one mandatory argument, which is the name of a MGL script. This command will compile the corresponding script and include the resulting image. It is useful when you have a script outside the LaTeX document, and you want to include the image, but you don't want to type the script again.

'`\mglinclude`'
: This is like '`\mglgraphics`' but, instead of creating/including the corresponding image, it writes the contents of the MGL script to the LaTeX document, and numerates the lines.

'`\mgldir`'
: This command can be used in the preamble of the document to specify a directory where LaTeX will save the MGL scripts and generate the corresponding images. This directory is also where '`\mglgraphics`' and '`\mglinclude`' will look for scripts.

'`\mglquality`'
: Adjust the quality of the MGL graphics produced similarly to [quality], page 27.

'`\mgltexon, \mgltexoff`'
: Activate/deactivate the creation of MGL scripts and images. Notice these commands have local behavior in the sense that their effect is from the point they are called on.

'`\mglcomment, \mglnocomment`'
: Make visible/invisible the contents of the `mglcomment` environments. These commands have local effect too.

'`\mglTeX`'
: It just pretty prints the name of the package.

As an additional feature, when an image is not found or cannot be included, instead of issuing an error, `mgltex` prints a box with the word '`MGL image not found`' in the LaTeX document.

2 General concepts

The set of MathGL features is rather rich – just the number of basic graphics types is larger than 50. Also there are functions for data handling, plot setup and so on. In spite of it I tried to keep a similar style in function names and in the order of arguments. Mostly it is used for different drawing functions.

There are six most general (base) concepts:

1. **Any picture is created in memory first.** The internal (memory) representation can be different: bitmap picture (for `SetQuality(MGL_DRAW_LMEM)` or `[quality], page 27, 6`) or the list of vector primitives (default). After that the user may decide what he/she want: save to file, display on the screen, run animation, do additional editing and so on. This approach assures a high portability of the program – the source code will produce exactly the same picture in *any* OS. Another big positive consequence is the ability to create the picture in the console program (using command line, without creating a window)!
2. **Every plot settings (style of lines, font, color scheme) are specified by a string.** It provides convenience for user/programmer – short string with parameters is more comprehensible than a large set of parameters. Also it provides portability – the strings are the same in any OS so that it is not necessary to think about argument types.
3. **All functions have "simplified" and "advanced" forms.** It is done for user's convenience. One needs to specify only one data array in the "simplified" form in order to see the result. But one may set parametric dependence of coordinates and produce rather complex curves and surfaces in the "advanced" form. In both cases the order of function arguments is the same: first data arrays, second the string with style, and later string with options for additional plot tuning.
4. **All data arrays for plotting are encapsulated in mglData(A) class.** This reduces the number of errors while working with memory and provides a uniform interface for data of different types (mreal, double and so on) or for formula plotting.
5. **All plots are vector plots.** The MathGL library is intended for handling scientific data which have vector nature (lines, faces, matrices and so on). As a result, vector representation is used in all cases! In addition, the vector representation allows one to scale the plot easily – change the canvas size by a factor of 2, and the picture will be proportionally scaled.
6. **New drawing never clears things drawn already.** This, in some sense, unexpected, idea allows to create a lot of "combined" graphics. For example, to make a surface with contour lines one needs to call the function for surface plotting and the function for contour lines plotting (in any order). Thus the special functions for making this "combined" plots (as it is done in Matlab and some other plotting systems) are superfluous.

In addition to the general concepts I want to comment on some non-trivial or less commonly used general ideas – plot positioning, axis specification and curvilinear coordinates, styles for lines, text and color scheme.

2.1 Coordinate axes

Two axis representations are used in MathGL. The first one consists of normalizing coordinates of data points in axis range (see Section 3.3 [Axis settings], page 21). If `SetCut()` is

`true` then the outlier points are omitted, otherwise they are projected to the bounding box (see Section 3.2.5 [Cutting], page 19). Also, the point will be omitted if it lies inside the box defined by `SetCutBox()` or if the value of formula `CutOff()` is nonzero for its coordinates. After that, transformation formulas defined by `SetFunc()` or `SetCoor()` are applied to the data point (see Section 3.3.2 [Curved coordinates], page 22). Finally, the data point is plotted by one of the functions.

The range of *x, y, z*-axis can be specified by `SetRange()` or [ranges], page 21, functions. Its origin is specified by [origin], page 21, function. At this you can you can use `NAN` values for selecting axis origin automatically.

There is 4-th axis *c* (color axis or colorbar) in addition to the usual axes *x, y, z*. It sets the range of values for the surface coloring. Its borders are automatically set to values of z-range during the call of [ranges], page 21, function. Also, one can directly set it by call `SetRange('c', ...)`. Use [colorbar], page 32, function for drawing the colorbar.

The form (appearence) of tick labels is controlled by `SetTicks()` function (see Section 3.3.3 [Ticks], page 23). Function *SetTuneTicks* switches on/off tick enhancing by factoring out acommon multiplier (for small coordinate values, like 0.001 to 0.002, or large, like from 1000 to 2000) or common component (for narrow range, like from 0.999 to 1.000). Finally, you may use functions `SetTickTempl()` for setting templates for tick labels (it supports TeX symbols). Also, there is a possibility to print arbitrary text as tick labels the by help of `SetTicksVal()` function.

2.2 Color styles

Base colors are defined by one of symbol '`wkrgbcymhRGBCYMHWlenupqLENUPQ`'. The color types are: '`k`' – black, '`r`' – red, '`R`' – dark red, '`g`' – green, '`G`' – dark green, '`b`' – blue, '`B`' – dark blue, '`c`' – cyan, '`C`' – dark cyan, '`m`' – magenta, '`M`' – dark magenta, '`y`' – yellow, '`Y`' – dark yellow (gold), '`h`' – gray, '`H`' – dark gray, '`w`' – white, '`W`' – bright gray, '`l`' – green-blue, '`L`' – dark green-blue, '`e`' – green-yellow, '`E`' – dark green-yellow, '`n`' – sky-blue, '`N`' – dark sky-blue, '`u`' – blue-violet, '`U`' – dark blue-violet, '`p`' – purple, '`P`' – dark purple, '`q`' – orange, '`Q`' – dark orange (brown).

You can also use "bright" colors. The "bright" color contain 2 symbols in brackets '`{cN}`': first one is the usual symbol for color id, the second one is a digit for its brightness. The digit can be in range '`1`'...'`9`'. Number '`5`' corresponds to a normal color, '`1`' is a very dark version of the color (practically black), and '`9`' is a very bright version of the color (practically white). For example, the colors can be '`{b2}`' '`{b7}`' '`{r7}`' and so on.

Finally, you can specify RGB or RGBA values of a color using format '`{xRRGGBB}`' or '`{xRRGGBBAA}`' correspondingly. For example, '`{xFF9966}`' give you melone color.

2.3 Line styles

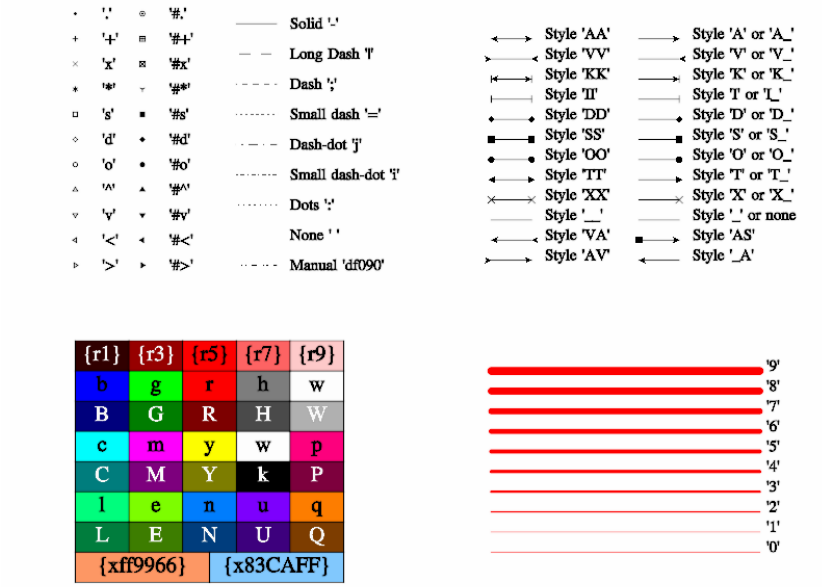
The line style is defined by the string which may contain specifications for color ('`wkrgbcymhRGBCYMHWlenupqLENUPQ`'), dashing style ('`-|;:ji=`' or space), width ('`123456789`') and marks ('`*o+xsd.^v<>`' and '`#`' modifier). If one of the type of information is omitted then default values used with next color from palette (see Section 3.2.7 [Palette and colors], page 20). Note, that internal color counter will be nullified by any change of palette. This includes even hidden change (for example, by [box],

page 33, or [axis], page 31, functions). By default palette contain following colors: dark gray ‘H’, blue ‘b’, green ‘g’, red ‘r’, cyan ‘c’, magenta ‘m’, yellow ‘y’, gray ‘h’, blue-green ‘l’, sky-blue ‘n’, orange ‘q’, yellow-green ‘e’, blue-violet ‘u’, purple ‘p’.

Dashing style has the following meaning: space – no line (usable for plotting only marks), ‘-’ – solid line (################), ‘|’ – long dashed line (########--------), ‘;’ – dashed line (####----####----), ‘=’ – small dashed line (##--##--##--##--), ‘:’ – dotted line (#---#---#---#---), ‘j’ – dash-dotted line (#######----#----), ‘i’ – small dash-dotted line (###--#--###--#--), ‘{dNNNN}’ – manual mask style (for v.2.3 and later, like ‘{df090}’ for (####----#--#----)).

Marker types are: ‘o’ – circle, ‘+’ – cross, ‘x’ – skew cross, ‘s’ - square, ‘d’ - rhomb (or diamond), ‘.’ – dot (point), ‘^’ – triangle up, ‘v’ – triangle down, ‘<’ – triangle left, ‘>’ – triangle right, ‘#*’ – Y sign, ‘#+’ – squared cross, ‘#x’ – squared skew cross, ‘#.’ – circled dot. If string contain symbol ‘#’ then the solid versions of markers are used.

One may specify to draw a special symbol (an arrow) at the beginning and at the end of line. This is done if the specification string contains one of the following symbols: ‘A’ – outer arrow, ‘V’ – inner arrow, ‘I’ – transverse hatches, ‘K’ – arrow with hatches, ‘T’ – triangle, ‘S’ – square, ‘D’ – rhombus, ‘O’ – circle, ‘X’ – skew cross, ‘_’ – nothing (the default). The following rule applies: the first symbol specifies the arrow at the end of line, the second specifies the arrow at the beginning of the line. For example, ‘r-A’ defines a red solid line with usual arrow at the end, ‘b|AI’ defines a blue dash line with an arrow at the end and with hatches at the beginning, ‘_O’ defines a line with the current style and with a circle at the beginning. These styles are applicable during the graphics plotting as well (for example, Section 3.11 [1D plotting], page 34).



2.4 Color scheme

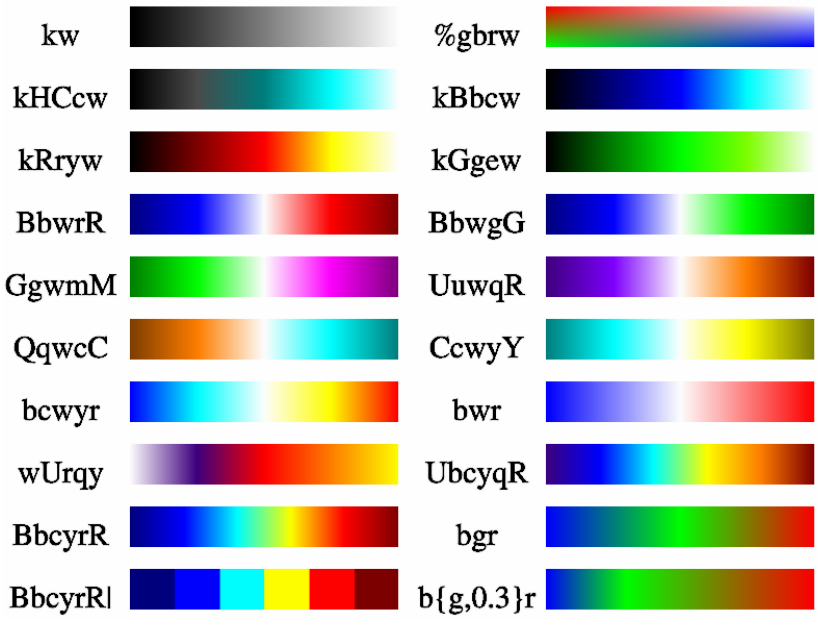
The color scheme is used for determining the color of surfaces, isolines, isosurfaces and so on. The color scheme is defined by the string, which may contain several characters that are color id (see Section 2.3 [Line styles], page 9) or characters '`#:|`'. Symbol '`#`' switches to mesh drawing or to a wire plot. Symbol '`|`' disables color interpolation in color scheme, which can be useful, for example, for sharp colors during matrix plotting. Symbol '`:`' terminate the color scheme parsing. Following it, the user may put styles for the text, rotation axis for curves/isocontours, and so on. Color scheme may contain up to 32 color values.

The final color is a linear interpolation of color array. The color array is constructed from the string ids (including "bright" colors, see Section 2.2 [Color styles], page 9). The argument is the amplitude normalized in color range (see Section 3.3 [Axis settings], page 21). For example, string containing 4 characters '`bcyr`' corresponds to a colorbar from blue (lowest value) through cyan (next value) through yellow (next value) to the red (highest value). String '`kw`' corresponds to a colorbar from black (lowest value) to white (highest value). String '`m`' corresponds to a simple magenta color.

The special 2-axis color scheme (like in [map], page 46, plot) can be used if it contain symbol '`%`'. In this case the second direction (alpha channel) is used as second coordinate for colors. At this, up to 4 colors can be specified for corners: {c1,a1}, {c2,a1}, {c1,a2}, {c2,a2}. Here color and alpha ranges are {c1,c2} and {a1,a2} correspondingly. If one specify less than 4 colors then black color is used for corner {c1,a1}. If only 2 colors are specified then the color of their sum is used for corner {c2,a2}.

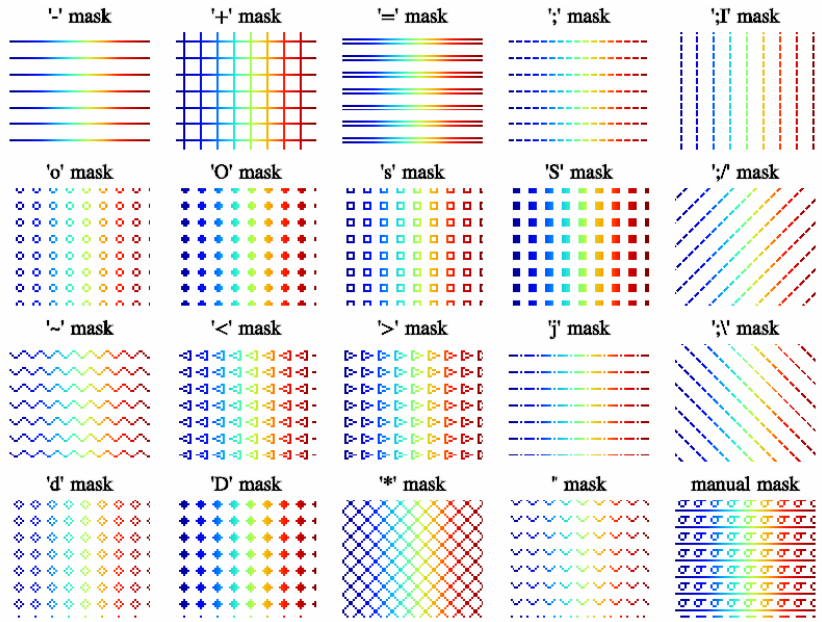
There are several useful combinations. String '`kw`' corresponds to the simplest gray color scheme where higher values are brighter. String '`wk`' presents the inverse gray color scheme where higher value is darker. Strings '`kRryw`', '`kGgw`', '`kBbcw`' present the well-known *hot*, *summer* and *winter* color schemes. Strings '`BbwrR`' and '`bBkRr`' allow to view bi-color figure on white or black background, where negative values are blue and positive values are red. String '`BbcyrR`' gives a color scheme similar to the well-known *jet* color scheme.

For more precise coloring, you can change default (equidistant) position of colors in color scheme. The format is '`{CN,pos}`', '`{CN,pos}`' or '`{xRRGGBB,pos}`'. The position value *pos* should be in range [0, 1]. Note, that alternative method for fine tuning of the color scheme is using the formula for coloring (see Section 3.3.2 [Curved coordinates], page 22).



When coloring by *coordinate* (used in [map], page 46), the final color is determined by the position of the point in 3d space and is calculated from formula c=x*c[1] + y*c[2]. Here, c[1], c[2] are the first two elements of color array; x, y are normalized to axis range coordinates of the point.

Additionally, MathGL can apply mask to face filling at bitmap rendering. The kind of mask is specified by one of symbols '`-+=;oOsS~<>jdD*^`' in color scheme. Mask can be rotated by arbitrary angle by command [mask], page 20, or by three predefined values +45, -45 and 90 degree by symbols '`\/I`' correspondingly. Examples of predefined masks are shown on the figure below.



However, you can redefine mask for one symbol by specifying new matrix of size 8*8 as second argument for [mask], page 20, command. For example, the right-down subplot on the figure above is produced by code
```
mask '+' 'ff00182424f800':dens a '3+'
```
or just use manual mask style (for v.2.3 and later)
```
dens a '3{s00ff00182424f800}'
```

2.5 Font styles

Text style is specified by the string which may contain: color id characters '`wkrgbcymhRGBCYMHW`' (see Section 2.2 [Color styles], page 9), and font style ('`ribwou`') and/or alignment ('`LRC`') specifications. At this, font style and alignment begin after the separator '`:`'. For example, '`r:iCb`' sets the bold ('`b`') italic ('`i`') font text aligned at the center ('`C`') and with red color ('`r`'). Starting from MathGL v.2.3, you can set not single color for whole text, but use color gradient for printed text (see Section 2.4 [Color scheme], page 11).

The font styles are: '`r`' – roman (or regular) font, '`i`' – italic style, '`b`' – bold style. By default roman roman font is used. The align types are: '`L`' – align left (default), '`C`' – align center, '`R`' – align right, '`T`' – align under, '`V`' – align center vertical. Additional font effects are: '`w`' – wired, '`o`' – over-lined, '`u`' – underlined.

Also a parsing of the LaTeX-like syntax is provided. There are commands for the font style changing inside the string (for example, use \b for bold font): \a or \overline – over-lined, \b or \textbf – bold, \i or \textit – italic, \r or \textrm – roman (disable bold and italic attributes), \u or \underline – underlined, \w or \wire – wired, \big – bigger size, @ – smaller size. The lower and upper indexes are specified by '`_`' and '`^`' symbols. At this the changed font style is applied only on next symbol or symbols in braces {}. The text in braces {} are treated as single symbol that allow one to print the index of index. For example, compare the strings '`sin (x^{2^3})`' and '`sin (x^2^3)`'. You may also change text color inside string by command #? or by \color? where '`?`' is symbolic id of the color (see Section 2.2 [Color styles], page 9). For example, words '`blue`' and '`red`' will be colored in the string '`#b{blue} and \colorr{red} text`'. The most of functions understand the newline symbol '`\n`' and allows to print multi-line text. Finally, you can use arbitrary (if it was defined in font-face) UTF codes by command `\utf0x????`. For example, `\utf0x3b1` will produce α symbol.

The most of commands for special TeX or AMSTeX symbols, the commands for font style changing (\textrm, \textbf, \textit, \textsc, \overline, \underline), accents (\hat, \tilde, \dot, \ddot, \acute, \check, \grave, \bar, \breve) and roots (\sqrt, \sqrt3, \sqrt4) are recognized. The full list contain approximately 2000 commands. Note that first space symbol after the command is ignored, but second one is printed as normal symbol (space). For example, the following strings produce the same result $\tilde{a}$: '`\tilde{a}`'; '`\tilde a`'; '`\tilde{}a`'.

In particular, the Greek letters are recognizable special symbols: α – \alpha, β – \beta, γ – \gamma, δ – \delta, ϵ – \epsilon, η – \eta, ι – \iota, χ – \chi, κ – \kappa, λ – \lambda, μ – \mu, ν – \nu, o – \o, ω – \omega, ϕ – \phi, π – \pi, ψ – \psi, ρ – \rho, σ – \sigma, θ – \theta, τ – \tau, υ – \upsilon, ξ – \xi, ζ – \zeta, ς – \varsigma, ε – \varepsilon, ϑ – \vartheta, φ – \varphi, A – \Alpha, B – \Beta, Γ – \Gamma, Δ – \Delta, E – \Epsilon,

H – \Eta, I – \Iota, C – \Chi, K – \Kappa, Λ – \Lambda, M – \Mu, N – \Nu, O – \O, Ω – \Omega, Φ – \Phi, Π – \Pi, Ψ – \Psi, R – \Rho, Σ – \Sigma, Θ – \Theta, T – \Tau, ϒ – \Upsilon, Ξ – \Xi, Z – \Zeta.

The small part of most common special TeX symbols are: ∠ – \angle, ℵ – \aleph, · – \cdot, ♣ – \clubsuit, ∪ – \cup, ∩ – \cap, ♢ – \diamondsuit, ⋄ – \diamond, ÷ – \div, ↓ – \downarrow, † – \dag, ‡ – \ddag, ≡ – \equiv, ∃ – \exists, ⌢ – \frown, ♭ – \flat, ≥ – \ge, ≥ – \geq, ← – \gets, ♡ – \heartsuit, ∞ – \infty, ∈ – \in, ∫ – \int, ℑ – \Im, ⟨ – \langle, ≤ – \le, ≤ – \leq, ← – \leftarrow, ∓ – \mp, ∇ – \nabla, ≠ – \ne, ≠ – \neq, ♮ – \natural, ∮ – \oint, ⊙ – \odot, ⊕ – \oplus, ∂ – \partial, ∥ – \parallel, ⊥ – \perp, ± – \pm, ∝ – \propto, ∏ – \prod, ℜ – \Re, → – \rightarrow, ⟩ – \rangle, ♠ – \spadesuit, ∼ – \sim, ⌣ – \smile, ⊂ – \subset, ⊃ – \supset, √ – \sqrt or \surd, § – \S, ♯ – \sharp, ∑ – \sum, × – \times, → – \to, ↑ – \uparrow, ℘ – \wp and so on.

The font size can be defined explicitly (if *size*>0) or relatively to a base font size as |*size*|**FontSize* (if *size*<0). The value *size*=0 specifies that the string will not be printed. The base font size is measured in internal "MathGL" units. Special functions `SetFontSizePT(), SetFontSizeCM(), SetFontSizeIN()` (see Section 3.2.6 [Font settings], page 20) allow one to set it in more "common" variables for a given dpi value of the picture.

2.6 Textual formulas

MathGL have the fast variant of textual formula evaluation . There are a lot of functions and operators available. The operators are: '`+`' – addition, '`-`' – subtraction, '`*`' – multiplication, '`/`' – division, '`^`' – integer power. Also there are logical "operators": '`<`' – true if x<y, '`>`' – true if x>y, '`=`' – true if x=y, '`&`' – true if x and y both nonzero, '`|`' – true if x or y nonzero. These logical operators have lowest priority and return 1 if true or 0 if false.

The basic functions are: '`sqrt(x)`' – square root of *x*, '`pow(x,y)`' – power *x* in *y*, '`ln(x)`' – natural logarithm of *x*, '`lg(x)`' – decimal logarithm of *x*, '`log(a,x)`' – logarithm base *a* of *x*, '`abs(x)`' – absolute value of *x*, '`sign(x)`' – sign of *x*, '`mod(x,y)`' – *x* modulo *y*, '`step(x)`' – step function, '`int(x)`' – integer part of *x*, '`rnd`' – random number, '`random(x)`' – random data of size as in *x*, '`pi`' – number $\pi = 3.1415926..., inf = \infty$

Functions for complex numbers '`real(x)`', '`imag(x)`', '`abs(x)`', '`arg(x)`', '`conj(x)`'.

Trigonometric functions are: '`sin(x)`', '`cos(x)`', '`tan(x)`' (or '`tg(x)`'). Inverse trigonometric functions are: '`asin(x)`', '`acos(x)`', '`atan(x)`'. Hyperbolic functions are: '`sinh(x)`' (or '`sh(x)`'), '`cosh(x)`' (or '`ch(x)`'), '`tanh(x)`' (or '`th(x)`'). Inverse hyperbolic functions are: '`asinh(x)`', '`acosh(x)`', '`atanh(x)`'.

There are a set of special functions: '`gamma(x)`' – Gamma function $\Gamma(x) = \int_0^\infty dt t^{x-1} exp(-t)$, '`gamma_inc(x,y)`' – incomplete Gamma function $\Gamma(x,y) = \int_y^\infty dt t^{x-1} exp(-t)$, '`psi(x)`' – digamma function $\psi(x) = \Gamma'(x)/\Gamma(x)$ for x!=0, '`ai(x)`' – Airy function Ai(x), '`bi(x)`' – Airy function Bi(x), '`cl(x)`' – Clausen function, '`li2(x)`' (or '`dilog(x)`') – dilogarithm $Li_2(x) = -\Re \int_0^x ds \log(1-s)/s$, '`sinc(x)`' – compute $sinc(x) = \sin(\pi x)/(\pi x)$ for any value of x, '`zeta(x)`' – Riemann zeta function $\zeta(s) = \sum_k k = 1^\infty k^{-s}$ for arbitrary s!=1, '`eta(x)`' – eta function $\eta(s) = (1 - 2^{1-s})\zeta(s)$ for arbitrary s, '`lp(l,x)`' – Legendre polynomial $P_l(x)$, (|x|<=1, l>=0), '`w0(x)`', '`w1(x)`' – principal branch of the Lambert W functions. Function W(x) is defined to be solution of the equation $W \exp(W) = x$.

The exponent integrals are: '`ci(x)`' – Cosine integral $Ci(x) = \int_0^x dt \cos(t)/t$, '`si(x)`' – Sine integral $Si(x) = \int_0^x dt \sin(t)/t$, '`erf(x)`' – error function $erf(x) = (2/\sqrt(\pi)) \int_0^x dt \exp(-t^2)$, '`ei(x)`' – exponential integral $Ei(x) := -PV(\int_{\{} -x\}^{\infty} dt \exp(-t)/t)$ (where PV denotes the principal value of the integral), '`e1(x)`' – exponential integral $E_1(x) := Re \int_1^{\infty} dt \exp(-xt)/t$, '`e2(x)`' – exponential integral $E_2(x) := Re \int_1^{\infty} dt \exp(-xt)/t^2$, '`ei3(x)`' – exponential integral $Ei_3(x) = \int_0^x dt \exp(-t^3)$ for x>=0.

Bessel functions are: '`j(nu,x)`' – regular cylindrical Bessel function of fractional order *nu*, '`y(nu,x)`' – irregular cylindrical Bessel function of fractional order *nu*, '`i(nu,x)`' – regular modified Bessel function of fractional order *nu*, '`k(nu,x)`' – irregular modified Bessel function of fractional order *nu*.

Elliptic integrals are: '`ee(k)`' – complete elliptic integral is denoted by $E(k) = E(\pi/2, k)$, '`ek(k)`' – complete elliptic integral is denoted by $K(k) = F(\pi/2, k)$, '`e(phi,k)`' – elliptic integral $E(\phi, k) = \int_0^{\phi} dt \sqrt(1 - k^2 \sin^2(t)))$, '`f(phi,k)`' – elliptic integral $F(\phi, k) = \int_0^{\phi} dt 1/\sqrt(1 - k^2 \sin^2(t)))$.

Jacobi elliptic functions are: '`sn(u,m)`', '`cn(u,m)`', '`dn(u,m)`', '`sc(u,m)`', '`sd(u,m)`', '`ns(u,m)`', '`cs(u,m)`', '`cd(u,m)`', '`nc(u,m)`', '`ds(u,m)`', '`dc(u,m)`', '`nd(u,m)`'.

Note, some of these functions are unavailable if MathGL was compiled without GSL support.

There is no difference between lower or upper case in formulas. If argument value lie outside the range of function definition then function returns NaN.

2.7 Command options

Command options allow the easy setup of the selected plot by changing global settings only for this plot. Each option start from symbol '`;`'. Options work so that MathGL remember the current settings, change settings as it being set in the option, execute function and return the original settings back. So, the options are most usable for plotting functions.

The most useful options are `xrange, yrange, zrange`. They sets the boundaries for data change. This boundaries are used for automatically filled variables. So, these options allow one to change the position of some plots. For example, in command `Plot(y,"","xrange 0.1 0.9");` or `plot y; xrange 0.1 0.9` the x coordinate will be equidistantly distributed in range 0.1 ... 0.9. See Section 5.9.18 [Using options], page 169, for sample code and picture.

The full list of options are:

`alpha val` [MGL option]
: Sets alpha value (transparency) of the plot. The value should be in range [0, 1]. See also [alphadef], page 17.

`xrange val1 val2` [MGL option]
: Sets boundaries of x coordinate change for the plot. See also [xrange], page 21.

`yrange val1 val2` [MGL option]
: Sets boundaries of y coordinate change for the plot. See also [yrange], page 21.

`zrange val1 val2` [MGL option]
Sets boundaries of z coordinate change for the plot. See also [zrange], page 21.

`cut val` [MGL option]
Sets whether to cut or to project the plot points lying outside the bounding box. See also [cut], page 19.

`size val` [MGL option]
Sets the size of text, marks and arrows. See also [font], page 20, [marksize], page 19, [arrowsize], page 19.

`meshnum val` [MGL option]
Work like [meshnum], page 19, command.

`legend` *'txt'* [MGL option]
Adds string 'txt' to internal legend accumulator. The style of described line and mark is taken from arguments of the last Section 3.11 [1D plotting], page 34, command. See also [legend], page 33.

`value val` [MGL option]
Set the value to be used as additional numeric parameter in plotting command.

2.8 Interfaces

You can use `mglParse` class for executing MGL scripts from different languages.

3 MathGL core

This chapter contains a lot of plotting commands for 1D, 2D and 3D data. It also encapsulates parameters for axes drawing. Moreover an arbitrary coordinate transformation can be used for each axis. Additional information about colors, fonts, formula parsing can be found in Chapter 2 [General concepts], page 8. The full list of symbols used by MathGL for setting up plots can be found in Section A.1 [Symbols for styles], page 175.

Some of MathGL features will appear only in novel versions. To test used MathGL version you can use following function.

`version` *'ver'* [MGL command]
: Return zero if MathGL version is appropriate for required by *ver*, i.e. if major version is the same and minor version is greater or equal to one in *ver*.

3.1 Create and delete objects

You don't need to create canvas object in MGL.

3.2 Graphics setup

Functions and variables in this group influences on overall graphics appearance. So all of them should be placed *before* any actual plotting function calls.

`reset` [MGL command]
: Restore initial values for all of parameters and clear the image.

3.2.1 Transparency

There are several functions and variables for setup transparency. The general function is [alpha], page 17, which switch on/off the transparency for overall plot. It influence only for graphics which created after [alpha], page 17, call (with one exception, OpenGL). Function [alphadef], page 17, specify the default value of alpha-channel. Finally, function [transptype], page 17, set the kind of transparency. See Section 5.9.2 [Transparency and lighting], page 146, for sample code and picture.

`alpha` [`val=on`] [MGL command]
: Sets the transparency on/off and returns previous value of transparency. It is recommended to call this function before any plotting command. Default value is transparency off.

`alphadef val` [MGL command]
: Sets default value of alpha channel (transparency) for all plotting functions. Initial value is 0.5.

`transptype val` [MGL command]
: Set the type of transparency. Possible values are:
  - Normal transparency ('`0`') – below things is less visible than upper ones. It does not look well in OpenGL mode (mglGraphGL) for several surfaces.
  - Glass-like transparency ('`1`') – below and upper things are commutable and just decrease intensity of light by RGB channel.

- Lamp-like transparency ('2') – below and upper things are commutable and are the source of some additional light. I recommend to set `SetAlphaDef(0.3)` or less for lamp-like transparency.

See Section 5.9.3 [Types of transparency], page 147, for sample code and picture..

3.2.2 Lighting

There are several functions for setup lighting. The general function is [light], page 18, which switch on/off the lighting for overall plot. It influence only for graphics which created after [light], page 18, call (with one exception, OpenGL). Generally MathGL support up to 10 independent light sources. But in OpenGL mode only 8 of light sources is used due to OpenGL limitations. The position, color, brightness of each light source can be set separately. By default only one light source is active. It is source number `0` with white color, located at top of the plot. See Section 5.9.6 [Lighting sample], page 151, for sample code and picture.

`light [val=on]` [MGL command]
Sets the using of light on/off for overall plot. Function returns previous value of lighting. Default value is lightning off.

`light num val` [MGL command]
Switch on/off *n*-th light source separately.

`light num xdir ydir zdir ['col'='w' br=0.5]` [MGL command]
`light num xdir ydir zdir xpos ypos zpos ['col'='w' br=0.5]` [MGL command]
The function adds a light source with identification *n* in direction *d* with color *c* and with brightness *bright* (which must be in range [0,1]). If position *r* is specified and isn't NAN then light source is supposed to be local otherwise light source is supposed to be placed at infinity.

`diffuse val` [MGL command]
Set brightness of diffusive light (only for local light sources).

`ambient val` [MGL command]
Sets the brightness of ambient light. The value should be in range [0,1].

`attachlight val` [MGL command]
Set to attach light settings to [inplot], page 25/[subplot], page 24. Note, OpenGL and some output formats don't support this feature.

3.2.3 Fog

`fog val [dz=0.25]` [MGL command]
Function imitate a fog in the plot. Fog start from relative distance *dz* from view point and its density growths exponentially in depth. So that the fog influence is determined by law ~ 1-exp(-*d***z*). Here *z* is normalized to 1 depth of the plot. If value *d*=0 then the fog is absent. Note, that fog was applied at stage of image creation, not at stage of drawing. See Section 5.9.5 [Adding fog], page 150, for sample code and picture.

3.2.4 Default sizes

These variables control the default (initial) values for most graphics parameters including sizes of markers, arrows, line width and so on. As any other settings these ones will influence only on plots created after the settings change.

`barwidth val` [MGL command]
Sets relative width of rectangles in [bars], page 35, [barh], page 36, [boxplot], page 36, [candle], page 37, [ohlc], page 37. Default value is `0.7`.

`marksize val` [MGL command]
Sets size of marks for Section 3.11 [1D plotting], page 34. Default value is `1`.

`arrowsize val` [MGL command]
Sets size of arrows for Section 3.11 [1D plotting], page 34, lines and curves (see Section 3.7 [Primitives], page 27). Default value is `1`.

`meshnum val` [MGL command]
Sets approximate number of lines in [mesh], page 40, [fall], page 40, [grid2], page 42, and also the number of hachures in [vect], page 47, [dew], page 48, and the number of cells in [cloud], page 43. By default (=0) it draws all lines/hachures/cells.

`facenum val` [MGL command]
Sets approximate number of visible faces. Can be used for speeding up drawing by cost of lower quality. By default (=0) it draws all of them.

`plotid` *'id'* [MGL command]
Sets default name *id* as filename for saving (in FLTK window for example).

`pendelta val` [MGL command]
Changes the blur around lines and text (default is 1). For *val*>1 the text and lines are more sharped. For *val*<1 the text and lines are more blurred.

3.2.5 Cutting

These variables and functions set the condition when the points are excluded (cutted) from the drawing. Note, that a point with NAN value(s) of coordinate or amplitude will be automatically excluded from the drawing. See Section 5.2.9 [Cutting sample], page 84, for sample code and picture.

`cut val` [MGL command]
Flag which determines how points outside bounding box are drawn. If it is `true` then points are excluded from plot (it is default) otherwise the points are projected to edges of bounding box.

`cut x1 y1 z1 x2 y2 z2` [MGL command]
Lower and upper edge of the box in which never points are drawn. If both edges are the same (the variables are equal) then the cutting box is empty.

`cut` *'cond'* [MGL command]
Sets the cutting off condition by formula *cond*. This condition determine will point be plotted or not. If value of formula is nonzero then point is omitted, otherwise it plotted. Set argument as `""` to disable cutting off condition.

3.2.6 Font settings

`font` *'fnt'* [`val=6`] [MGL command]
Font style for text and labels (see text). Initial style is 'fnt'=':rC' give Roman font with centering. Parameter `val` sets the size of font for tick and axis labels. Default font size of axis labels is 1.4 times large than for tick labels. For more detail, see Section 2.5 [Font styles], page 13.

`rotatetext val` [MGL command]
Sets to use or not text rotation.

`loadfont` ['*name*'=''] [MGL command]
Load font typeface from *path/name*. Empty name will load default font.

3.2.7 Palette and colors

`palette` *'colors'* [MGL command]
Sets the palette as selected colors. Default value is `"Hbgrcmyhlnqeup"` that corresponds to colors: dark gray '`H`', blue '`b`', green '`g`', red '`r`', cyan '`c`', magenta '`m`', yellow '`y`', gray '`h`', blue-green '`l`', sky-blue '`n`', orange '`q`', yellow-green '`e`', blue-violet '`u`', purple '`p`'. The palette is used mostly in 1D plots (see Section 3.11 [1D plotting], page 34) for curves which styles are not specified. Internal color counter will be nullified by any change of palette. This includes even hidden change (for example, by [box], page 33, or [axis], page 31, functions).

`gray val` [MGL command]
Sets the gray-scale mode on/off.

3.2.8 Masks

`mask` *'id' 'hex'* [MGL command]
Sets new bit matrix *hex* of size 8*8 for mask with given *id*. This is global setting which influence on any later usage of symbol *id*. The predefined masks are (see Section 2.4 [Color scheme], page 11): '`-`' is 000000FF00000000, '`+`' is 080808FF08080808, '`=`' is 0000FF00FF000000, '`;`' is 0000007700000000, '`o`' is 0000182424180000, '`O`' is 0000183C3C180000, '`s`' is 00003C24243C0000, '`S`' is 00003C3C3C3C0000, '`~`' is 0000060990600000, '`<`' is 0060584658600000, '`>`' is 00061A621A060000, '`j`' is 0000005F00000000, '`d`' is 0008142214080000, '`D`' is 00081C3E1C080000, '`*`' is 8142241818244281, '`^`' is 0000001824420000.

`mask` *angle* [MGL command]
Sets the default rotation angle (in degrees) for masks. Note, you can use symbols '`\`', '`/`', '`I`' in color scheme for setting rotation angles as 45, -45 and 90 degrees correspondingly.

3.2.9 Error handling

All warnings will be displayed automatically in special tool-window or in console.

3.2.10 Stop drawing

You can use [stop], page 4, command or press corresponding toolbutton to stop drawing and script execution.

3.3 Axis settings

These large set of variables and functions control how the axis and ticks will be drawn. Note that there is 3-step transformation of data coordinates are performed. Firstly, coordinates are projected if `Cut=true` (see Section 3.2.5 [Cutting], page 19), after it transformation formulas are applied, and finally the data was normalized in bounding box. Note, that MathGL will produce warning if axis range and transformation formulas are not compatible.

3.3.1 Ranges (bounding box)

`xrange v1 v2 [add=off]` [MGL command]
`yrange v1 v2 [add=off]` [MGL command]
`zrange v1 v2 [add=off]` [MGL command]
`crange v1 v2 [add=off]` [MGL command]

Sets or adds the range for '`x`'-,'`y`'-,'`z`'- coordinate or coloring ('`c`'). If one of values is `NAN` then it is ignored. See also [ranges], page 21.

`xrange` *dat* `[add=off]` [MGL command]
`yrange` *dat* `[add=off]` [MGL command]
`zrange` *dat* `[add=off]` [MGL command]
`crange` *dat* `[add=off]` [MGL command]

Sets the range for '`x`'-,'`y`'-,'`z`'- coordinate or coloring ('`c`') as minimal and maximal values of data *dat*. Parameter `add=on` shows that the new range will be joined to existed one (not replace it).

`ranges x1 x2 y1 y2 [z1=0 z2=0]` [MGL command]

Sets the ranges of coordinates. If minimal and maximal values of the coordinate are the same then they are ignored. Also it sets the range for coloring (analogous to `crange z1 z2`). This is default color range for 2d plots. Initial ranges are [-1, 1].

`origin x0 y0 [z0=nan]` [MGL command]

Sets center of axis cross section. If one of values is NAN then MathGL try to select optimal axis position.

`zoomaxis x1 x2` [MGL command]
`zoomaxis x1 y1 x2 y2` [MGL command]
`zoomaxis x1 y1 z1 x2 y2 z2` [MGL command]
`zoomaxis x1 y1 z1 c1 x2 y2 z2 c2` [MGL command]

Additionally extend axis range for any settings made by `SetRange` or `SetRanges` functions according the formula $min+ = (max - min) * p1$ and $max+ = (max - min) * p1$ (or $min* = (max/min)^{p1}$ and $max* = (max/min)^{p1}$ for log-axis range when $inf > max/min > 100$ or $0 < max/min < 0.01$). Initial ranges are [0, 1]. Attention! this settings can not be overwritten by any other functions, including `DefaultPlotParam()`.

3.3.2 Curved coordinates

`axis` *'fx' 'fy' 'fz'* [*'fa'=''*] [MGL command]

Sets transformation formulas for curvilinear coordinate. Each string should contain mathematical expression for real coordinate depending on internal coordinates '`x`', '`y`', '`z`' and '`a`' or '`c`' for colorbar. For example, the cylindrical coordinates are introduced as `SetFunc("x*cos(y)", "x*sin(y)", "z");`. For removing of formulas the corresponding parameter should be empty or `NULL`. Using transformation formulas will slightly slowing the program. Parameter *EqA* set the similar transformation formula for color scheme. See Section 2.6 [Textual formulas], page 14.

`axis how` [MGL command]

Sets one of the predefined transformation formulas for curvilinear coordinate. Parameter *how* define the coordinates: `mglCartesian=0` – Cartesian coordinates (no transformation); `mglPolar=1` – Polar coordinates $x_n = x*cos(y), y_n = x*sin(y), z_n = z$; `mglSpherical=2` – Sperical coordinates $x_n = x*sin(y)*cos(z), y_n = x*sin(y)*sin(z), z_n = x*cos(y)$; `mglParabolic=3` – Parabolic coordinates $x_n = x*y, y_n = (x*x - y*y)/2, z_n = z$; `mglParaboloidal=4` – Paraboloidal coordinates $x_n = (x*x - y*y)*cos(z)/2, y_n = (x*x - y*y)*sin(z)/2, z_n = x*y$; `mglOblate=5` – Oblate coordinates $x_n = cosh(x)*cos(y)*cos(z), y_n = cosh(x)*cos(y)*sin(z), z_n = sinh(x)*sin(y)$; `mglProlate=6` – Prolate coordinates $x_n = sinh(x)*sin(y)*cos(z), y_n = sinh(x)*sin(y)*sin(z), z_n = cosh(x)*cos(y)$; `mglElliptic=7` – Elliptic coordinates $x_n = cosh(x)*cos(y), y_n = sinh(x)*sin(y), z_n = z$; `mglToroidal=8` – Toroidal coordinates $x_n = sinh(x)*cos(z)/(cosh(x)-cos(y)), y_n = sinh(x)*sin(z)/(cosh(x)-cos(y)), z_n = sin(y)/(cosh(x)-cos(y))$; `mglBispherical=9` – Bispherical coordinates $x_n = sin(y)*cos(z)/(cosh(x)-cos(y)), y_n = sin(y)*sin(z)/(cosh(x)-cos(y)), z_n = sinh(x)/(cosh(x)-cos(y))$; `mglBipolar=10` – Bipolar coordinates $x_n = sinh(x)/(cosh(x)-cos(y)), y_n = sin(y)/(cosh(x)-cos(y)), z_n = z$; `mglLogLog=11` – log-log coordinates $x_n = lg(x), y_n = lg(y), z_n = lg(z)$; `mglLogX=12` – log-x coordinates $x_n = lg(x), y_n = y, z_n = z$; `mglLogY=13` – log-y coordinates $x_n = x, y_n = lg(y), z_n = z$.

`ternary val` [MGL command]

The function sets to draws Ternary (*tern*=`1`), Quaternary (*tern*=`2`) plot or projections (*tern*=`4,5,6`).

Ternary plot is special plot for 3 dependent coordinates (components) a, b, c so that a+b+c=1. MathGL uses only 2 independent coordinates a=x and b=y since it is enough to plot everything. At this third coordinate z act as another parameter to produce contour lines, surfaces and so on.

Correspondingly, Quaternary plot is plot for 4 dependent coordinates a, b, c and d so that a+b+c+d=1. MathGL uses only 3 independent coordinates a=x, b=y and d=z since it is enough to plot everything.

Projections can be obtained by adding value `4` to *tern* argument. So, that *tern*=`4` will draw projections in Cartesian coordinates, *tern*=`5` will draw projections in Ternary coordinates, *tern*=`6` will draw projections in Quaternary coordinates. If you add `8` instead of `4` then all text labels will not be printed on projections.

Use `Ternary(0)` for returning to usual axis. See Section 5.2.6 [Ternary axis], page 79, for sample code and picture. See Section 5.9.4 [Axis projection], page 149, for sample code and picture.

3.3.3 Ticks

`adjust` ['*dir*'='*xyzc*'] [MGL command]

Set the ticks step, number of sub-ticks and initial ticks position to be the most human readable for the axis along direction(s) *dir*. Also set `SetTuneTicks(true)`. Usually you don't need to call this function except the case of returning to default settings.

`xtick val` [`sub=0 org=nan`] [MGL command]
`ytick val` [`sub=0 org=nan`] [MGL command]
`ztick val` [`sub=0 org=nan`] [MGL command]
`ctick val` [`sub=0 org=nan`] [MGL command]

Set the ticks step *d*, number of sub-ticks *ns* (used for positive *d*) and initial ticks position *org* for the axis along direction *dir* (use 'c' for colorbar ticks). Variable *d* set step for axis ticks (if positive) or it's number on the axis range (if negative). Zero value set automatic ticks. If *org* value is NAN then axis origin is used. Parameter *fact* set text which will be printed after tick label (like `"\pi"` for *d*=M_PI).

`xtick val1` *'lbl1'* [`val2` *'lbl2'* ...] [MGL command]
`ytick val1` *'lbl1'* [`val2` *'lbl2'* ...] [MGL command]
`ztick val1` *'lbl1'* [`val2` *'lbl2'* ...] [MGL command]
`xtick` *vdat 'lbls'* [`add=off`] [MGL command]
`ytick` *vdat 'lbls'* [`add=off`] [MGL command]
`ztick` *vdat 'lbls'* [`add=off`] [MGL command]

Set the manual positions *val* and its labels *lbl* for ticks along axis *dir*. If array *val* is absent then values equidistantly distributed in x-axis range are used. Labels are separated by '`\n`' symbol. Use `SetTicks()` to restore automatic ticks.

`xtick` *'templ'* [MGL command]
`ytick` *'templ'* [MGL command]
`ztick` *'templ'* [MGL command]
`ctick` *'templ'* [MGL command]

Set template *templ* for x-,y-,z-axis ticks or colorbar ticks. It may contain TeX symbols also. If *templ*=`""` then default template is used (in simplest case it is '`%.2g`'). If template start with '`&`' symbol then `long` integer value will be passed instead of default type `double`. Setting on template switch off automatic ticks tuning.

`ticktime` *'dir'* [`dv` *'tmpl'*] [MGL command]

Sets time labels with step *val* and template *templ* for x-,y-,z-axis ticks or colorbar ticks. It may contain TeX symbols also. The format of template *templ* is the same as described in `http://www.manpagez.com/man/3/strftime/`. Most common variants are '`%X`' for national representation of time, '`%x`' for national representation of date, '`%Y`' for year with century. If *val*=0 and/or *templ*=`""` then automatic tick step and/or template will be selected. You can use `mgl_get_time`() function for obtaining number of second for given date/time string. Note, that MS Visual Studio couldn't handle date before 1970.

`tuneticks val [pos=1.15]` [MGL command]
Switch on/off ticks enhancing by factoring common multiplier (for small, like from 0.001 to 0.002, or large, like from 1000 to 2000, coordinate values – enabled if *tune*&1 is nonzero) or common component (for narrow range, like from 0.999 to 1.000 – enabled if *tune*&2 is nonzero). Also set the position *pos* of common multiplier/component on the axis: =0 at minimal axis value, =1 at maximal axis value. Default value is 1.15.

`tickshift dx [dy=0 dz=0 dc=0]` [MGL command]
Set value of additional shift for ticks labels.

`origintick val` [MGL command]
Enable/disable drawing of ticks labels at axis origin. In C/Fortran you can use `mgl_set_flag(gr,val, MGL_NO_ORIGIN);`.

`ticklen val [stt=1]` [MGL command]
The relative length of axis ticks. Default value is `0.1`. Parameter *stt*>0 set relative length of subticks which is in `sqrt(1+stt)` times smaller.

`axisstl` *'stl' ['tck'=" 'sub'="]* [MGL command]
The line style of axis (*stl*), ticks (*tck*) and subticks (*sub*). If *stl* is empty then default style is used ('`k`' or '`w`' depending on transparency type). If *tck* or *sub* is empty then axis style is used (i.e. *stl*).

3.4 Subplots and rotation

These functions control how and where further plotting will be placed. There is a certain calling order of these functions for the better plot appearance. First one should be [subplot], page 24, [multiplot], page 25, or [inplot], page 25, for specifying the place. Second one can be [title], page 25, for adding title for the subplot. After it a [rotate], page 25, [shear], page 26, and [aspect], page 26. And finally any other plotting functions may be called. Alternatively you can use [columnplot], page 25, [gridplot], page 25, [stickplot], page 25, [shearplot], page 25, or relative [inplot], page 25, for positioning plots in the column (or grid, or stick) one by another without gap between plot axis (bounding boxes). See Section 5.2.1 [Subplots], page 70, for sample code and picture.

`subplot nx ny m ['stl'='<>_^' dx=0 dy=0]` [MGL command]
Puts further plotting in a *m*-th cell of *nx***ny* grid of the whole frame area. This function set off any aspects or rotations. So it should be used first for creating the subplot. Extra space will be reserved for axis/colorbar if *stl* contain:

- '`L`' or '`<`' – at left side,
- '`R`' or '`>`' – at right side,
- '`A`' or '`^`' – at top side,
- '`U`' or '`_`' – at bottom side,
- '`#`' – reserve none space (use whole region for axis range) – axis and tick labels will be invisible by default.

From the aesthetical point of view it is not recommended to use this function with different matrices in the same frame. The position of the cell can be shifted from its default position by relative size *dx*, *dy*. Note, colorbar can be invisible (be out of image borders) if you set empty style ''.

`multiplot nx ny m dx dy` [*'style'='<>_^'*] [MGL command]

Puts further plotting in a rectangle of *dx***dy* cells starting from *m*-th cell of *nx***ny* grid of the whole frame area. This function set off any aspects or rotations. So it should be used first for creating subplot. Extra space will be reserved for axis/colorbar if *stl* contain:

- '`L`' or '`<`' – at left side,
- '`R`' or '`>`' – at right side,
- '`A`' or '`^`' – at top side,
- '`U`' or '`_`' – at bottom side.

`inplot x1 x2 y1 y2` [`rel=on`] [MGL command]

Puts further plotting in some region of the whole frame surface. This function allows one to create a plot in arbitrary place of the screen. The position is defined by rectangular coordinates [*x1*, *x2*]*[*y1*, *y2*]. The coordinates *x1*, *x2*, *y1*, *y2* are normalized to interval [0, 1]. If parameter *rel*=`true` then the relative position to current [subplot], page 24, (or [inplot], page 25, with *rel*=`false`) is used. This function set off any aspects or rotations. So it should be used first for creating subplot.

`columnplot num ind` [`d=0`] [MGL command]

Puts further plotting in *ind*-th cell of column with *num* cells. The position is relative to previous [subplot], page 24, (or [inplot], page 25, with *rel*=`false`). Parameter *d* set extra gap between cells.

`gridplot nx ny ind` [`d=0`] [MGL command]

Puts further plotting in *ind*-th cell of *nx***ny* grid. The position is relative to previous [subplot], page 24, (or [inplot], page 25, with *rel*=`false`). Parameter *d* set extra gap between cells.

`stickplot num ind tet phi` [MGL command]

Puts further plotting in *ind*-th cell of stick with *num* cells. At this, stick is rotated on angles *tet*, *phi*. The position is relative to previous [subplot], page 24, (or [inplot], page 25, with *rel*=`false`).

`shearplot num ind sx sy` [`xd yd`] [MGL command]

Puts further plotting in *ind*-th cell of stick with *num* cells. At this, cell is sheared on values *sx*, *sy*. Stick direction is specified be *xd* and *yd*. The position is relative to previous [subplot], page 24, (or [inplot], page 25, with *rel*=`false`).

`title` *'title'* [*'stl'='*" `size=-2`] [MGL command]

Add text *title* for current subplot/inplot. Parameter *stl* can contain:

- font style (see, Section 2.5 [Font styles], page 13);
- '`#`' for box around the title.

Parameter *size* set font size. This function set off any aspects or rotations. So it should be used just after creating subplot.

`rotate tetx tetz` [`tety=0`] [MGL command]

Rotates a further plotting relative to each axis {x, z, y} consecutively on angles *TetX*, *TetZ*, *TetY*.

`rotate tet x y z` [MGL command]
Rotates a further plotting around vector {*x*, *y*, *z*} on angle *Tet*.

`shear sx sy` [MGL command]
Shears a further plotting on values *sx*, *sy*.

`aspect ax ay [az=1]` [MGL command]
Defines aspect ratio for the plot. The viewable axes will be related one to another as the ratio *Ax:Ay:Az*. For the best effect it should be used after [rotate], page 25, function. If *Ax* is `NAN` then function try to select optimal aspect ratio to keep equal ranges for x-y axis. At this, *Ay* will specify proportionality factor, or set to use automatic one if *Ay*=`NAN`.

There are 3 functions `View()`, `Zoom()` and `Perspective()` which transform whole image. I.e. they act as secondary transformation matrix. They were introduced for rotating/zooming the whole plot by mouse. It is not recommended to call them for picture drawing.

`perspective val` [MGL command]
Add (switch on) the perspective to plot. The parameter $a = Depth/(Depth + dz) \in [0, 1)$. By default (`a=0`) the perspective is off.

`view tetx tetz [tety=0]` [MGL command]
Rotates a further plotting relative to each axis {x, z, y} consecutively on angles *TetX*, *TetZ*, *TetY*. Rotation is done independently on [rotate], page 25. Attention! this settings can not be overwritten by `DefaultPlotParam()`. Use `Zoom(0,0,1,1)` to return default view.

`zoom x1 y1 x2 y2` [MGL command]
The function changes the scale of graphics that correspond to zoom in/out of the picture. After function call the current plot will be cleared and further the picture will contain plotting from its part [x1,x2]*[y1,y2]. Here picture coordinates *x1*, *x2*, *y1*, *y2* changes from 0 to 1. Attention! this settings can not be overwritten by any other functions, including `DefaultPlotParam()`. Use `Zoom(0,0,1,1)` to return default view.

3.5 Export picture

Functions in this group save or give access to produced picture. So, usually they should be called after plotting is done.

`setsize w h` [MGL command]
Sets size of picture in pixels. This function **should be** called before any other plotting because it completely remove picture contents if *clear*=`true`. Function just clear pixels and scale all primitives if *clear*=`false`.

`setsizescl factor` [MGL command]
Set factor for width and height in all further calls of [setsize], page 26.

`quality` [`val`=*2*] [MGL command]
: Sets quality of the plot depending on value *val*: `MGL_DRAW_WIRE=0` – no face drawing (fastest), `MGL_DRAW_FAST=1` – no color interpolation (fast), `MGL_DRAW_NORM=2` – high quality (normal), `MGL_DRAW_HIGH=3` – high quality with 3d primitives (arrows and marks); `MGL_DRAW_LMEM=0x4` – direct bitmap drawing (low memory usage); `MGL_DRAW_DOTS=0x8` – for dots drawing instead of primitives (extremely fast).

3.5.1 Export to file

These functions export current view to a graphic file. The filename *fname* should have appropriate extension. Parameter *descr* gives the short description of the picture. Just now the transparency is supported in PNG, SVG, OBJ and PRC files.

`write` [*'fname'*=*''*] [MGL command]
: Exports current frame to a file *fname* which type is determined by the extension. Parameter *descr* adds description to file (can be `""`). If *fname*=`""` then the file '`frame####.jpg`' is used, where '`####`' is current frame id and name '`frame`' is defined by [plotid], page 19, class property.

3.5.2 Frames/Animation

There are no commands for making animation in MGL. However you can use features of `mglconv` and `mglview` utilities. For example, by busing special comments '`##a` ' or '`##c` '.

3.5.3 Bitmap in memory

3.5.4 Parallelization

3.6 Background

These functions change background image.

`clf` [*'col'*] [MGL command]
: Clear the picture and fill background by specified color.

`rasterize` [MGL command]
: Force drawing the plot and use it as background. After it, function clear the list of primitives, like [clf], page 27. This function is useful if you want save part of plot as bitmap one (for example, large surfaces, isosurfaces or vector fields) and keep some parts as vector one (like annotation, curves, axis and so on).

`background` *'fname'* [`alpha=1`] [MGL command]
: Load PNG or JPEG file *fname* as background for the plot. Parameter *alpha* manually set transparency of the background.

3.7 Primitives

These functions draw some simple objects like line, point, sphere, drop, cone and so on. See Section 5.9.7 [Using primitives], page 153, for sample code and picture.

`ball x y ['col'='r.']` [MGL command]
`ball x y z ['col'='r.']` [MGL command]
Draws a mark (point '`.`' by default) at position $p=\{x, y, z\}$ with color *col*.

`errbox x y ex ey ['stl'='']` [MGL command]
`errbox x y z ex ey ez ['stl'='']` [MGL command]
Draws a 3d error box at position $p=\{x, y, z\}$ with sizes $e=\{ex, ey, ez\}$ and style *stl*. Use NAN for component of e to reduce number of drawn elements.

`line x1 y1 x2 y2 ['stl'='']` [MGL command]
`line x1 y1 z1 x2 y2 z2 ['stl'='']` [MGL command]
Draws a geodesic line (straight line in Cartesian coordinates) from point *p1* to *p2* using line style *stl*. Parameter *num* define the "quality" of the line. If *num*=2 then the stright line will be drawn in all coordinate system (independently on transformation formulas (see Section 3.3.2 [Curved coordinates], page 22). Contrary, for large values (for example, =100) the geodesic line will be drawn in corresponding coordinate system (straight line in Cartesian coordinates, circle in polar coordinates and so on). Line will be drawn even if it lies out of bounding box.

`curve x1 y1 dx1 dy1 x2 y2 dx2 dy2 ['stl'='']` [MGL command]
`curve x1 y1 z1 dx1 dy1 dz1 x2 y2 z2 dx2 dy2 dz2 ['stl'='']` [MGL command]
Draws Bezier-like curve from point *p1* to *p2* using line style *stl*. At this tangent is codirected with *d1*, *d2* and proportional to its amplitude. Parameter *num* define the "quality" of the curve. If *num*=2 then the straight line will be drawn in all coordinate system (independently on transformation formulas, see Section 3.3.2 [Curved coordinates], page 22). Contrary, for large values (for example, =100) the spline like Bezier curve will be drawn in corresponding coordinate system. Curve will be drawn even if it lies out of bounding box.

`face x1 y1 x2 y2 x3 y3 x4 y4 ['stl'='']` [MGL command]
`face x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4 ['stl'='']` [MGL command]
Draws the solid quadrangle (face) with vertexes *p1*, *p2*, *p3*, *p4* and with color(s) *stl*. At this colors can be the same for all vertexes or different if all 4 colors are specified for each vertex. Face will be drawn even if it lies out of bounding box.

`rect x1 y1 x2 y2 ['stl'='']` [MGL command]
`rect x1 y1 z1 x2 y2 z2 ['stl'='']` [MGL command]
Draws the solid rectangle (face) with vertexes $\{x1, y1, z1\}$ and $\{x2, y2, z2\}$ with color *stl*. At this colors can be the same for all vertexes or separately if all 4 colors are specified for each vertex. Face will be drawn even if it lies out of bounding box.

`facex x0 y0 z0 wy wz ['stl'='' d1=0 d2=0]` [MGL command]
`facey x0 y0 z0 wx wz ['stl'='' d1=0 d2=0]` [MGL command]
`facez x0 y0 z0 wx wy ['stl'='' d1=0 d2=0]` [MGL command]
Draws the solid rectangle (face) perpendicular to [x,y,z]-axis correspondingly at position $\{x0, y0, z0\}$ with color *stl* and with widths *wx*, *wy*, *wz* along corresponding directions. At this colors can be the same for all vertexes or separately if all 4 colors are specified for each vertex. Parameters *d1*!=0, *d2*!=0 set additional shift of the last vertex (i.e. to draw quadrangle). Face will be drawn even if it lies out of bounding box.

`sphere x0 y0 r ['col'='r']` [MGL command]
`sphere x0 y0 z0 r ['col'='r']` [MGL command]
Draw the sphere with radius *r* and center at point *p*={*x0*, *y0*, *z0*} and color *stl*.

`drop x0 y0 dx dy r ['col'='r' sh=1 asp=1]` [MGL command]
`drop x0 y0 z0 dx dy dz r ['col'='r' sh=1 asp=1]` [MGL command]
Draw the drop with radius *r* at point *p* elongated in direction *d* and with color *col*. Parameter *shift* set the degree of drop oblongness: '`0`' is sphere, '`1`' is maximally oblongness drop. Parameter *ap* set relative width of the drop (this is analogue of "ellipticity" for the sphere).

`cone x1 y1 z1 x2 y2 z2 r1 [r2=-1 'stl'='']` [MGL command]
Draw tube (or truncated cone if *edge*=`false`) between points *p1*, *p2* with radius at the edges *r1*, *r2*. If *r2*<0 then it is supposed that *r2*=*r1*. The cone color is defined by string *stl*. Parameter *stl* can contain:

- '`@`' for drawing edges;
- '`#`' for wired cones;
- '`t`' for drawing tubes/cylinder instead of cones/prisms;
- '`4`', '`6`', '`8`' for drawing square, hex- or octo-prism instead of cones.

`circle x0 y0 r ['col'='r']` [MGL command]
`circle x0 y0 z0 r ['col'='r']` [MGL command]
Draw the circle with radius *r* and center at point *p*={*x0*, *y0*, *z0*}. Parameter *col* may contain

- colors for filling and boundary (second one if style '`@`' is used, black color is used by default);
- '`#`' for wire figure (boundary only);
- '`@`' for filling and boundary.

`ellipse x1 y1 x2 y2 r ['col'='r']` [MGL command]
`ellipse x1 y1 z1 x2 y2 z2 r ['col'='r']` [MGL command]
Draw the ellipse with radius *r* and focal points *p1*, *p2*. Parameter *col* may contain

- colors for filling and boundary (second one if style '`@`' is used, black color is used by default);
- '`#`' for wire figure (boundary only);
- '`@`' for filling and boundary.

`rhomb x1 y1 x2 y2 r ['col'='r']` [MGL command]
`rhomb x1 y1 z1 x2 y2 z2 r ['col'='r']` [MGL command]
Draw the rhombus with width *r* and edge points *p1*, *p2*. Parameter *col* may contain

- colors for filling and boundary (second one if style '`@`' is used, black color is used by default);
- '`#`' for wire figure (boundary only);
- '`@`' for filling and boundary.

`arc x0 y0 x1 y1 a` ['*col*'='*r*'] [MGL command]
`arc x0 y0 z0 x1 y1 a` ['*col*'='*r*'] [MGL command]
`arc x0 y0 z0 xa ya za x1 y1 z1 a` ['*col*'='*r*'] [MGL command]
Draw the arc around axis *pa* (default is z-axis *pa*={0,0,1}) with center at *p0* and starting from point *p1*. Parameter *a* set the angle of arc in degree. Parameter *col* may contain color of the arc and arrow style for arc edges.

`polygon x0 y0 x1 y1 num` ['*col*'='*r*'] [MGL command]
`polygon x0 y0 z0 x1 y1 z1 num` ['*col*'='*r*'] [MGL command]
Draw the polygon with *num* edges starting from *p1*. The center of polygon is located in *p0*. Parameter *col* may contain

- colors for filling and boundary (second one if style '`@`' is used, black color is used by default);
- '`#`' for wire figure (boundary only);
- '`@`' for filling and boundary.

`logo` *'fname'* [*smooth=off*] [MGL command]
Draw bitmap (logo) along whole axis range, which can be changed by Section 2.7 [Command options], page 15. Bitmap can be loaded from file or specified as RGBA values for pixels. Parameter *smooth* set to draw bitmap without or with color interpolation.

3.8 Text printing

These functions draw the text. There are functions for drawing text in arbitrary place, in arbitrary direction and along arbitrary curve. MathGL can use arbitrary font-faces and parse many TeX commands (for more details see Section 2.5 [Font styles], page 13). All these functions have 2 variant: for printing 8-bit text (`char *`) and for printing Unicode text (`wchar_t *`). In first case the conversion into the current locale is used. So sometimes you need to specify it by `setlocale()` function. The *size* argument control the size of text: if positive it give the value, if negative it give the value relative to `SetFontSize()`. The font type (STIX, arial, courier, times and so on) can be selected by function LoadFont(). See Section 3.2.6 [Font settings], page 20.

The font parameters are described by string. This string may set the text color '`wkrgbcymhRGBCYMHW`' (see Section 2.2 [Color styles], page 9). Starting from MathGL v.2.3, you can set color gradient for text (see Section 2.4 [Color scheme], page 11). Also, after delimiter symbol '`:`', it can contain characters of font type ('`rbiwou`') and/or align ('`LRCTV`') specification. The font types are: '`r`' – roman (or regular) font, '`i`' – italic style, '`b`' – bold style, '`w`' – wired style, '`o`' – over-lined text, '`u`' – underlined text. By default roman font is used. The align types are: '`L`' – align left (default), '`C`' – align center, '`R`' – align right, '`T`' – align under, '`V`' – align center vertical. For example, string '`b:iC`' correspond to italic font style for centered text which printed by blue color.

If string contains symbols '`aA`' then text is printed at absolute position {*x*, *y*} (supposed to be in range [0,1]) of picture (for '`A`') or subplot/inplot (for '`a`'). If string contains symbol '`@`' then box around text is drawn.

See Section 5.2.7 [Text features], page 81, for sample code and picture.

`text x y` *'text'* [*'fnt'*=" `size=-1`] [MGL command]
`text x y z` *'text'* [*'fnt'*=" `size=-1`] [MGL command]
The function plots the string *text* at position *p* with fonts specifying by the criteria *fnt*. The size of font is set by *size* parameter (default is `-1`).

`text x y dx dy` *'text'* [*'fnt'*=*':L'* `size=-1`] [MGL command]
`text x y z dx dy dz` *'text'* [*'fnt'*=*':L'* `size=-1`] [MGL command]
The function plots the string *text* at position *p* along direction *d* with specified *size*. Parameter *fnt* set text style and text position: under ('`T`') or above ('`t`') the line.

`fgets x y` *'fname'* [`n=0` *'fnt'*=" `size=-1.4`] [MGL command]
`fgets x y z` *'fname'* [`n=0` *'fnt'*=" `size=-1.4`] [MGL command]
Draws unrotated *n*-th line of file *fname* at position {*x*,*y*,*z*} with specified *size*. By default parameters from [font], page 20, command are used.

`text` *ydat 'text'* [*'fnt'*="] [MGL command]
`text` *xdat ydat 'text'* [*'fnt'*="] [MGL command]
`text` *xdat ydat zdat 'text'* [*'fnt'*="] [MGL command]
The function draws *text* along the curve between points {*x*[i], *y*[i], *z*[i]} by font style *fnt*. The string *fnt* may contain symbols '`t`' for printing the text under the curve (default), or '`T`' for printing the text under the curve. The sizes of 1st dimension must be equal for all arrays `x.nx=y.nx=z.nx`. If array *x* is not specified then its an automatic array is used with values equidistantly distributed in x-axis range (see Section 3.3.1 [Ranges (bounding box)], page 21). If array *z* is not specified then *z*[i] equal to minimal z-axis value is used. String *opt* contain command options (see Section 2.7 [Command options], page 15).

3.9 Axis and Colorbar

These functions draw the "things for measuring", like axis with ticks, colorbar with ticks, grid along axis, bounding box and labels for axis. For more information see Section 3.3 [Axis settings], page 21.

`axis` [*'dir'*=*'xyz' 'stl'*="] [MGL command]
Draws axes with ticks (see Section 3.3 [Axis settings], page 21). Parameter *dir* may contain:

- '`xyz`' for drawing axis in corresponding direction;
- '`XYZ`' for drawing axis in corresponding direction but with inverted positions of labels;
- '`~`' or '`_`' for disabling tick labels;
- '`U`' for disabling rotation of tick labels;
- '`^`' for inverting default axis origin;
- '`!`' for disabling ticks tuning (see [tuneticks], page 24);
- '`AKDTVISO`' for drawing arrow at the end of axis;
- '`a`' for forced adjusting of axis ticks;
- '`f`' for printing ticks labels in fixed format;

- '`E`' for using '`E`' instead of '`e`' in ticks labels;
- '`F`' for printing ticks labels in LaTeX format;
- '`+`' for printing '`+`' for positive ticks;
- '`-`' for printing usual '`-`' in ticks labels;
- '`0123456789`' for precision at printing ticks labels.

Styles of ticks and axis can be overrided by using *stl* string. Option `value` set the manual rotation angle for the ticks. See Section 5.2.2 [Axis and ticks], page 72, for sample code and picture.

`colorbar` ['*sch*'="] [MGL command]

Draws colorbar. Parameter *sch* may contain:

- color scheme (see Section 2.4 [Color scheme], page 11);
- '`<>^_`' for positioning at left, at right, at top or at bottom correspondingly;
- '`I`' for positioning near bounding (by default, is positioned at edges of subplot);
- '`A`' for using absolute coordinates;
- '`~`' for disabling tick labels.
- '`!`' for disabling ticks tuning (see [tuneticks], page 24);
- '`f`' for printing ticks labels in fixed format;
- '`E`' for using '`E`' instead of '`e`' in ticks labels;
- '`F`' for printing ticks labels in LaTeX format;
- '`+`' for printing '`+`' for positive ticks;
- '`-`' for printing usual '`-`' in ticks labels;
- '`0123456789`' for precision at printing ticks labels.

See Section 5.2.4 [Colorbars], page 77, for sample code and picture.

`colorbar` *vdat* ['*sch*'="] [MGL command]

The same as previous but with sharp colors *sch* (current palette if `sch=""`) for values *v*. See Section 5.6.14 [ContD sample], page 125, for sample code and picture.

`colorbar` '*sch*' `x y` [`w=1 h=1`] [MGL command]

The same as first one but at arbitrary position of subplot {*x*, *y*} (supposed to be in range [0,1]). Parameters *w*, *h* set the relative width and height of the colorbar.

`colorbar` *vdat* '*sch*' `x y` [`w=1 h=1`] [MGL command]

The same as previous but with sharp colors *sch* (current palette if `sch=""`) for values *v*. See Section 5.6.14 [ContD sample], page 125, for sample code and picture.

`grid` ['*dir*'='xyz' '*pen*'='B'] [MGL command]

Draws grid lines perpendicular to direction determined by string parameter *dir*. If *dir* contain '`!`' then grid lines will be drawn at coordinates of subticks also. The step of grid lines is the same as tick step for [axis], page 31. The style of lines is determined by *pen* parameter (default value is dark blue solid line '`B-`').

`box` [*'stl'='k'* `ticks=on`] [MGL command]
Draws bounding box outside the plotting volume with color *col*. If *col* contain '`@`' then filled faces are drawn. At this first color is used for faces (default is light yellow), last one for edges. See Section 5.2.5 [Bounding box], page 79, for sample code and picture.

`xlabel` *'text'* [`pos=1`] [MGL command]
`ylabel` *'text'* [`pos=1`] [MGL command]
`zlabel` *'text'* [`pos=1`] [MGL command]
`tlabel` *'text'* [`pos=1`] [MGL command]
Prints the label *text* for axis *dir*='`x`','`y`','`z`','`t`' (here '`t`' is "ternary" axis $t = 1-x-y$). The position of label is determined by *pos* parameter. If *pos*=0 then label is printed at the center of axis. If *pos*>0 then label is printed at the maximum of axis. If *pos*<0 then label is printed at the minimum of axis. Option `value` set additional shifting of the label. See Section 3.8 [Text printing], page 30.

3.10 Legend

These functions draw legend to the graph (useful for Section 3.11 [1D plotting], page 34). Legend entry is a pair of strings: one for style of the line, another one with description text (with included TeX parsing). The arrays of strings may be used directly or by accumulating first to the internal arrays (by function [addlegend], page 34) and further plotting it. The position of the legend can be selected automatic or manually (even out of bounding box). Parameters *fnt* and *size* specify the font style and size (see Section 3.2.6 [Font settings], page 20). Parameter *llen* set the relative width of the line sample and the text indent. If line style string for entry is empty then the corresponding text is printed without indent. Parameter *fnt* may contain:

- font style for legend text;
- '`A`' for positioning in absolute coordinates;
- '`^`' for positioning outside of specified point;
- '`#`' for drawing box around legend;
- '`-`' for arranging legend entries horizontally;
- colors for face (1st one), for border (2nd one) and for text (last one). If less than 3 colors are specified then the color for border is black (for 2 and less colors), and the color for face is white (for 1 or none colors).

See Section 5.2.8 [Legend sample], page 83, for sample code and picture.

`legend` [`pos=3` *'fnt'='#'*] [MGL command]
Draws legend of accumulated legend entries by font *fnt* with *size*. Parameter *pos* sets the position of the legend: '`0`' is bottom left corner, '`1`' is bottom right corner, '`2`' is top left corner, '`3`' is top right corner (is default). Option `value` set the space between line samples and text (default is 0.1).

`legend x y` [*'fnt'='#'*] [MGL command]
Draws legend of accumulated legend entries by font *fnt* with *size*. Position of legend is determined by parameter *x*, *y* which supposed to be normalized to interval [0,1]. Option `value` set the space between line samples and text (default is 0.1).

`addlegend` *'text' 'stl'* [MGL command]
Adds string *text* to internal legend accumulator. The style of described line and mark is specified in string *style* (see Section 2.3 [Line styles], page 9).

`clearlegend` [MGL command]
Clears saved legend strings.

`legendmarks val` [MGL command]
Set the number of marks in the legend. By default 1 mark is used.

3.11 1D plotting

These functions perform plotting of 1D data. 1D means that data depended from only 1 parameter like parametric curve {x[i],y[i],z[i]}, i=1...n. By default (if absent) values of x[i] are equidistantly distributed in axis range, and z[i] equal to minimal z-axis value. The plots are drawn for each row if one of the data is the matrix. By any case the sizes of 1st dimension **must be equal** for all arrays `x.nx=y.nx=z.nx`.

String *pen* specifies the color and style of line and marks (see Section 2.3 [Line styles], page 9). By default (`pen=""`) solid line with color from palette is used (see Section 3.2.7 [Palette and colors], page 20). Symbol '`!`' set to use new color from palette for each point (not for each curve, as default). String *opt* contain command options (see Section 2.7 [Command options], page 15). See Section 5.5 [1D samples], page 92, for sample code and picture.

`plot` *ydat* ['stl'=''] [MGL command]
`plot` *xdat ydat* ['stl'=''] [MGL command]
`plot` *xdat ydat zdat* ['stl'=''] [MGL command]
These functions draw continuous lines between points {x[i], y[i], z[i]}. See also [area], page 35, [step], page 34, [stem], page 35, [tube], page 39, [mark], page 37, [error], page 37, [belt], page 40, [tens], page 34, [tape], page 35. See Section 5.5.1 [Plot sample], page 92, for sample code and picture.

`radar` *adat* ['stl'=''] [MGL command]
This functions draws radar chart which is continuous lines between points located on an radial lines (like plot in Polar coordinates). Option `value` set the additional shift of data (i.e. the data *a*`+value` is used instead of *a*). If `value<0` then `r=max(0, -min(value)`. If *pen* containt '`#`' symbol then `"grid"` (radial lines and circle for r) is drawn. See also [plot], page 34. See Section 5.5.2 [Radar sample], page 93, for sample code and picture.

`step` *ydat* ['stl'=''] [MGL command]
`step` *xdat ydat* ['stl'=''] [MGL command]
`step` *xdat ydat zdat* ['stl'=''] [MGL command]
These functions draw continuous stairs for points to axis plane. See also [plot], page 34, [stem], page 35, [tile], page 40, [boxs], page 40. See Section 5.5.3 [Step sample], page 94, for sample code and picture.

`tens` *ydat cdat* ['stl'=''] [MGL command]
`tens` *xdat ydat cdat* ['stl'=''] [MGL command]

`tens` *xdat ydat zdat cdat* ['*stl*'=''] [MGL command]
These functions draw continuous lines between points {*x*[i], *y*[i], *z*[i]} with color defined by the special array *c*[i] (look like tension plot). String *pen* specifies the color scheme (see Section 2.4 [Color scheme], page 11) and style and/or width of line (see Section 2.3 [Line styles], page 9). See also [plot], page 34, [mesh], page 40, [fall], page 40. See Section 5.5.4 [Tens sample], page 95, for sample code and picture.

`tape` *ydat* ['*stl*'=''] [MGL command]
`tape` *xdat ydat* ['*stl*'=''] [MGL command]
`tape` *xdat ydat zdat* ['*stl*'=''] [MGL command]
These functions draw tapes of normals for curve between points {*x*[i], *y*[i], *z*[i]}. Initial tape(s) was selected in x-y plane (for '`x`' in *pen*) and/or y-z plane (for '`x`' in *pen*). The width of tape is proportional to [barwidth], page 19, and can be changed by option `value`. See also [plot], page 34, [flow], page 48, [barwidth], page 19. See Section 5.5.21 [Tape sample], page 111, for sample code and picture.

`area` *ydat* ['*stl*'=''] [MGL command]
`area` *xdat ydat* ['*stl*'=''] [MGL command]
`area` *xdat ydat zdat* ['*stl*'=''] [MGL command]
These functions draw continuous lines between points and fills it to axis plane. Also you can use gradient filling if number of specified colors is equal to 2*number of curves. See also [plot], page 34, [bars], page 35, [stem], page 35, [region], page 35. See Section 5.5.5 [Area sample], page 96, for sample code and picture.

`region` *ydat1 ydat2* ['*stl*'=''] [MGL command]
`region` *xdat ydat1 ydat2* ['*stl*'=''] [MGL command]
`region` *xdat1 ydat1 xdat2 ydat2* ['*stl*'=''] [MGL command]
`region` *xdat1 ydat1 zdat1 xdat2 ydat2 zdat2* ['*stl*'=''] [MGL command]
These functions fill area between 2 curves. Dimensions of arrays *y1* and *y2* must be equal. Also you can use gradient filling if number of specified colors is equal to 2*number of curves. If for 2D version *pen* contain symbol '`i`' then only area with y1<y<y2 will be filled else the area with y2<y<y1 will be filled too. See also [area], page 35, [bars], page 35, [stem], page 35. See Section 5.5.6 [Region sample], page 97, for sample code and picture.

`stem` *ydat* ['*stl*'=''] [MGL command]
`stem` *xdat ydat* ['*stl*'=''] [MGL command]
`stem` *xdat ydat zdat* ['*stl*'=''] [MGL command]
These functions draw vertical lines from points to axis plane. See also [area], page 35, [bars], page 35, [plot], page 34, [mark], page 37. See Section 5.5.7 [Stem sample], page 98, for sample code and picture.

`bars` *ydat* ['*stl*'=''] [MGL command]
`bars` *xdat ydat* ['*stl*'=''] [MGL command]
`bars` *xdat ydat zdat* ['*stl*'=''] [MGL command]
These functions draw vertical bars from points to axis plane. If string *pen* contain symbol '`a`' then lines are drawn one above another (like summation). If string contain symbol '`f`' then waterfall chart is drawn for determining the cumulative effect of sequentially introduced positive or negative values. You can give different colors for

positive and negative values if number of specified colors is equal to 2*number of curves. If *pen* contain '<', '^' or '>' then boxes will be aligned left, right or centered at its x-coordinates. See also [barh], page 36, [cones], page 36, [area], page 35, [stem], page 35, [chart], page 36, [barwidth], page 19. See Section 5.5.8 [Bars sample], page 99, for sample code and picture.

`barh` *vdat* ['stl'=''] [MGL command]
`barh` *ydat vdat* ['stl'=''] [MGL command]
These functions draw horizontal bars from points to axis plane. If string contain symbol 'a' then lines are drawn one above another (like summation). If string contain symbol 'f' then waterfall chart is drawn for determining the cumulative effect of sequentially introduced positive or negative values. You can give different colors for positive and negative values if number of specified colors is equal to 2*number of curves. If *pen* contain '<', '^' or '>' then boxes will be aligned left, right or centered at its x-coordinates. See also [bars], page 35, [barwidth], page 19. See Section 5.5.9 [Barh sample], page 100, for sample code and picture.

`cones` *ydat* ['stl'=''] [MGL command]
`cones` *xdat ydat* ['stl'=''] [MGL command]
`cones` *xdat ydat zdat* ['stl'=''] [MGL command]
These functions draw cones from points to axis plane. If string contain symbol 'a' then cones are drawn one above another (like summation). You can give different colors for positive and negative values if number of specified colors is equal to 2*number of curves. Parameter *pen* can contain:

- '@' for drawing edges;
- '#' for wired cones;
- 't' for drawing tubes/cylinders instead of cones/prisms;
- '4', '6', '8' for drawing square, hex- or octo-prism instead of cones;
- '<', '^' or '>' for aligning boxes left, right or centering them at its x-coordinates.

See also [bars], page 35, [cone], page 29, [barwidth], page 19. See Section 5.5.10 [Cones sample], page 101, for sample code and picture.

`chart` *adat* ['col'=''] [MGL command]
The function draws colored stripes (boxes) for data in array *a*. The number of stripes is equal to the number of rows in *a* (equal to *a.ny*). The color of each next stripe is cyclically changed from colors specified in string *col* or in palette Pal (see Section 3.2.7 [Palette and colors], page 20). Spaces in colors denote transparent "color" (i.e. corresponding stripe(s) are not drawn). The stripe width is proportional to value of element in *a*. Chart is plotted only for data with non-negative elements. If string *col* have symbol '#' then black border lines are drawn. The most nice form the chart have in 3d (after rotation of coordinates) or in cylindrical coordinates (becomes so called Pie chart). See Section 5.5.11 [Chart sample], page 102, for sample code and picture.

`boxplot` *adat* ['stl'=''] [MGL command]
`boxplot` *xdat adat* ['stl'=''] [MGL command]
These functions draw boxplot (also known as a box-and-whisker diagram) at points *x*[i]. This is five-number summaries of data *a*[i,j] (minimum, lower quartile (Q1),

median (Q2), upper quartile (Q3) and maximum) along second (j-th) direction. If *pen* contain '<', '^' or '>' then boxes will be aligned left, right or centered at its x-coordinates. See also [plot], page 34, [error], page 37, [bars], page 35, [barwidth], page 19. See Section 5.5.12 [BoxPlot sample], page 103, for sample code and picture.

`candle` *vdat1* ['stl'=''] [MGL command]
`candle` *vdat1 vdat2* ['stl'=''] [MGL command]
`candle` *vdat1 ydat1 ydat2* ['stl'=''] [MGL command]
`candle` *vdat1 vdat2 ydat1 ydat2* ['stl'=''] [MGL command]
`candle` *xdat vdat1 vdat2 ydat1 ydat2* ['stl'=''] [MGL command]

These functions draw candlestick chart at points x[i]. This is a combination of a line-chart and a bar-chart, in that each bar represents the range of price movement over a given time interval. Wire (or white) candle correspond to price growth *v1*[i]<*v2*[i], opposite case – solid (or dark) candle. You can give different colors for growth and decrease values if number of specified colors is equal to 2. If *pen* contain '#' then the wire candle will be used even for 2-color scheme. "Shadows" show the minimal *y1* and maximal *y2* prices. If *v2* is absent then it is determined as *v2*[i]=*v1*[i+1]. See also [plot], page 34, [bars], page 35, [ohlc], page 37, [barwidth], page 19. See Section 5.5.13 [Candle sample], page 104, for sample code and picture.

`ohlc` *odat hdat ldat cdat* ['stl'=''] [MGL command]
`ohlc` *xdat odat hdat ldat cdat* ['stl'=''] [MGL command]

These functions draw Open-High-Low-Close diagram. This diagram show vertical line for between maximal(high *h*) and minimal(low *l*) values, as well as horizontal lines before/after vertical line for initial(open *o*)/final(close *c*) values of some process (usually price). You can give different colors for up and down values (when closing values higher or not as in previous point) if number of specified colors is equal to 2*number of curves. See also [candle], page 37, [plot], page 34, [barwidth], page 19. See Section 5.5.14 [OHLC sample], page 104, for sample code and picture.

`error` *ydat yerr* ['stl'=''] [MGL command]
`error` *xdat ydat yerr* ['stl'=''] [MGL command]
`error` *xdat ydat xerr yerr* ['stl'=''] [MGL command]

These functions draw error boxes {*ex*[i], *ey*[i]} at points {x[i], y[i]}. This can be useful, for example, in experimental points, or to show numeric error or some estimations and so on. If string *pen* contain symbol '@' than large semitransparent mark is used instead of error box. See also [plot], page 34, [mark], page 37. See Section 5.5.15 [Error sample], page 105, for sample code and picture.

`mark` *ydat rdat* ['stl'=''] [MGL command]
`mark` *xdat ydat rdat* ['stl'=''] [MGL command]
`mark` *xdat ydat zdat rdat* ['stl'=''] [MGL command]

These functions draw marks with size *r*[i]*[marksize], page 19, at points {x[i], y[i], z[i]}. If you need to draw markers of the same size then you can use [plot], page 34, function with empty line style ' '. For markers with size in axis range use [error], page 37, with style '@'. See also [plot], page 34, [textmark], page 38, [error], page 37, [stem], page 35. See Section 5.5.16 [Mark sample], page 107, for sample code and picture.

`textmark` *ydat 'txt'* ['*stl*'=''] [MGL command]
`textmark` *ydat rdat 'txt'* ['*stl*'=''] [MGL command]
`textmark` *xdat ydat rdat 'txt'* ['*stl*'=''] [MGL command]
`textmark` *xdat ydat zdat rdat 'txt'* ['*stl*'=''] [MGL command]

These functions draw string *txt* as marks with size proportional to *r*[i]**marksize* at points {*x*[i], *y*[i], *z*[i]}. By default (if omitted) *r*[i]=1. See also [plot], page 34, [mark], page 37, [stem], page 35. See Section 5.5.17 [TextMark sample], page 108, for sample code and picture.

`label` *ydat 'txt'* ['*stl*'=''] [MGL command]
`label` *xdat ydat 'txt'* ['*stl*'=''] [MGL command]
`label` *xdat ydat zdat 'txt'* ['*stl*'=''] [MGL command]

These functions draw string *txt* at points {*x*[i], *y*[i], *z*[i]}. If string *txt* contain '`%x`', '`%y`', '`%z`' or '`%n`' then it will be replaced by the value of x-,y-,z-coordinate of the point or its index. String *fnt* may contain:

- Section 2.5 [Font styles], page 13;
- '`f`' for fixed format of printed numbers;
- '`E`' for using '`E`' instead of '`e`';
- '`F`' for printing in LaTeX format;
- '`+`' for printing '`+`' for positive numbers;
- '`-`' for printing usual '`-`';
- '`0123456789`' for precision at printing numbers.

See also [plot], page 34, [mark], page 37, [textmark], page 38, [table], page 38. See Section 5.5.18 [Label sample], page 108, for sample code and picture.

`table` *vdat 'txt'* ['*stl*'='#'] [MGL command]
`table` *x y vdat 'txt'* ['*stl*'='#'] [MGL command]

These functions draw table with values of *val* and captions from string *txt* (separated by newline symbol '`\n`') at points {x, *y*} (default at {0,0}) related to current subplot. String *fnt* may contain:

- Section 2.5 [Font styles], page 13;
- '`#`' for drawing cell borders;
- '`|`' for limiting table widh by subplot one (equal to option '`value 1`');
- '`=`' for equal width of all cells;
- '`f`' for fixed format of printed numbers;
- '`E`' for using '`E`' instead of '`e`';
- '`F`' for printing in LaTeX format;
- '`+`' for printing '`+`' for positive numbers;
- '`-`' for printing usual '`-`';
- '`0123456789`' for precision at printing numbers.

Option `value` set the width of the table (default is 1). See also [plot], page 34, [label], page 38. See Section 5.5.19 [Table sample], page 109, for sample code and picture.

`tube` *ydat rdat* ['stl'=''] [MGL command]
`tube` *ydat* `rval` ['stl'=''] [MGL command]
`tube` *xdat ydat rdat* ['stl'=''] [MGL command]
`tube` *xdat ydat* `rval` ['stl'=''] [MGL command]
`tube` *xdat ydat zdat rdat* ['stl'=''] [MGL command]
`tube` *xdat ydat zdat* `rval` ['stl'=''] [MGL command]

These functions draw the tube with variable radius r[i] along the curve between points {x[i], y[i], z[i]}. See also [plot], page 34. See Section 5.5.20 [Tube sample], page 110, for sample code and picture.

`torus` *rdat zdat* ['stl'=''] [MGL command]

These functions draw surface which is result of curve {r, z} rotation around axis. If string *pen* contain symbols '`x`' or '`z`' then rotation axis will be set to specified direction (default is '`y`'). If string *pen* have symbol '`#`' then wire plot is produced. If string *pen* have symbol '`.`' then plot by dots is produced. See also [plot], page 34, [axial], page 42. See Section 5.5.22 [Torus sample], page 112, for sample code and picture.

`lamerey x0` *ydat* ['stl'=''] [MGL command]
`lamerey x0` *'y(x)'* ['stl'=''] [MGL command]

These functions draw Lamerey diagram for mapping x_new = y(x_old) starting from point *x0*. String *stl* may contain line style, symbol '`v`' for drawing arrows, symbol '`~`' for disabling first segment. Option `value` set the number of segments to be drawn (default is 20). See also [plot], page 34, [fplot], page 50, [bifurcation], page 39, [pmap], page 39. See Section 5.5.23 [Lamerey sample], page 113, for sample code and picture.

`bifurcation dx` *ydat* ['stl'=''] [MGL command]
`bifurcation dx` *'y(x)'* ['stl'=''] [MGL command]

These functions draw bifurcation diagram for mapping x_new = y(x_old). Parameter *dx* set the accuracy along x-direction. String *stl* set color. Option `value` set the number of stationary points (default is 1024). See also [plot], page 34, [fplot], page 50, [lamerey], page 39. See Section 5.5.24 [Bifurcation sample], page 114, for sample code and picture.

`pmap` *ydat sdat* ['stl'=''] [MGL command]
`pmap` *xdat ydat sdat* ['stl'=''] [MGL command]
`pmap` *xdat ydat zdat sdat* ['stl'=''] [MGL command]

These functions draw Poincare map for curve {x, y, z} at surface s=0. Basically, it show intersections of the curve and the surface. String *stl* set the style of marks. See also [plot], page 34, [mark], page 37, [lamerey], page 39. See Section 5.5.25 [Pmap sample], page 114, for sample code and picture.

3.12 2D plotting

These functions perform plotting of 2D data. 2D means that data depend from 2 independent parameters like matrix $f(x_i, y_j), i = 1...n, j = 1...m$. By default (if absent) values of x, y are equidistantly distributed in axis range. The plots are drawn for each z slice of the data. The minor dimensions of arrays x, y, z should be equal `x.nx=z.nx && y.nx=z.ny` or `x.nx=y.nx=z.nx && x.ny=y.ny=z.ny`. Arrays x and y can be vectors (not matrices as z). String *sch* sets the color scheme (see Section 2.4 [Color scheme], page 11) for plot. String *opt*

contain command options (see Section 2.7 [Command options], page 15). See Section 5.6 [2D samples], page 115, for sample code and picture.

`surf` *zdat* ['*sch*'=''] [MGL command]
`surf` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`.`' then plot by dots is produced. See also [mesh], page 40, [dens], page 41, [belt], page 40, [tile], page 40, [boxs], page 40, [surfc], page 44, [surfa], page 45. See Section 5.6.1 [Surf sample], page 115, for sample code and picture.

`mesh` *zdat* ['*sch*'=''] [MGL command]
`mesh` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws mesh lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. See also [surf], page 40, [fall], page 40, [meshnum], page 19, [cont], page 41, [tens], page 34. See Section 5.6.5 [Mesh sample], page 118, for sample code and picture.

`fall` *zdat* ['*sch*'=''] [MGL command]
`fall` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws fall lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. This plot can be used for plotting several curves shifted in depth one from another. If *sch* contain '`x`' then lines are drawn along x-direction else (by default) lines are drawn along y-direction. See also [belt], page 40, [mesh], page 40, [tens], page 34, [meshnum], page 19. See Section 5.6.6 [Fall sample], page 119, for sample code and picture.

`belt` *zdat* ['*sch*'=''] [MGL command]
`belt` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws belts for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. This plot can be used as 3d generalization of [plot], page 34). If *sch* contain '`x`' then belts are drawn along x-direction else (by default) belts are drawn along y-direction. See also [fall], page 40, [surf], page 40, [plot], page 34, [meshnum], page 19. See Section 5.6.7 [Belt sample], page 120, for sample code and picture.

`boxs` *zdat* ['*sch*'=''] [MGL command]
`boxs` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws vertical boxes for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. Symbol '`@`' in *sch* set to draw filled boxes. See also [surf], page 40, [dens], page 41, [tile], page 40, [step], page 34. See Section 5.6.8 [Boxs sample], page 120, for sample code and picture.

`tile` *zdat* ['*sch*'=''] [MGL command]
`tile` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws horizontal tiles for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. Such plot can be used as 3d generalization of [step], page 34. See also [surf], page 40, [boxs], page 40, [step], page 34, [tiles], page 46. See Section 5.6.9 [Tile sample], page 121, for sample code and picture.

`dens` *zdat* ['*sch*'=''] [MGL command]
`dens` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws density plot for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z* equal to minimal z-axis value. If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`.`' then plot by dots is produced. See also [surf], page 40, [cont], page 41, [contf], page 41, [boxs], page 40, [tile], page 40, `dens[xyz]`. See Section 5.6.11 [Dens sample], page 122, for sample code and picture.

`cont` *vdat zdat* ['*sch*'=''] [MGL command]
`cont` *vdat xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws contour lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z*=*v*[k], or at *z* equal to minimal z-axis value if *sch* contain symbol '`_`'. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v*. If string *sch* have symbol '`t`' or '`T`' then contour labels *v*[k] will be drawn below (or above) the contours. See also [dens], page 41, [contf], page 41, [contd], page 41, [axial], page 42, `cont[xyz]`. See Section 5.6.12 [Cont sample], page 123, for sample code and picture.

`cont` *zdat* ['*sch*'=''] [MGL command]
`cont` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`contf` *vdat zdat* ['*sch*'=''] [MGL command]
`contf` *vdat xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws solid (or filled) contour lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z*=*v*[k], or at *z* equal to minimal z-axis value if *sch* contain symbol '`_`'. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v* (must be `v.nx>2`). See also [dens], page 41, [cont], page 41, [contd], page 41, `contf[xyz]`. See Section 5.6.13 [ContF sample], page 124, for sample code and picture.

`contf` *zdat* ['*sch*'=''] [MGL command]
`contf` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`contd` *vdat zdat* ['*sch*'=''] [MGL command]
`contd` *vdat xdat ydat zdat* ['*sch*'=''] [MGL command]
The function draws solid (or filled) contour lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z*=*v*[k] (or at *z* equal to minimal z-axis value if *sch* contain symbol '`_`') with manual colors. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v* (must be `v.nx>2`). String *sch* sets the contour colors: the color of k-th contour is determined by character `sch[k%strlen(sch)]`. See also [dens], page 41, [cont], page 41, [contf], page 41. See Section 5.6.14 [ContD sample], page 125, for sample code and picture.

`contd` *zdat* ['*sch*'=''] [MGL command]
`contd` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`contv` *vdat zdat* ['*sch*'="] [MGL command]
`contv` *vdat xdat ydat zdat* ['*sch*'="] [MGL command]
The function draws vertical cylinder (tube) at contour lines for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z*=*v*[k], or at *z* equal to minimal z-axis value if *sch* contain symbol '`_`'. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v*. See also [cont], page 41, [contf], page 41. See Section 5.6.15 [ContV sample], page 126, for sample code and picture.

`contv` *zdat* ['*sch*'="] [MGL command]
`contv` *xdat ydat zdat* ['*sch*'="] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`axial` *vdat zdat* ['*sch*'="] [MGL command]
`axial` *vdat xdat ydat zdat* ['*sch*'="] [MGL command]
The function draws surface which is result of contour plot rotation for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v*. If string *sch* have symbol '`#`' then wire plot is produced. If string *sch* have symbol '`.`' then plot by dots is produced. If string contain symbols '`x`' or '`z`' then rotation axis will be set to specified direction (default is '`y`'). See also [cont], page 41, [contf], page 41, [torus], page 39, [surf3], page 42. See Section 5.6.16 [Axial sample], page 127, for sample code and picture.

`axial` *zdat* ['*sch*'="] [MGL command]
`axial` *xdat ydat zdat* ['*sch*'="] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 3).

`grid2` *zdat* ['*sch*'="] [MGL command]
`grid2` *xdat ydat zdat* ['*sch*'="] [MGL command]
The function draws grid lines for density plot of surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} at *z* equal to minimal z-axis value. See also [dens], page 41, [cont], page 41, [contf], page 41, [grid3], page 44, [meshnum], page 19.

3.13 3D plotting

These functions perform plotting of 3D data. 3D means that data depend from 3 independent parameters like matrix $f(x_i, y_j, z_k), i = 1...n, j = 1...m, k = 1...l$. By default (if absent) values of *x*, *y*, *z* are equidistantly distributed in axis range. The minor dimensions of arrays *x*, *y*, *z*, *a* should be equal `x.nx=a.nx && y.nx=a.ny && z.nz=a.nz` or `x.nx=y.nx=z.nx=a.nx && x.ny=y.ny=z.ny=a.ny && x.nz=y.nz=z.nz=a.nz`. Arrays *x*, *y* and *z* can be vectors (not matrices as *a*). String *sch* sets the color scheme (see Section 2.4 [Color scheme], page 11) for plot. String *opt* contain command options (see Section 2.7 [Command options], page 15). See Section 5.7 [3D samples], page 129, for sample code and picture.

`surf3` *adat* `val` ['*sch*'="] [MGL command]
`surf3` *xdat ydat zdat adat* `val` ['*sch*'="] [MGL command]
The function draws isosurface plot for 3d array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) at *a*(x,y,z)=*val*. If string contain '`#`' then wire plot is produced. If

string *sch* have symbol '`.`' then plot by dots is produced. Note, that there is possibility of incorrect plotting due to uncertainty of cross-section defining if there are two or more isosurface intersections inside one cell. See also [cloud], page 43, [dens3], page 43, [surf3c], page 45, [surf3a], page 45, [axial], page 42. See Section 5.7.1 [Surf3 sample], page 129, for sample code and picture.

`surf3` *adat* ['*sch*'="] [MGL command]
`surf3` *xdat ydat zdat adat* ['*sch*'="] [MGL command]
Draws *num*-th uniformly distributed in color range isosurfaces for 3d data. Here *num* is equal to parameter `value` in options *opt* (default is 3).

`cloud` *adat* ['*sch*'="] [MGL command]
`cloud` *xdat ydat zdat adat* ['*sch*'="] [MGL command]
The function draws cloud plot for 3d data specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]). This plot is a set of cubes with color and transparency proportional to value of *a*. The resulting plot is like cloud – low value is transparent but higher ones are not. The number of plotting cells depend on [meshnum], page 19. If string *sch* contain symbol '`.`' then lower quality plot will produced with much low memory usage. If string *sch* contain symbol '`i`' then transparency will be inversed, i.e. higher become transparent and lower become not transparent. See also [surf3], page 42, [meshnum], page 19. See Section 5.7.5 [Cloud sample], page 132, for sample code and picture.

`dens3` *adat* ['*sch*'=" `sval=-1`] [MGL command]
`dens3` *xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The function draws density plot for 3d data specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]). Density is plotted at slice *sVal* in direction {'`x`', '`y`', '`z`'} if *sch* contain corresponding symbol (by default, '`y`' direction is used). If string *stl* have symbol '`#`' then grid lines are drawn. See also [cont3], page 43, [contf3], page 44, [dens], page 41, [grid3], page 44. See Section 5.7.6 [Dens3 sample], page 132, for sample code and picture.

`cont3` *vdat adat* ['*sch*'=" `sval=-1`] [MGL command]
`cont3` *vdat xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The function draws contour plot for 3d data specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]). Contours are plotted for values specified in array *v* at slice *sVal* in direction {'`x`', '`y`', '`z`'} if *sch* contain corresponding symbol (by default, '`y`' direction is used). If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`t`' or '`T`' then contour labels will be drawn below (or above) the contours. See also [dens3], page 43, [contf3], page 44, [cont], page 41, [grid3], page 44. See Section 5.7.7 [Cont3 sample], page 133, for sample code and picture.

`cont3` *adat* ['*sch*'=" `sval=-1`] [MGL command]
`cont3` *xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`contf3` *vdat adat* ['*sch*'=" `sval=-1`] [MGL command]
`contf3` *vdat xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The function draws solid (or filled) contour plot for 3d data specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]). Contours are plotted for values specified in array *v* at slice *sVal* in direction {'`x`', '`y`', '`z`'} if *sch* contain corresponding symbol (by default, '`y`' direction is used). If string *sch* have symbol '`#`' then grid lines are drawn. See also [dens3], page 43, [cont3], page 43, [contf], page 41, [grid3], page 44. See Section 5.7.8 [ContF3 sample], page 134, for sample code and picture.

`contf3` *adat* ['*sch*'=" `sval=-1`] [MGL command]
`contf3` *xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The same as previous with vector *v* of *num*-th elements equidistantly distributed in color range. Here *num* is equal to parameter `value` in options *opt* (default is 7).

`grid3` *adat* ['*sch*'=" `sval=-1`] [MGL command]
`grid3` *xdat ydat zdat adat* ['*sch*'=" `sval=-1`] [MGL command]
The function draws grid for 3d data specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]). Grid is plotted at slice *sVal* in direction {'`x`', '`y`', '`z`'} if *sch* contain corresponding symbol (by default, '`y`' direction is used). See also [cont3], page 43, [contf3], page 44, [dens3], page 43, [grid2], page 42, [meshnum], page 19.

`beam` *tr g1 g2 adat* `rval` ['*sch*'=" `flag=0 num=3`] [MGL command]
Draws the isosurface for 3d array *a* at constant values of *a*=*val*. This is special kind of plot for *a* specified in accompanied coordinates along curve *tr* with orts *g1*, *g2* and with transverse scale *r*. Variable *flag* is bitwise: '`0x1`' - draw in accompanied (not laboratory) coordinates; '`0x2`' - draw projection to $\rho - z$ plane; '`0x4`' - draw normalized in each slice field. The x-size of data arrays *tr*, *g1*, *g2* must be nx>2. The y-size of data arrays *tr*, *g1*, *g2* and z-size of the data array *a* must be equal. See also [surf3], page 42.

3.14 Dual plotting

These plotting functions draw *two matrix* simultaneously. There are 5 generally different types of data representations: surface or isosurface colored by other data (SurfC, Surf3C), surface or isosurface transpared by other data (SurfA, Surf3A), tiles with variable size (TileS), mapping diagram (Map), STFA diagram (STFA). By default (if absent) values of *x*, *y*, *z* are equidistantly distributed in axis range. The minor dimensions of arrays *x*, *y*, *z*, *c* should be equal. Arrays *x*, *y* (and *z* for `Surf3C, Surf3A`) can be vectors (not matrices as *c*). String *sch* sets the color scheme (see Section 2.4 [Color scheme], page 11) for plot. String *opt* contain command options (see Section 2.7 [Command options], page 15).

`surfc` *zdat cdat* ['*sch*'="] [MGL command]
`surfc` *xdat ydat zdat cdat* ['*sch*'="] [MGL command]
The function draws surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} and color it by matrix *c*[i,j]. If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`.`' then plot by dots is produced. All dimensions of arrays *z* and *c* must be equal. Surface is plotted for each z slice of the data. See also [surf], page 40, [surfa], page 45, [surfca], page 45, [surf3c], page 45. See Section 5.6.2 [SurfC sample], page 116, for sample code and picture.

`surf3c` *adat cdat* `val` ['*sch*'=''] [MGL command]
`surf3c` *xdat ydat zdat adat cdat* `val` ['*sch*'=''] [MGL command]
The function draws isosurface plot for 3d array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) at *a*(x,y,z)=*val*. It is mostly the same as [surf3], page 42, function but the color of isosurface depends on values of array *c*. If string *sch* contain '`#`' then wire plot is produced. If string *sch* have symbol '`.`' then plot by dots is produced. See also [surf3], page 42, [surfc], page 44, [surf3a], page 45, [surf3ca], page 46. See Section 5.7.2 [Surf3C sample], page 130, for sample code and picture.

`surf3c` *adat cdat* ['*sch*'=''] [MGL command]
`surf3c` *xdat ydat zdat adat cdat* ['*sch*'=''] [MGL command]
Draws *num*-th uniformly distributed in color range isosurfaces for 3d data. Here *num* is equal to parameter `value` in options *opt* (default is 3).

`surfa` *zdat cdat* ['*sch*'=''] [MGL command]
`surfa` *xdat ydat zdat cdat* ['*sch*'=''] [MGL command]
The function draws surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]} and transparent it by matrix *c*[i,j]. If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`.`' then plot by dots is produced. All dimensions of arrays *z* and *c* must be equal. Surface is plotted for each z slice of the data. See also [surf], page 40, [surfc], page 44, [surfca], page 45, [surf3a], page 45. See Section 5.6.3 [SurfA sample], page 117, for sample code and picture.

`surf3a` *adat cdat* `val` ['*sch*'=''] [MGL command]
`surf3a` *xdat ydat zdat adat cdat* `val` ['*sch*'=''] [MGL command]
The function draws isosurface plot for 3d array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) at *a*(x,y,z)=*val*. It is mostly the same as [surf3], page 42, function but the transparency of isosurface depends on values of array *c*. If string *sch* contain '`#`' then wire plot is produced. If string *sch* have symbol '`.`' then plot by dots is produced. See also [surf3], page 42, [surfc], page 44, [surf3a], page 45, [surf3ca], page 46. See Section 5.7.3 [Surf3A sample], page 130, for sample code and picture.

`surf3a` *adat cdat* ['*sch*'=''] [MGL command]
`surf3a` *xdat ydat zdat adat cdat* ['*sch*'=''] [MGL command]
Draws *num*-th uniformly distributed in color range isosurfaces for 3d data. At this array *c* can be vector with values of transparency and *num*=*c*.nx. In opposite case *num* is equal to parameter `value` in options *opt* (default is 3).

`surfca` *zdat cdat adat* ['*sch*'=''] [MGL command]
`surfca` *xdat ydat zdat cdat adat* ['*sch*'=''] [MGL command]
The function draws surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}, color it by matrix *c*[i,j] and transparent it by matrix *a*[i,j]. If string *sch* have symbol '`#`' then grid lines are drawn. If string *sch* have symbol '`.`' then plot by dots is produced. All dimensions of arrays *z* and *c* must be equal. Surface is plotted for each z slice of the data. Note, you can use [map], page 46-like coloring if use '`%`' in color scheme. See also [surf], page 40, [surfc], page 44, [surfa], page 45, [surf3ca], page 46. See Section 5.6.4 [SurfCA sample], page 118, for sample code and picture.

`surf3ca` *adat cdat bdat* `val` ['*sch*'="] [MGL command]
`surf3ca` *xdat ydat zdat adat cdat bdat* `val` ['*sch*'="] [MGL command]
The function draws isosurface plot for 3d array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) at *a*(x,y,z)=*val*. It is mostly the same as [surf3], page 42, function but the color and the transparency of isosurface depends on values of array *c* and *b* correspondingly. If string *sch* contain '`#`' then wire plot is produced. If string *sch* have symbol '`.`' then plot by dots is produced. Note, you can use [map], page 46-like coloring if use '`%`' in color scheme. See also [surf3], page 42, [surfca], page 45, [surf3c], page 45, [surf3a], page 45. See Section 5.7.4 [Surf3CA sample], page 131, for sample code and picture.

`surf3ca` *adat cdat bdat* ['*sch*'="] [MGL command]
`surf3ca` *xdat ydat zdat adat cdat bdat* ['*sch*'="] [MGL command]
Draws *num*-th uniformly distributed in color range isosurfaces for 3d data. Here parameter *num* is equal to parameter `value` in options *opt* (default is 3).

`tiles` *zdat rdat* ['*sch*'="] [MGL command]
`tiles` *xdat ydat zdat rdat* ['*sch*'="] [MGL command]
The function draws horizontal tiles for surface specified parametrically {*x*[i,j], *y*[i,j], *z*[i,j]}. It is mostly the same as [tile], page 40, but the size of tiles is determined by *r* array. This is some kind of "transparency" useful for exporting to EPS files. Tiles is plotted for each z slice of the data. See also [surfa], page 45, [tile], page 40. See Section 5.6.10 [TileS sample], page 121, for sample code and picture.

`map` *udat vdat* ['*sch*'="] [MGL command]
`map` *xdat ydat udat vdat* ['*sch*'="] [MGL command]
The function draws mapping plot for matrices {*ax*, *ay* } which parametrically depend on coordinates *x*, *y*. The initial position of the cell (point) is marked by color. Height is proportional to Jacobian(ax,ay). This plot is like Arnold diagram ??? If string *sch* contain symbol '`.`' then the color ball at matrix knots are drawn otherwise face is drawn. See Section 5.9.9 [Mapping visualization], page 157, for sample code and picture.

`stfa` *re im* `dn` ['*sch*'="] [MGL command]
`stfa` *xdat ydat re im* `dn` ['*sch*'="] [MGL command]
Draws spectrogram of complex array *re*+i**im* for Fourier size of *dn* points at plane *z* equal to minimal z-axis value. For example in 1D case, result is density plot of data $res[i,j] = |\sum_d^d nexp(I*j*d)*(re[i*dn+d]+I*im[i*dn+d])|/dn$ with size {int(nx/dn), dn, ny}. At this array *re*, *im* parametrically depend on coordinates *x*, *y*. The size of *re* and *im* must be the same. The minor dimensions of arrays *x*, *y*, *re* should be equal. Arrays *x*, *y* can be vectors (not matrix as *re*). See Section 5.9.8 [STFA sample], page 156, for sample code and picture.

3.15 Vector fields

These functions perform plotting of 2D and 3D vector fields. There are 5 generally different types of vector fields representations: simple vector field (Vect), vectors along the curve (Traj), vector field by dew-drops (Dew), flow threads (Flow, FlowP), flow pipes (Pipe). By default (if absent) values of *x*, *y*, *z* are equidistantly distributed in axis range. The minor

dimensions of arrays *x*, *y*, *z*, *ax* should be equal. The size of *ax*, *ay* and *az* must be equal. Arrays *x*, *y*, *z* can be vectors (not matrices as *ax*). String *sch* sets the color scheme (see Section 2.4 [Color scheme], page 11) for plot. String *opt* contain command options (see Section 2.7 [Command options], page 15).

`traj` *xdat ydat udat vdat* ['*sch*'=''] [MGL command]
`traj` *xdat ydat zdat udat vdat wdat* ['*sch*'=''] [MGL command]
The function draws vectors {*ax*, *ay*, *az*} along a curve {*x*, *y*, *z*}. The length of arrows are proportional to $\sqrt{\{ax^2+ay^2+az^2\}}$. String *pen* specifies the color (see Section 2.3 [Line styles], page 9). By default (`pen=""`) color from palette is used (see Section 3.2.7 [Palette and colors], page 20). Option `value` set the vector length factor (if non-zero) or vector length to be proportional the distance between curve points (if `value=0`). The minor sizes of all arrays must be equal and large 2. The plots are drawn for each row if one of the data is the matrix. See also [vect], page 47. See Section 5.8.3 [Traj sample], page 142, for sample code and picture.

`vect` *udat vdat* ['*sch*'=''] [MGL command]
`vect` *xdat ydat udat vdat* ['*sch*'=''] [MGL command]
The function draws plane vector field plot for the field {*ax*, *ay*} depending parametrically on coordinates *x*, *y* at level *z* equal to minimal z-axis value. The length and color of arrows are proportional to $\sqrt{\{ax^2+ay^2\}}$. The number of arrows depend on [meshnum], page 19. The appearance of the hachures (arrows) can be changed by symbols:

- '`f`' for drawing arrows with fixed lengths,
- '`>`', '`<`' for drawing arrows to or from the cell point (default is centering),
- '`.`' for drawing hachures with dots instead of arrows,
- '`=`' for enabling color gradient along arrows.

See also [flow], page 48, [dew], page 48. See Section 5.8.1 [Vect sample], page 140, for sample code and picture.

`vect` *udat vdat wdat* ['*sch*'=''] [MGL command]
`vect` *xdat ydat zdat udat vdat wdat* ['*sch*'=''] [MGL command]
This is 3D version of the first functions. Here arrays *ax*, *ay*, *az* must be 3-ranged tensors with equal sizes and the length and color of arrows is proportional to $\sqrt{\{ax^2+ay^2+az^2\}}$.

`vect3` *udat vdat wdat* ['*sch*'='' *sval*] [MGL command]
`vect3` *xdat ydat zdat udat vdat wdat* ['*sch*'='' *sval*] [MGL command]
The function draws 3D vector field plot for the field {*ax*, *ay*, *az*} depending parametrically on coordinates *x*, *y*, *z*. Vector field is drawn at slice *sVal* in direction {'`x`', '`y`', '`z`'} if *sch* contain corresponding symbol (by default, '`y`' direction is used). The length and color of arrows are proportional to $\sqrt{\{ax^2+ay^2+az^2\}}$. The number of arrows depend on [meshnum], page 19. The appearance of the hachures (arrows) can be changed by symbols:

- '`f`' for drawing arrows with fixed lengths,
- '`>`', '`<`' for drawing arrows to or from the cell point (default is centering),

- '.' for drawing hachures with dots instead of arrows,
- '=' for enabling color gradient along arrows.

See also [vect], page 47, [flow], page 48, [dew], page 48. See Section 5.8.2 [Vect3 sample], page 141, for sample code and picture.

`dew` *udat vdat* ['*sch*'=''] [MGL command]
`dew` *xdat ydat udat vdat* ['*sch*'=''] [MGL command]
The function draws dew-drops for plane vector field {*ax*, *ay*} depending parametrically on coordinates *x*, *y* at level *z* equal to minimal z-axis value. Note that this is very expensive plot in memory usage and creation time! The color of drops is proportional to $\sqrt{\{ax^2+ay^2\}}$. The number of drops depend on [meshnum], page 19. See also [vect], page 47. See Section 5.8.6 [Dew sample], page 144, for sample code and picture.

`flow` *udat vdat* ['*sch*'=''] [MGL command]
`flow` *xdat ydat udat vdat* ['*sch*'=''] [MGL command]
The function draws flow threads for the plane vector field {*ax*, *ay*} parametrically depending on coordinates *x*, *y* at level *z* equal to minimal z-axis value. Number of threads is proportional to `value` option (default is 5). String *sch* may contain:

- color scheme – up-half (warm) corresponds to normal flow (like attractor), bottom-half (cold) corresponds to inverse flow (like source);
- '#' for starting threads from edges only;
- '*' for starting threads from a 2D array of points inside the data;
- 'v' for drawing arrows on the threads;
- 'x', 'z' for drawing tapes of normals in x-y and y-z planes correspondingly.

See also [pipe], page 49, [vect], page 47, [tape], page 35, [barwidth], page 19. See Section 5.8.4 [Flow sample], page 143, for sample code and picture.

`flow` *udat vdat wdat* ['*sch*'=''] [MGL command]
`flow` *xdat ydat zdat udat vdat wdat* ['*sch*'=''] [MGL command]
This is 3D version of the first functions. Here arrays *ax*, *ay*, *az* must be 3-ranged tensors with equal sizes and the color of line is proportional to $\sqrt{\{ax^2+ay^2+az^2\}}$.

`flow x0 y0` *udat vdat* ['*sch*'=''] [MGL command]
`flow x0 y0` *xdat ydat udat vdat* ['*sch*'=''] [MGL command]
The same as first one ([flow], page 48) but draws single flow thread starting from point *p0*={*x0*,*y0*,*z0*}.

`flow x0 y0 z0` *udat vdat wdat* ['*sch*'=''] [MGL command]
`flow x0 y0 z0` *xdat ydat zdat udat vdat wdat* ['*sch*'=''] [MGL command]
This is 3D version of the previous functions.

`grad` *pdat* ['*sch*'=''] [MGL command]
`grad` *xdat ydat pdat* ['*sch*'=''] [MGL command]
`grad` *xdat ydat zdat pdat* ['*sch*'=''] [MGL command]
The function draws gradient lines for scalar field *phi*[i,j] (or *phi*[i,j,k] in 3d case) specified parametrically {*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]}. Number of lines is proportional to `value` option (default is 5). See also [dens], page 41, [cont], page 41, [flow], page 48.

`pipe` *udat vdat* ['*sch*'=" `r0=0.05`] [MGL command]
`pipe` *xdat ydat udat vdat* ['*sch*'=" `r0=0.05`] [MGL command]
The function draws flow pipes for the plane vector field {*ax*, *ay*} parametrically depending on coordinates *x*, *y* at level *z* equal to minimal z-axis value. Number of pipes is proportional to `value` option (default is 5). If '`#`' symbol is specified then pipes start only from edges of axis range. The color of lines is proportional to $\sqrt{\{ax^2 + ay^2\}}$. Warm color corresponds to normal flow (like attractor). Cold one corresponds to inverse flow (like source). Parameter *r0* set the base pipe radius. If *r0*<0 or symbol '`i`' is specified then pipe radius is inverse proportional to amplitude. The vector field is plotted for each z slice of *ax*, *ay*. See also [flow], page 48, [vect], page 47. See Section 5.8.5 [Pipe sample], page 143, for sample code and picture.

`pipe` *udat vdat wdat* ['*sch*'=" `r0=0.05`] [MGL command]
`pipe` *xdat ydat zdat udat vdat wdat* ['*sch*'=" `r0=0.05`] [MGL command]
This is 3D version of the first functions. Here arrays *ax*, *ay*, *az* must be 3-ranged tensors with equal sizes and the color of line is proportional to $\sqrt{\{ax^2 + ay^2 + az^2\}}$.

3.16 Other plotting

These functions perform miscellaneous plotting. There is unstructured data points plots (Dots), surface reconstruction (Crust), surfaces on the triangular or quadrangular mesh (TriPlot, TriCont, QuadPlot), textual formula plotting (Plots by formula), data plots at edges (Dens[XYZ], Cont[XYZ], ContF[XYZ]). Each type of plotting has similar interface. There are 2 kind of versions which handle the arrays of data and coordinates or only single data array. Parameters of color scheme are specified by the string argument. See Section 2.4 [Color scheme], page 11.

`densx` *dat* ['*sch*'=" `sval=nan`] [MGL command]
`densy` *dat* ['*sch*'=" `sval=nan`] [MGL command]
`densz` *dat* ['*sch*'=" `sval=nan`] [MGL command]
These plotting functions draw density plot in x, y, or z plain. If *a* is a tensor (3-dimensional data) then interpolation to a given *sVal* is performed. These functions are useful for creating projections of the 3D data array to the bounding box. See also [ContXYZ], page 49, [ContFXYZ], page 49, [dens], page 41, Section 3.18 [Data manipulation], page 52. See Section 5.7.9 [Dens projection sample], page 134, for sample code and picture.

`contx` *dat* ['*sch*'=" `sval=nan`] [MGL command]
`conty` *dat* ['*sch*'=" `sval=nan`] [MGL command]
`contz` *dat* ['*sch*'=" `sval=nan`] [MGL command]
These plotting functions draw contour lines in x, y, or z plain. If *a* is a tensor (3-dimensional data) then interpolation to a given *sVal* is performed. These functions are useful for creating projections of the 3D data array to the bounding box. Option `value` set the number of contours. See also [ContFXYZ], page 49, [DensXYZ], page 49, [cont], page 41, Section 3.18 [Data manipulation], page 52. See Section 5.7.10 [Cont projection sample], page 135, for sample code and picture.

`contfx` *dat* ['*sch*'=" `sval=nan`] [MGL command]
`contfy` *dat* ['*sch*'=" `sval=nan`] [MGL command]

`contfz` *dat* ['*sch*'=" `sval=nan`] [MGL command]
: These plotting functions draw solid contours in x, y, or z plain. If *a* is a tensor (3-dimensional data) then interpolation to a given *sVal* is performed. These functions are useful for creating projections of the 3D data array to the bounding box. Option `value` set the number of contours. See also [ContFXYZ], page 49, [DensXYZ], page 49, [cont], page 41, Section 3.18 [Data manipulation], page 52. See Section 5.7.11 [ContF projection sample], page 136, for sample code and picture.

`fplot` '*y*(*x*)' ['*pen*'="] [MGL command]
: Draws command function '`y(x)`' at plane *z* equal to minimal z-axis value, where '`x`' variable is changed in `xrange`. You do not need to create the data arrays to plot it. Option `value` set initial number of points. See also [plot], page 34.

`fplot` '*x*(*t*)' '*y*(*t*)' '*z*(*t*)' ['*pen*'="] [MGL command]
: Draws command parametrical curve {'`x(t)`', '`y(t)`', '`z(t)`'} where '`t`' variable is changed in range [0, 1]. You do not need to create the data arrays to plot it. Option `value` set number of points. See also [plot], page 34.

`fsurf` '*z*(*x*,*y*)' ['*sch*'="] [MGL command]
: Draws command surface for function '`z(x,y)`' where '`x`', '`y`' variable are changed in `xrange, yrange`. You do not need to create the data arrays to plot it. Option `value` set number of points. See also [surf], page 40.

`fsurf` '*x*(*u*,*v*)' '*y*(*u*,*v*)' '*z*(*u*,*v*)' ['*sch*'="] [MGL command]
: Draws command parametrical surface {'`x(u,v)`', '`y(u,v)`', '`z(u,v)`'} where '`u`', '`v`' variable are changed in range [0, 1]. You do not need to create the data arrays to plot it. Option `value` set number of points. See also [surf], page 40.

`triplot` *idat xdat ydat* ['*sch*'="] [MGL command]
`triplot` *idat xdat ydat zdat* ['*sch*'="] [MGL command]
`triplot` *idat xdat ydat zdat cdat* ['*sch*'="] [MGL command]
: The function draws the surface of triangles. Triangle vertexes are set by indexes *id* of data points {*x*[i], *y*[i], *z*[i]}. String *sch* sets the color scheme. If string contain '`#`' then wire plot is produced. First dimensions of *id* must be 3 or greater. Arrays *x*, *y*, *z* must have equal sizes. Parameter *c* set the colors of triangles (if *id*.ny=*c*.nx) or colors of vertexes (if x.nx=*c*.nx). See also [dots], page 51, [crust], page 51, [quadplot], page 51, [triangulation], page 67. See Section 5.7.12 [TriPlot and QuadPlot], page 136, for sample code and picture.

`tricont` *vdat idat xdat ydat zdat cdat* ['*sch*'="] [MGL command]
`tricont` *vdat idat xdat ydat zdat* ['*sch*'="] [MGL command]
`tricont` *idat xdat ydat zdat* ['*sch*'="] [MGL command]
: The function draws contour lines for surface of triangles at *z*=*v*[k] (or at *z* equal to minimal z-axis value if *sch* contain symbol '`_`'). Triangle vertexes are set by indexes *id* of data points {*x*[i], *y*[i], *z*[i]}. Contours are plotted for *z*[i,j]=*v*[k] where *v*[k] are values of data array *v*. If *v* is absent then arrays of option `value` elements equidistantly distributed in color range is used. String *sch* sets the color scheme. Array *c* (if specified) is used for contour coloring. First dimensions of *id* must be 3 or greater. Arrays *x*, *y*, *z* must have equal sizes. Parameter *c* set the colors of triangles

(if *id*.ny=*c*.nx) or colors of vertexes (if x.nx=*c*.nx). See also [triplot], page 50, [cont], page 41, [triangulation], page 67.

`quadplot` *idat xdat ydat* ['*sch*'=''] [MGL command]
`quadplot` *idat xdat ydat zdat* ['*sch*'=''] [MGL command]
`quadplot` *idat xdat ydat zdat cdat* ['*sch*'=''] [MGL command]
The function draws the surface of quadrangles. Quadrangles vertexes are set by indexes *id* of data points {*x*[i], *y*[i], *z*[i]}. String *sch* sets the color scheme. If string contain '`#`' then wire plot is produced. First dimensions of *id* must be 4 or greater. Arrays *x*, *y*, *z* must have equal sizes. Parameter *c* set the colors of quadrangles (if *id*.ny=*c*.nx) or colors of vertexes (if x.nx=*c*.nx). See also [triplot], page 50. See Section 5.7.12 [TriPlot and QuadPlot], page 136, for sample code and picture.

`dots` *xdat ydat zdat* ['*sch*'=''] [MGL command]
`dots` *xdat ydat zdat adat* ['*sch*'=''] [MGL command]
The function draws the arbitrary placed points {*x*[i], *y*[i], *z*[i]}. String *sch* sets the color scheme and kind of marks. If arrays *c*, *a* are specified then they define colors and transparencies of dots. You can use [tens], page 34, plot with style '` .`' to draw non-transparent dots with specified colors. Arrays *x*, *y*, *z*, *a* must have equal sizes. See also [crust], page 51, [tens], page 34, [mark], page 37, [plot], page 34. See Section 5.7.13 [Dots sample], page 137, for sample code and picture.

`crust` *xdat ydat zdat* ['*sch*'=''] [MGL command]
The function reconstruct and draws the surface for arbitrary placed points {*x*[i], *y*[i], *z*[i]}. String *sch* sets the color scheme. If string contain '`#`' then wire plot is produced. Arrays *x*, *y*, *z* must have equal sizes. See also [dots], page 51, [triplot], page 50.

3.17 Nonlinear fitting

These functions fit data to formula. Fitting goal is to find formula parameters for the best fit the data points, i.e. to minimize the sum $\sum_i (f(x_i, y_i, z_i) - a_i)^2 / s_i^2$. At this, approximation function '`f`' can depend only on one argument '`x`' (1D case), on two arguments '`x,y`' (2D case) and on three arguments '`x,y,z`' (3D case). The function '`f`' also may depend on parameters. Normally the list of fitted parameters is specified by *var* string (like, '`abcd`'). Usually user should supply initial values for fitted parameters by *ini* variable. But if he/she don't supply it then the zeros are used. Parameter *print*=`true` switch on printing the found coefficients to *Message* (see Section 3.2.9 [Error handling], page 20).

Functions Fit() and FitS() do not draw the obtained data themselves. They fill the data *fit* by formula '`f`' with found coefficients and return it. At this, the '`x,y,z`' coordinates are equidistantly distributed in the axis range. Number of points in *fit* is defined by option `value` (default is *mglFitPnts*=100). Note, that this functions use GSL library and do something only if MathGL was compiled with GSL support. See Section 5.9.13 [Nonlinear fitting hints], page 162, for sample code and picture.

`fits` *res adat sdat 'func' 'var'* [*ini=0*] [MGL command]
`fits` *res xdat adat sdat 'func' 'var'* [*ini=0*] [MGL command]
`fits` *res xdat ydat adat sdat 'func' 'var'* [*ini=0*] [MGL command]

`fits` *res xdat ydat zdat adat sdat 'func' 'var'* [*ini=0*] [MGL command]
Fit data along x-, y- and z-directions for array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) with weight factor *s*[i,j,k].

`fit` *res adat 'func' 'var'* [*ini=0*] [MGL command]
`fit` *res xdat adat 'func' 'var'* [*ini=0*] [MGL command]
`fit` *res xdat ydat adat 'func' 'var'* [*ini=0*] [MGL command]
`fit` *res xdat ydat zdat adat 'func' 'var'* [*ini=0*] [MGL command]
Fit data along x-, y- and z-directions for array specified parametrically *a*[i,j,k](*x*[i,j,k], *y*[i,j,k], *z*[i,j,k]) with weight factor 1.

`putsfit x y` ['*pre*'=" '*fnt*'=" `size=-1`] [MGL command]
Print last fitted formula with found coefficients (as numbers) at position *p0*. The string *prefix* will be printed before formula. All other parameters are the same as in Section 3.8 [Text printing], page 30.

3.18 Data manipulation

`hist` RES *xdat adat* [MGL command]
`hist` RES *xdat ydat adat* [MGL command]
`hist` RES *xdat ydat zdat adat* [MGL command]
These functions make distribution (histogram) of data. They do not draw the obtained data themselves. These functions can be useful if user have data defined for random points (for example, after PIC simulation) and he want to produce a plot which require regular data (defined on grid(s)). The range for grids is always selected as axis range. Arrays *x*, *y*, *z* define the positions (coordinates) of random points. Array *a* define the data value. Number of points in output array *res* is defined by option `value` (default is *mglFitPnts*=100).

`fill` *dat 'eq'* [MGL command]
`fill` *dat 'eq' vdat* [MGL command]
`fill` *dat 'eq' vdat wdat* [MGL command]
Fills the value of array '`u`' according to the formula in string *eq*. Formula is an arbitrary expression depending on variables '`x`', '`y`', '`z`', '`u`', '`v`', '`w`'. Coordinates '`x`', '`y`', '`z`' are supposed to be normalized in axis range. Variable '`u`' is the original value of the array. Variables '`v`' and '`w`' are values of arrays *v*, *w* which can be `NULL` (i.e. can be omitted).

`datagrid` *dat xdat ydat zdat* [MGL command]
Fills the value of array '`u`' according to the linear interpolation of triangulated surface, found for arbitrary placed points '`x`', '`y`', '`z`'. Interpolation is done at points equidistantly distributed in axis range. NAN value is used for grid points placed outside of triangulated surface. See Section 5.9.11 [Making regular data], page 160, for sample code and picture.

`refill` *dat xdat vdat* [*sl=-1*] [MGL command]
`refill` *dat xdat ydat vdat* [*sl=-1*] [MGL command]

`refill` *dat xdat ydat zdat vdat* [MGL command]

Fills by interpolated values of array *v* at the point {*x*, *y*, *z*}={`X[i], Y[j], Z[k]`} (or {*x*, *y*, *z*}={`X[i,j,k], Y[i,j,k], Z[i,j,k]`} if *x*, *y*, *z* are not 1d arrays), where `X,Y,Z` are equidistantly distributed in axis range and have the same sizes as array *dat*. If parameter *sl* is 0 or positive then changes will be applied only for slice *sl*.

`pde` RES *'ham' ini_re ini_im* [`dz=0.1 k0=100`] [MGL command]

Solves equation du/dz = i*k0**ham*(p,q,x,y,z,|u|)[u], where p=-i/k0*d/dx, q=-i/k0*d/dy are pseudo-differential operators. Parameters *ini_re*, *ini_im* specify real and imaginary part of initial field distribution. Coordinates '`x`', '`y`', '`z`' are supposed to be normalized in axis range. Note, that really this ranges are increased by factor 3/2 for purpose of reducing reflection from boundaries. Parameter *dz* set the step along evolutionary coordinate z. At this moment, simplified form of function *ham* is supported – all "mixed" terms (like '`x*p`'->x*d/dx) are excluded. For example, in 2D case this function is effectively $ham = f(p,z) + g(x,z,u)$. However commutable combinations (like '`x*q`'->x*d/dy) are allowed. Here variable '`u`' is used for field amplitude `|u|`. This allow one solve nonlinear problems – for example, for nonlinear Shrodinger equation you may set `ham="p^2 + q^2 - u^2"`. You may specify imaginary part for wave absorption, like `ham = "p^2 + i*x*(x>0)"`, but only if dependence on variable '`i`' is linear (i.e. $ham = hre + i * him$). See Section 5.9.14 [PDE solving hints], page 163, for sample code and picture.

4 Data processing

This chapter describe commands for allocation, resizing, loading and saving, modifying of data arrays. Also it can numerically differentiate and integrate data, interpolate, fill data by formula and so on. Class supports data with dimensions up to 3 (like function of 3 variables – x,y,z). Data arrays are denoted by Small Caps (like DAT) if it can be (re-)created by MGL commands.

4.1 Public variables

MGL don't support direct access to data arrays. See section Section 4.4 [Data filling], page 55,

4.2 Data constructor

There are many functions, which can create data for output (see Section 4.4 [Data filling], page 55, Section 4.5 [File I/O], page 57, Section 4.6 [Make another data], page 58, Section 4.11 [Global functions], page 65). Here I put most useful of them.

`new` DAT [`nx=1` *'eq'*] [MGL command]
`new` DAT `nx ny` [*'eq'*] [MGL command]
`new` DAT `nx ny nz` [*'eq'*] [MGL command]
Default constructor. Allocates the memory for data array and initializes it by zero. If string *eq* is specified then data will be filled by corresponding formula as in [fill], page 56.

`copy` DAT *dat2* [*'eq'*="] [MGL command]
`copy` DAT `val` [MGL command]
Copy constructor. Allocates the memory for data array and copy values from other array. At this, if parameter *eq* is specified then the data will be modified by corresponding formula similarly to [fill], page 56.

`read` DAT *'fname'* [MGL command]
Reads data from tab-separated text file with auto determining sizes of the data.

`delete` *dat* [MGL command]
Deletes the instance of class mglData.

4.3 Data resizing

`new` DAT [`nx=1 ny=1 nz=1`] [MGL command]
Creates or recreates the array with specified size and fills it by zero. This function does nothing if one of parameters *mx*, *my*, *mz* is zero or negative.

`rearrange` *dat* `mx` [`my=0 mz=0`] [MGL command]
Rearrange dimensions without changing data array so that resulting sizes should be *mx***my***mz* < nx*ny*nz. If some of parameter *my* or *mz* are zero then it will be selected to optimal fill of data array. For example, if *my*=0 then it will be change to *my*=nx*ny*nz/*mx* and *mz*=1.

`transpose` *dat* ['dim'='yxz'] [MGL command]
Transposes (shift order of) dimensions of the data. New order of dimensions is specified in string *dim*. This function can be useful also after reading of one-dimensional data.

`extend` *dat* `n1` [`n2=0`] [MGL command]
Increase the dimensions of the data by inserting new (|*n1*|+1)-th slices after (for *n1*>0) or before (for *n1*<0) of existed one. It is possible to insert 2 dimensions simultaneously for 1d data by using parameter *n2*. Data to new slices is copy from existed one. For example, for *n1*>0 new array will be $a_{ij}^{new} = a_i^{old}$ where j=0...*n1*. Correspondingly, for *n1*<0 new array will be $a_{ij}^{new} = a_j^{old}$ where i=0...|*n1*|.

`squeeze` *dat* `rx` [`ry=1 rz=1 sm=off`] [MGL command]
Reduces the data size by excluding data elements which indexes are not divisible by *rx*, *ry*, *rz* correspondingly. Parameter *smooth* set to use smoothing (i.e. $a_{out}[i] = \sum_{j=i,i+r} a[j]/r$) or not (i.e. $a_{out}[i] = a[j*r]$).

`crop` *dat* `n1 n2` *'dir'* [MGL command]
Cuts off edges of the data *i*<*n1* and *i*>*n2* if *n2*>0 or *i*>`n[xyz]`-*n2* if *n2*<=0 along direction *dir*.

`insert` *dat 'dir'* [`pos=off num=0`] [MGL command]
Insert *num* slices along *dir*-direction at position *pos* and fill it by zeros.

`delete` *dat 'dir'* [`pos=off num=0`] [MGL command]
Delete *num* slices along *dir*-direction at position *pos*.

`sort` *dat* `idx` [`idy=-1`] [MGL command]
Sort data rows (or slices in 3D case) by values of specified column *idx* (or cell {*idx*,*idy*} for 3D case). Note, this function is not thread safe!

`clean` *dat* `idx` [MGL command]
Delete rows which values are equal to next row for given column *idx*.

`join` *dat vdat* [*v2dat ...*] [MGL command]
Join data cells from *vdat* to *dat*. At this, function increase *dat* sizes according following: z-size for data arrays arrays with equal x-,y-sizes; or y-size for data arrays with equal x-sizes; or x-size otherwise.

4.4 Data filling

`list` DAT `v1 ...` [MGL command]
Creates new variable with name *dat* and fills it by numeric values of command arguments `v1 ...`. Command can create one-dimensional and two-dimensional arrays with arbitrary values. For creating 2d array the user should use delimiter '`|`' which means that the following values lie in next row. Array sizes are [maximal of row sizes * number of rows]. For example, command `list 1 | 2 3` creates the array [1 0; 2 3]. Note, that the maximal number of arguments is 1000.

`list` DAT *d1 ...* [MGL command]
Creates new variable with name *dat* and fills it by data values of arrays of command arguments *d1 ...*. Command can create two-dimensional or three-dimensional (if arrays in arguments are 2d arrays) arrays with arbitrary values. Minor dimensions of all arrays in arguments should be equal to dimensions of first array d1. In the opposite case the argument will be ignored. Note, that the maximal number of arguments is 1000.

`var` DAT `num v1 [v2=nan]` [MGL command]
Creates new variable with name *dat* for one-dimensional array of size *num*. Array elements are equidistantly distributed in range [*v1*, *v2*]. If *v2*=`nan` then *v2*=*v1* is used.

`fill` *dat v1 v2 ['dir'='x']* [MGL command]
Equidistantly fills the data values to range [*v1*, *v2*] in direction *dir*={'`x`','`y`','`z`'}.

`fill` *dat 'eq'* [MGL command]
`fill` *dat 'eq' vdat* [MGL command]
`fill` *dat 'eq' vdat wdat* [MGL command]
Fills the value of array according to the formula in string *eq*. Formula is an arbitrary expression depending on variables '`x`', '`y`', '`z`', '`u`', '`v`', '`w`'. Coordinates '`x`', '`y`', '`z`' are supposed to be normalized in axis range of canvas *gr* (in difference from `Modify` functions). Variable '`u`' is the original value of the array. Variables '`v`' and '`w`' are values of *vdat*, *wdat* which can be `NULL` (i.e. can be omitted).

`modify` *dat 'eq'* [`dim=0`] [MGL command]
`modify` *dat 'eq' vdat* [MGL command]
`modify` *dat 'eq' vdat wdat* [MGL command]
The same as previous ones but coordinates '`x`', '`y`', '`z`' are supposed to be normalized in range [0,1]. If *dim*>0 is specified then modification will be fulfilled only for slices >=*dim*.

`fillsample` *dat 'how'* [MGL command]
Fills data by 'x' or 'k' samples for Hankel ('h') or Fourier ('f') transform.

`datagrid` *dat xdat ydat zdat* [MGL command]
Fills the value of array according to the linear interpolation of triangulated surface assuming x-,y-coordinates equidistantly distributed in axis range (or in range [x1,x2]*[y1,y2]). Triangulated surface is found for arbitrary placed points '`x`', '`y`', '`z`'. NAN value is used for grid points placed outside of triangulated surface. See Section 5.9.11 [Making regular data], page 160, for sample code and picture.

`put` *dat* `val [i=: j=: k=:]` [MGL command]
Sets value(s) of array a[*i*, *j*, *k*] = *val*. Negative indexes *i*, *j*, *k*=-1 set the value *val* to whole range in corresponding direction(s). For example, `Put(val,-1,0,-1);` sets a[i,0,j]=*val* for i=0...(nx-1), j=0...(nz-1).

`put` *dat vdat* `[i=: j=: k=:]` [MGL command]
Copies value(s) from array *v* to the range of original array. Negative indexes *i*, *j*, *k*=-1 set the range in corresponding direction(s). At this minor dimensions of array *v* should

be large than corresponding dimensions of this array. For example, `Put(v,-1,0,-1);` sets a[i,0,j]=v.ny>nz ? v[i,j] : v[i], where i=0...(nx-1), j=0...(nz-1) and condition v.nx>=nx is true.

`refill` *dat xdat vdat* [*sl=-1*] [MGL command]
`refill` *dat xdat ydat vdat* [*sl=-1*] [MGL command]
`refill` *dat xdat ydat zdat vdat* [MGL command]
Fills by interpolated values of array *v* at the point {*x*, *y*, *z*}={`X[i], Y[j], Z[k]`} (or {*x*, *y*, *z*}={`X[i,j,k], Y[i,j,k], Z[i,j,k]`} if *x*, *y*, *z* are not 1d arrays), where `X,Y,Z` are equidistantly distributed in range [*x1*,*x2*]*[*y1*,*y2*]*[*z1*,*z2*] and have the same sizes as this array. If parameter *sl* is 0 or positive then changes will be applied only for slice *sl*.

`gspline` *dat xdat vdat* [*sl=-1*] [MGL command]
Fills by global cubic spline values of array *v* at the point *x*=`X[i]`, where `X` are equidistantly distributed in range [*x1*,*x2*] and have the same sizes as this array. If parameter *sl* is 0 or positive then changes will be applied only for slice *sl*.

`idset` *dat 'ids'* [MGL command]
Sets the symbol *ids* for data columns. The string should contain one symbol 'a'...'z' per column. These ids are used in [column], page 59.

4.5 File I/O

`read` DAT *'fname'* [MGL command]
Reads data from tab-separated text file with auto determining sizes of the data. Double newline means the beginning of new z-slice.

`read` DAT *'fname'* `mx` [`my=1 mz=1`] [MGL command]
Reads data from text file with specified data sizes. This function does nothing if one of parameters *mx*, *my* or *mz* is zero or negative.

`readmat` DAT *'fname'* [`dim=2`] [MGL command]
Read data from text file with size specified at beginning of the file by first *dim* numbers. At this, variable *dim* set data dimensions.

`readall` DAT *'templ'* `v1 v2` [`dv=1 slice=off`] [MGL command]
Join data arrays from several text files. The file names are determined by function call `sprintf(fname,templ,val);`, where *val* changes from *from* to *to* with step *step*. The data load one-by-one in the same slice if *as_slice*=`false` or as slice-by-slice if *as_slice*=`true`.

`readall` DAT *'templ'* [`slice=off`] [MGL command]
Join data arrays from several text files which filenames satisfied the template *templ* (for example, *templ*=`"t_*.dat"`). The data load one-by-one in the same slice if *as_slice*=`false` or as slice-by-slice if *as_slice*=`true`.

`scanfile` DAT *'fname' 'templ'* [MGL command]
Read file *fname* line-by-line and scan each line for numbers according the template *templ*. The numbers denoted as '`%g`' in the template. See Section 5.9.22 [Saving and scanning file], page 172, for sample code and picture.

`save` *dat 'fname'* [MGL command]
Saves the whole data array (for *ns*=`-1`) or only *ns*-th slice to the text file *fname*.

`save` *'str' 'fname'* [*'mode'='a'*] [MGL command]
Saves the string *str* to the text file *fname*. For parameter *mode*='`a`' will append string to the file (default); for *mode*='`w`' will overwrite the file. See Section 5.9.22 [Saving and scanning file], page 172, for sample code and picture.

`readhdf` DAT *'fname' 'dname'* [MGL command]
Reads data array named *dname* from HDF5 or HDF4 file. This function does nothing if HDF5|HDF4 was disabled during library compilation.

`savehdf` *dat 'fname' 'dname'* [`rewrite=off`] [MGL command]
Saves data array named *dname* to HDF5 file. This function does nothing if HDF5 was disabled during library compilation.

`datas` *'fname'* [MGL command]
Put data names from HDF5 file *fname* into *buf* as '\t' separated fields. In MGL version the list of data names will be printed as message. This function does nothing if HDF5 was disabled during library compilation.

`import` DAT *'fname' 'sch'* [`v1=0 v2=1`] [MGL command]
Reads data from bitmap file (now support only PNG format). The RGB values of bitmap pixels are transformed to mreal values in range [*v1*, *v2*] using color scheme *scheme* (see Section 2.4 [Color scheme], page 11).

`export` *dat 'fname' 'sch'* [`v1=0 v2=0`] [MGL command]
Saves data matrix (or `ns`-th slice for 3d data) to bitmap file (now support only PNG format). The data values are transformed from range [*v1*, *v2*] to RGB pixels of bitmap using color scheme *scheme* (see Section 2.4 [Color scheme], page 11). If *v1*>=*v2* then the values of *v1*, *v2* are automatically determined as minimal and maximal value of the data array.

4.6 Make another data

`subdata` RES *dat* `xx` [`yy=: zz=:`] [MGL command]
Extracts sub-array data from the original data array keeping fixed positive index. For example `SubData(-1,2)` extracts 3d row (indexes are zero based), `SubData(4,-1)` extracts 5th column, `SubData(-1,-1,3)` extracts 4th slice and so on. If argument(s) are non-integer then linear interpolation between slices is used. In MGL version this command usually is used as inline one `dat(xx,yy,zz)`. Function return NULL or create empty data if data cannot be created for given arguments.

`subdata` RES *dat xdat* [*ydat=: zdat=:*] [MGL command]
Extracts sub-array data from the original data array for indexes specified by arrays *xx*, *yy*, *zz* (indirect access). This function work like previous one for 1D arguments or numbers, and resulting array dimensions are equal dimensions of 1D arrays for corresponding direction. For 2D and 3D arrays in arguments, the resulting array have the same dimensions as input arrays. The dimensions of all argument must be

the same (or to be scalar 1*1*1) if they are 2D or 3D arrays. In MGL version this command usually is used as inline one `dat(xx,yy,zz)`. Function return NULL or create empty data if data cannot be created for given arguments. In C function some of *xx*, *yy*, *zz* can be NULL.

`column` RES *dat 'eq'* [MGL command]
Get column (or slice) of the data filled by formula *eq* on column ids. For example, `Column("n*w^2/exp(t)");`. The column ids must be defined first by [idset], page 57, function or read from files. In MGL version this command usually is used as inline one `dat('eq')`. Function return NULL or create empty data if data cannot be created for given arguments.

`resize` RES *dat* `mx` [`my=1 mz=1`] [MGL command]
Resizes the data to new size *mx*, *my*, *mz* from box (part) [*x1*,*x2*] x [*y1*,*y2*] x [*z1*,*z2*] of original array. Initially x,y,z coordinates are supposed to be in [0,1]. If one of sizes *mx*, *my* or *mz* is 0 then initial size is used. Function return NULL or create empty data if data cannot be created for given arguments.

`evaluate` RES *dat idat* [`norm=on`] [MGL command]
`evaluate` RES *dat idat jdat* [`norm=on`] [MGL command]
`evaluate` RES *dat idat jdat kdat* [`norm=on`] [MGL command]
Gets array which values is result of interpolation of original array for coordinates from other arrays. All dimensions must be the same for data *idat*, *jdat*, *kdat*. Coordinates from *idat*, *jdat*, *kdat* are supposed to be normalized in range [0,1] (if *norm*=`true`) or in ranges [0,nx], [0,ny], [0,nz] correspondingly. Function return NULL or create empty data if data cannot be created for given arguments.

`solve` RES *dat* `val` *'dir'* [`norm=on`] [MGL command]
`solve` RES *dat* `val` *'dir' idat* [`norm=on`] [MGL command]
Gets array which values is indexes (roots) along given direction *dir*, where interpolated values of data *dat* are equal to *val*. Output data will have the sizes of *dat* in directions transverse to *dir*. If data *idat* is provided then its values are used as starting points. This allows to find several branches by consequentive calls. Indexes are supposed to be normalized in range [0,1] (if *norm*=`true`) or in ranges [0,nx], [0,ny], [0,nz] correspondingly. Function return NULL or create empty data if data cannot be created for given arguments. See [Solve sample], page 88, for sample code and picture.

`roots` RES *'func' ini* [*'var'='x'*] [MGL command]
`roots` RES *'func'* `ini` [*'var'='x'*] [MGL command]
Find roots of equation 'func'=0 for variable *var* with initial guess *ini*. Secant method is used for root finding. Function return NULL or create empty data if data cannot be created for given arguments.

`hist` RES *dat* `num v1 v2` [`nsub=0`] [MGL command]
`hist` RES *dat wdat* `num v1 v2` [`nsub=0`] [MGL command]
Creates *n*-th points distribution of the data values in range [*v1*, *v2*]. Array *w* specifies weights of the data elements (by default is 1). Parameter *nsub* define the number of additional interpolated points (for smoothness of histogram). Function return NULL or create empty data if data cannot be created for given arguments. See also Section 3.18 [Data manipulation], page 52,

`momentum` RES *dat 'how' ['dir'='z']* [MGL command]
Gets momentum (1d-array) of the data along direction *dir*. String *how* contain kind of momentum. The momentum is defined like as $res_k = \sum_{ij} how(x_i, y_j, z_k) a_{ij} / \sum_{ij} a_{ij}$ if *dir*='`z`' and so on. Coordinates '`x`', '`y`', '`z`' are data indexes normalized in range [0,1]. Function return NULL or create empty data if data cannot be created for given arguments.

`sum` RES *dat 'dir'* [MGL command]
Gets array which is the result of summation in given direction or direction(s). Function return NULL or create empty data if data cannot be created for given arguments.

`max` RES *dat 'dir'* [MGL command]
Gets array which is the maximal data values in given direction or direction(s). Function return NULL or create empty data if data cannot be created for given arguments.

`min` RES *dat 'dir'* [MGL command]
Gets array which is the maximal data values in given direction or direction(s). Function return NULL or create empty data if data cannot be created for given arguments.

`combine` RES *adat bdat* [MGL command]
Returns direct multiplication of arrays (like, res[i,j] = this[i]*a[j] and so on). Function return NULL or create empty data if data cannot be created for given arguments.

`trace` RES *dat* [MGL command]
Gets array of diagonal elements a[i,i] (for 2D case) or a[i,i,i] (for 3D case) where i=0...nx-1. Function return copy of itself for 1D case. Data array must have dimensions ny,nz >= nx or ny,nz = 1. Function return NULL or create empty data if data cannot be created for given arguments.

`correl` RES *adat bdat 'dir'* [MGL command]
Find correlation between data *a* (or this in C++) and *b* along directions *dir*. Fourier transform is used to find the correlation. So, you may want to use functions [swap], page 61, or [norm], page 62, before plotting it. Function return NULL or create empty data if data cannot be created for given arguments.

`pulse` RES *dat 'dir'* [MGL command]
Find pulse properties along direction *dir*: pulse maximum (in column 0) and its position (in column 1), pulse width near maximum (in column 3) and by half height (in column 2), energy in first pulse (in column 4). NAN values are used for widths if maximum is located near the edges. Note, that there is uncertainty for complex data. Usually one should use square of absolute value (i.e. `|dat[i]|^2`) for them. So, MathGL don't provide this function for complex data arrays. However, C function will work even in this case but use absolute value (i.e. `|dat[i]|`). Function return NULL or create empty data if data cannot be created for given arguments. See also [max], page 60, [min], page 60, [momentum], page 60, [sum], page 60. See Section 5.9.16 [Pulse properties], page 167, for sample code and picture.

4.7 Data changing

These functions change the data in some direction like differentiations, integrations and so on. The direction in which the change will applied is specified by the string parameter, which may contain '`x`', '`y`' or '`z`' characters for 1-st, 2-nd and 3-d dimension correspondingly.

`cumsum` *dat 'dir'* [MGL command]
Cumulative summation of the data in given direction or directions.

`integrate` *dat 'dir'* [MGL command]
Integrates (like cumulative summation) the data in given direction or directions.

`diff` *dat 'dir'* [MGL command]
Differentiates the data in given direction or directions.

`diff` *dat xdat ydat* [*zdat=0*] [MGL command]
Differentiates the data specified parametrically in direction x with y, z=constant. Parametrical differentiation uses the formula (for 2D case): $da/dx = (a_j * y_i - a_i * y_j)/(x_j * y_i - x_i * y_j)$ where $a_i = da/di, a_j = da/dj$ denotes usual differentiation along 1st and 2nd dimensions. The similar formula is used for 3D case. Note, that you may change the order of arguments – for example, if you have 2D data a(i,j) which depend on coordinates {x(i,j), y(i,j)} then usual derivative along '`x`' will be `Diff(x,y);` and usual derivative along '`y`' will be `Diff(y,x);`.

`diff2` *dat 'dir'* [MGL command]
Double-differentiates (like Laplace operator) the data in given direction.

`sinfft` *dat 'dir'* [MGL command]
Do Sine transform of the data in given direction or directions. The Sine transform is $\sum a_j \sin(kj)$ (see `http://en.wikipedia.org/wiki/Discrete_sine_transform#DST-I`).

`cosfft` *dat 'dir'* [MGL command]
Do Cosine transform of the data in given direction or directions. The Cosine transform is $\sum a_j \cos(kj)$ (see `http://en.wikipedia.org/wiki/Discrete_cosine_transform#DCT-I`).

`hankel` *dat 'dir'* [MGL command]
Do Hankel transform of the data in given direction or directions. The Hankel transform is $\sum a_j J_0(kj)$ (see `http://en.wikipedia.org/wiki/Hankel_transform`).

`wavelet` *dat 'dir'* `k` [MGL command]
Apply wavelet transform of the data in given direction or directions. Parameter *dir* set the kind of wavelet transform: '`d`' for daubechies, '`D`' for centered daubechies, '`h`' for haar, '`H`' for centered haar, '`b`' for bspline, '`B`' for centered bspline. If string *dir* contain symbol '`i`' then inverse wavelet transform is applied. Parameter k set the size of wavelet transform.

`swap` *dat 'dir'* [MGL command]
Swaps the left and right part of the data in given direction (useful for Fourier spectrum).

`roll` *dat 'dir' num* [MGL command]
Rolls the data along direction *dir*. Resulting array will be out[i] = ini[(i+*num*)%nx] if `dir='x'`.

`mirror` *dat 'dir'* [MGL command]
Mirror the left-to-right part of the data in given direction. Looks like change the value index *i->n-i*. Note, that the similar effect in graphics you can reach by using options (see Section 2.7 [Command options], page 15), for example, `surf dat; xrange 1 -1`.

`sew` *dat* ['*dir*'='*xyz*' `da=2*pi`] [MGL command]
Remove value steps (like phase jumps after inverse trigonometric functions) with period *da* in given direction.

`smooth` *data* `type` ['*dir*'='*xyz*'] [MGL command]
Smooths the data on specified direction or directions. String *dirs* specifies the dimensions which will be smoothed. It may contain characters: '`x`' for 1st dimension, '`y`' for 2nd dimension, '`z`' for 3d dimension. If string *dir* contain: '`0`' then does nothing, '`3`' – linear averaging over 3 points, '`5`' – linear averaging over 5 points. If string *dir* contain '`dN`' (where '`N`' is digit 1,2,...,9) then linear averaging over (2*N+1)-th points is used. By default quadratic averaging over 5 points is used.

`envelop` *dat* ['*dir*'='*x*'] [MGL command]
Find envelop for data values along direction *dir*.

`diffract` *dat 'how'* `q` [MGL command]
Calculates one step of diffraction by finite-difference method with parameter $q=\delta t/\delta x^2$ using method with 3-d order of accuracy. Parameter *how* may contain:

- '`xyz`' for calculations along x-,y-,z-directions correspondingly;
- '`r`' for using axial symmetric Laplace operator for x-direction;
- '`0`' for zero boundary conditions;
- '`1`' for constant boundary conditions;
- '`2`' for linear boundary conditions;
- '`3`' for parabolic boundary conditions;
- '`4`' for exponential boundary conditions;
- '`5`' for gaussian boundary conditions.

`norm` *dat* `v1 v2` [`sym=off dim=0`] [MGL command]
Normalizes the data to range [*v1*,*v2*]. If flag *sym*=`true` then symmetrical interval [-max(|v1|,|v2|), max(|v1|,|v2|)] is used. Modification will be applied only for slices >=*dim*.

`normsl` *dat* `v1 v2` ['*dir*'='*z*' `keep=on sym=off`] [MGL command]
Normalizes data slice-by-slice along direction *dir* the data in slices to range [*v1*,*v2*]. If flag *sym*=`true` then symmetrical interval [-max(|v1|,|v2|), max(|v1|,|v2|)] is used. If *keep_en* is set then maximal value of k-th slice will be limited by $\sqrt{\sum a_{ij}(k)/\sum a_{ij}(0)}$.

`limit` *dat* `val` [MGL command]

Limits the data values to be inside the range [-*val*,*val*], keeping the original sign of the value (phase for complex numbers). This is equivalent to operation `a[i] *= abs(a[i])<val?1.:val/abs(a[i]);`.

4.8 Interpolation

MGL scripts can use spline interpolation by [evaluate], page 59, or [refill], page 57, commands. Also you can use [resize], page 59, for obtaining a data array with new sizes.

4.9 Data information

There are a set of functions for obtaining data properties in MGL language. However most of them can be found using `"suffixes"`. Suffix can get some numerical value of the data array (like its size, maximal or minimal value, the sum of elements and so on) as number. Later it can be used as usual number in command arguments. The suffixes start from point '`.`' right after (without spaces) variable name or its sub-array. For example, `a.nx` give the x-size of data *a*, `b(1).max` give maximal value of second row of variable *b*, `(c(:,0)^2).sum` give the sum of squares of elements in the first column of *c* and so on.

`info` *dat* [MGL command]

Gets or prints to file *fp* or as message (in MGL) information about the data (sizes, maximum/minimum, momentums and so on).

`info` *'txt'* [MGL command]

Prints string *txt* as message.

`info` *val* [MGL command]

Prints value of number *val* as message.

`print` *dat* [MGL command]
`print` *'txt'* [MGL command]
`print` *val* [MGL command]

The same as [info], page 63, but immediately print to stdout.

`echo` *dat* [MGL command]

Prints all values of the data array *dat* as message.

`(dat) .nx` [MGL suffix]
`(dat) .ny` [MGL suffix]
`(dat) .nz` [MGL suffix]

Gets the x-, y-, z-size of the data.

`(dat) .max` [MGL suffix]

Gets maximal value of the data.

`(dat) .min` [MGL suffix]

Gets minimal value of the data.

`(dat) .mx` [MGL suffix]
`(dat) .my` [MGL suffix]
`(dat) .mz` [MGL suffix]
Gets approximated (interpolated) position of maximum to variables x, y, z and returns the maximal value.

`(dat) .sum` [MGL suffix]
`(dat) .ax` [MGL suffix]
`(dat) .ay` [MGL suffix]
`(dat) .az` [MGL suffix]
`(dat) .aa` [MGL suffix]
`(dat) .wx` [MGL suffix]
`(dat) .wy` [MGL suffix]
`(dat) .wz` [MGL suffix]
`(dat) .wa` [MGL suffix]
`(dat) .sx` [MGL suffix]
`(dat) .sy` [MGL suffix]
`(dat) .sz` [MGL suffix]
`(dat) .sa` [MGL suffix]
`(dat) .kx` [MGL suffix]
`(dat) .ky` [MGL suffix]
`(dat) .kz` [MGL suffix]
`(dat) .ka` [MGL suffix]
Gets zero-momentum (energy, $I = \sum dat_i$) and write first momentum (median, $a = \sum \xi_i dat_i / I$), second momentum (width, $w^2 = \sum (\xi_i - a)^2 dat_i / I$), third momentum (skewness, $s = \sum (\xi_i - a)^3 dat_i / I w^3$) and fourth momentum (kurtosis, $k = \sum (\xi_i - a)^4 dat_i / 3 I w^4$) to variables. Here ξ is corresponding coordinate if *dir* is ‘`'x'`’, ‘`'y'`’ or ‘`'z'`’. Otherwise median is $a = \sum dat_i / N$, width is $w^2 = \sum (dat_i - a)^2 / N$ and so on.

`(dat) .fst` [MGL suffix]
Find position (after specified in i, j, k) of first nonzero value of formula *cond*. Function return the data value at found position.

`(dat) .lst` [MGL suffix]
Find position (before specified in i, j, k) of last nonzero value of formula *cond*. Function return the data value at found position.

`(dat) .a` [MGL suffix]
Give first (for `.a`, i.e. `dat->a[0]`).

4.10 Operators

`copy` DAT *dat2* [*'eq'="*] [MGL command]
Copies data from other variable.

`copy` *dat* `val` [MGL command]
Set all data values equal to *val*.

`multo` *dat dat2* [MGL command]
`multo` *dat* `val` [MGL command]
Multiplies data element by the other one or by value.

`divto` *dat dat2* [MGL command]
`divto` *dat* `val` [MGL command]
Divides each data element by the other one or by value.

`addto` *dat dat2* [MGL command]
`addto` *dat* `val` [MGL command]
Adds to each data element the other one or the value.

`subto` *dat dat2* [MGL command]
`subto` *dat* `val` [MGL command]
Subtracts from each data element the other one or the value.

4.11 Global functions

`transform` DAT *'type' real imag* [MGL command]
Does integral transformation of complex data *real*, *imag* on specified direction. The order of transformations is specified in string *type*: first character for x-dimension, second one for y-dimension, third one for z-dimension. The possible character are: '`f`' is forward Fourier transformation, '`i`' is inverse Fourier transformation, '`s`' is Sine transform, '`c`' is Cosine transform, '`h`' is Hankel transform, '`n`' or ' ' is no transformation.

`transforma` DAT *'type' ampl phase* [MGL command]
The same as previous but with specified amplitude *ampl* and phase *phase* of complex numbers.

`fourier` *reDat imDat 'dir'* [MGL command]
`fourier` *complexDat 'dir'* [MGL command]
Does Fourier transform of complex data *re*+i***im* in directions *dir*. Result is placed back into *re* and *im* data arrays.

`stfad` RES *real imag* `dn` ['*dir*'='x'] [MGL command]
Short time Fourier transformation for real and imaginary parts. Output is amplitude of partial Fourier of length *dn*. For example if *dir*='`x`', result will have size {int(nx/dn), dn, ny} and it will contain $res[i,j,k] = |\sum_d^d nexp(I*j*d)*(real[i*dn+d,k]+I*imag[i*dn+d,k])|/dn$.

`tridmat` RES ADAT BDAT CDAT DDAT *'how'* [MGL command]
Get array as solution of tridiagonal system of equations *A*[i]*x[i-1]+*B*[i]*x[i]+*C*[i]*x[i+1]=*D*[i]. String *how* may contain:

- '`xyz`' for solving along x-,y-,z-directions correspondingly;
- '`h`' for solving along hexagonal direction at x-y plain (require square matrix);
- '`c`' for using periodical boundary conditions;
- '`d`' for for diffraction/diffuse calculation (i.e. for using -*A*[i]**D*[i-1]+(2-*B*[i])**D*[i]-*C*[i]**D*[i+1] at right part instead of *D*[i]).

Data dimensions of arrays *A*, *B*, *C* should be equal. Also their dimensions need to be equal to all or to minor dimension(s) of array *D*. See Section 5.9.14 [PDE solving hints], page 163, for sample code and picture.

`pde` RES *'ham' ini_re ini_im* [`dz=0.1 k0=100`] [MGL command]
Solves equation du/dz = i*k0**ham*(p,q,x,y,z,|u|)[u], where p=-i/k0*d/dx, q=-i/k0*d/dy are pseudo-differential operators. Parameters *ini_re*, *ini_im* specify real and imaginary part of initial field distribution. Parameters *Min*, *Max* set the bounding box for the solution. Note, that really this ranges are increased by factor 3/2 for purpose of reducing reflection from boundaries. Parameter *dz* set the step along evolutionary coordinate z. At this moment, simplified form of function *ham* is supported – all "mixed" terms (like '`x*p`'->x*d/dx) are excluded. For example, in 2D case this function is effectively $ham = f(p,z) + g(x,z,u)$. However commutable combinations (like '`x*q`'->x*d/dy) are allowed. Here variable '`u`' is used for field amplitude `|u|`. This allow one solve nonlinear problems – for example, for nonlinear Shrodinger equation you may set `ham="p^2 + q^2 - u^2"`. You may specify imaginary part for wave absorption, like `ham = "p^2 + i*x*(x>0)"`, but only if dependence on variable '`i`' is linear (i.e. $ham = hre + i * him$). See also [qo2d], page 66, [qo3d], page 67. See Section 5.9.14 [PDE solving hints], page 163, for sample code and picture.

`ray` RES *'ham'* `x0 y0 z0 p0 q0 v0` [`dt=0.1 tmax=10`] [MGL command]
Solves GO ray equation like dr/dt = d *ham*/dp, dp/dt = -d *ham*/dr. This is Hamiltonian equations for particle trajectory in 3D case. Here *ham* is Hamiltonian which may depend on coordinates '`x`', '`y`', '`z`', momentums '`p`'=px, '`q`'=py, '`v`'=pz and time '`t`': $ham = H(x,y,z,p,q,v,t)$. The starting point (at `t=0`) is defined by variables *r0*, *p0*. Parameters *dt* and *tmax* specify the integration step and maximal time for ray tracing. Result is array of {x,y,z,p,q,v,t} with dimensions {7 * int(*tmax*/*dt*+1) }.

`ode` RES *'df' 'var' ini* [`dt=0.1 tmax=10`] [MGL command]
Solves ODE equations dx/dt = df(x). The functions *df* can be specified as string of ';'-separated textual formulas (argument *var* set the character ids of variables x[i]) or as callback function, which fill `dx` array for give `x`'s. Parameters *ini*, *dt*, *tmax* set initial values, time step and maximal time of the calculation. Result is data array with dimensions {*n* * int(*tmax*/*dt*+1)}.

`qo2d` RES *'ham' ini_re ini_im ray* [`r=1 k0=100` *xx yy*] [MGL command]
Solves equation du/dt = i*k0**ham*(p,q,x,y,|u|)[u], where p=-i/k0*d/dx, q=-i/k0*d/dy are pseudo-differential operators (see `mglPDE()` for details). Parameters *ini_re*, *ini_im* specify real and imaginary part of initial field distribution. Parameters *ray* set the reference ray, i.e. the ray around which the accompanied coordinate system will be maked. You may use, for example, the array created by [ray], page 66, function. Note, that the reference ray **must be** smooth enough to make accompanied coodrinates unambiguity. Otherwise errors in the solution may appear. If *xx* and *yy* are non-zero then Cartesian coordinates for each point will be written into them. See also [pde], page 66, [qo3d], page 67. See Section 5.9.14 [PDE solving hints], page 163, for sample code and picture.

`qo3d` RES *'ham' ini_re ini_im ray* [`r=1 k0=100` *xx yy zz*] [MGL command]
Solves equation du/dt = i*k0**ham*(p,q,v,x,y,z,|u|)[u], where p=-i/k0*d/dx, q=-i/k0*d/dy, v=-i/k0*d/dz are pseudo-differential operators (see `mglPDE()` for details). Parameters *ini_re*, *ini_im* specify real and imaginary part of initial field distribution. Parameters *ray* set the reference ray, i.e. the ray around which the accompanied coordinate system will be maked. You may use, for example, the array created by [ray], page 66, function. Note, that the reference ray **must be** smooth enough to make accompanied coodrinates unambiguity. Otherwise errors in the solution may appear. If *xx* and *yy* and *zz* are non-zero then Cartesian coordinates for each point will be written into them. See also [pde], page 66, [qo2d], page 66. See Section 5.9.14 [PDE solving hints], page 163, for sample code and picture.

`jacobian` RES *xdat ydat* [*zdat*] [MGL command]
Computes the Jacobian for transformation {i,j,k} to {*x*,*y*,*z*} where initial coordinates {i,j,k} are data indexes normalized in range [0,1]. The Jacobian is determined by formula det||$dr_\alpha/d\xi_\beta$|| where r={*x*,*y*,*z*} and ξ={i,j,k}. All dimensions must be the same for all data arrays. Data must be 3D if all 3 arrays {*x*,*y*,*z*} are specified or 2D if only 2 arrays {*x*,*y*} are specified.

`triangulation` RES *xdat ydat* [MGL command]
Computes triangulation for arbitrary placed points with coordinates {*x*,*y*} (i.e. finds triangles which connect points). MathGL use s-hull (`http://www.s-hull.org/`) code for triangulation. The sizes of 1st dimension **must be equal** for all arrays `x.nx=y.nx`. Resulting array can be used in [triplot], page 50, or [tricont], page 50, functions for visualization of reconstructed surface. See Section 5.9.11 [Making regular data], page 160, for sample code and picture.

`ifs2d` RES *dat* `num` [`skip=20`] [MGL command]
Computes *num* points {x[i]=res[0,i], y[i]=res[1,i]} for fractal using iterated function system. Matrix *dat* is used for generation according the formulas

```
x[i+1] = dat[0,i]*x[i] + dat[1,i]*y[i] + dat[4,i];
y[i+1] = dat[2,i]*x[i] + dat[3,i]*y[i] + dat[5,i];
```

Value `dat[6,i]` is used as weight factor for i-th row of matrix *dat*. At this first *skip* iterations will be omitted. Data array *dat* must have x-size greater or equal to 7. See Section 5.7.14 [IFS sample], page 138, for sample code and picture.

`ifs3d` RES *dat* `num` [`skip=20`] [MGL command]
Computes *num* points {x[i]=res[0,i], y[i]=res[1,i], z[i]=res[2,i]} for fractal using iterated function system. Matrix *dat* is used for generation according the formulas

```
x[i+1] = dat[0,i]*x[i] + dat[1,i]*y[i] + dat[2,i]*z[i] + dat[9,i];
y[i+1] = dat[3,i]*x[i] + dat[4,i]*y[i] + dat[5,i]*z[i] + dat[10,i];
z[i+1] = dat[6,i]*x[i] + dat[7,i]*y[i] + dat[8,i]*z[i] + dat[10,i];
```

Value `dat[12,i]` is used as weight factor for i-th row of matrix *dat*. At this first *skip* iterations will be omitted. Data array *dat* must have x-size greater or equal to 13. See Section 5.7.14 [IFS sample], page 138, for sample code and picture.

`ifsfile` RES *'fname' 'name'* `num` [`skip=20`] [MGL command]

Reads parameters of IFS fractal named *name* from file *fname* and computes *num* points for this fractal. At this first *skip* iterations will be omitted. See also [ifs2d], page 67, [ifs3d], page 67.

IFS file may contain several records. Each record contain the name of fractal ('`binary`' in the example below) and the body of fractal, which is enclosed in curly braces {}. Symbol '`;`' start the comment. If the name of fractal contain '`(3D)`' or '`(3d)`' then the 3d IFS fractal is specified. The sample below contain two fractals: '`binary`' – usual 2d fractal, and '`3dfern (3D)`' – 3d fractal.

```
binary
{ ; comment allowed here
 ; and here
 .5  .0 .0 .5 -2.563477 -0.000003 .333333   ; also comment allowed here
 .5  .0 .0 .5  2.436544 -0.000003 .333333
 .0 -.5 .5 .0  4.873085  7.563492 .333333
 }

3dfern (3D) {
  .00  .00 0 .0 .18 .0 0  0.0 0.00 0 0.0 0 .01
  .85  .00 0 .0 .85 .1 0 -0.1 0.85 0 1.6 0 .85
  .20 -.20 0 .2 .20 .0 0  0.0 0.30 0 0.8 0 .07
 -.20  .20 0 .2 .20 .0 0  0.0 0.30 0 0.8 0 .07
 }
```

4.12 Evaluate expression

You can use arbitrary formulas of existed data arrays or constants as any argument of data processing or data plotting commands. There are only 2 limitations: formula shouldn't contain spaces (to be recognized as single argument), and formula cannot be used as argument which will be (re)created by MGL command.

4.13 Special data classes

MGL use these special classes automatically.

5 MathGL examples

This chapter contain information about basic and advanced MathGL, hints and samples for all types of graphics. I recommend you read first 2 sections one after another and at least look on Section 5.9 [Hints], page 145, section. Also I recommend you to look at Chapter 2 [General concepts], page 8, and Section 5.10 [FAQ], page 172.

Most of sample scripts placed below use a set of functions for preparing the data.

```
func 'prepare1d'
new y 50 3
modify y '0.7*sin(2*pi*x)+0.5*cos(3*pi*x)+0.2*sin(pi*x)'
modify y 'sin(2*pi*x)' 1
modify y 'cos(2*pi*x)' 2
new x1 50 'x'
new x2 50 '0.05-0.03*cos(pi*x)'
new y1 50 '0.5-0.3*cos(pi*x)'
new y2 50 '-0.3*sin(pi*x)'
return

func 'prepare2d'
new a 50 40 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
new b 50 40 '0.6*cos(pi*(x+1))*cos(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
return

func 'prepare3d'
new c 61 50 40 '-2*(x^2+y^2+z^4-z^2)+0.2'
new d 61 50 40 '1-2*tanh((x+y)*(x+y))'
return

func 'prepare2v'
new a 20 30 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
new b 20 30 '0.6*cos(pi*(x+1))*cos(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
return

func 'prepare3v'
define $1 pow(x*x+y*y+(z-0.3)*(z-0.3)+0.03,1.5)
define $2 pow(x*x+y*y+(z+0.3)*(z+0.3)+0.03,1.5)
new ex 10 10 10 '0.2*x/$1-0.2*x/$2'
new ey 10 10 10 '0.2*y/$1-0.2*y/$2'
new ez 10 10 10 '0.2*(z-0.3)/$1-0.2*(z+0.3)/$2'
return
```

Basically, you can put this text after the script. Note, that you need to terminate main script by [stop], page 4, command before defining a function.

5.1 Basic usage

MGL script can be used by several manners. Each has positive and negative sides:

- *Using UDAV.*

  Positive sides are possibilities to view the plot at once and to modify it, rotate, zoom or switch on transparency or lighting by hands or by mouse. Negative side is the needness of the X-terminal.

- *Using command line tools.*

  Positive aspects are: batch processing of similar data set, for example, a set of resulting data files for different calculation parameters), running from the console program, including the cluster calculation), fast and automated drawing, saving pictures for further analysis, or demonstration). Negative sides are: the usage of the external program for picture viewing. Also, the data plotting is non-visual. So, you have to imagine the picture, view angles, lighting and so on) before the plotting. I recommend to use graphical window for determining the optimal parameters of plotting on the base of some typical data set. And later use these parameters for batch processing in console program.

  In this case you can use the program: `mglconv` or `mglview` for viewing.

- *Using C/C++/... code.*

  You can easily execute MGL script within C/C++/Fortan code. This can be useful for fast data plotting, for example, in web applications, where textual string (MGL script) may contain all necessary information for plot. The basic C++ code may look as following

  ```
  const char *mgl_script; // script itself, can be of type const wchar_t*
  mglGraph gr;
  mglParse pr;
  pr.Execute(&gr, mgl_script);
  ```

The simplest script is

```
box         # draw bounding box
axis        # draw axis
fplot 'x^3' # draw some function
```

Just type it in UDAV and press F5. Also you can save it in text file '`test.mgl`' and type in the console `mglconv test.mgl` what produce file '`test.mgl.png`' with resulting picture.

5.2 Advanced usage

Now I show several non-obvious features of MGL: several subplots in a single picture, curvilinear coordinates, text printing and so on. Generally you may miss this section at first reading, but I don't recommend it.

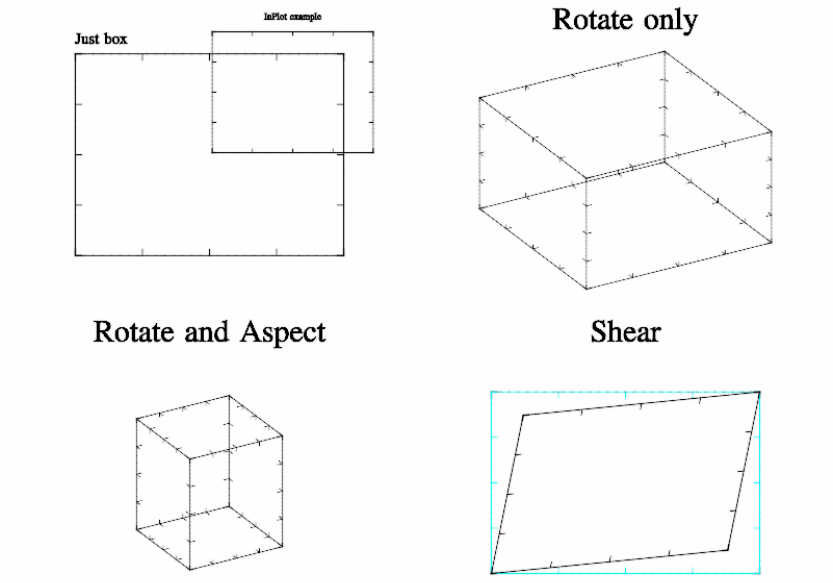
5.2.1 Subplots

Let me demonstrate possibilities of plot positioning and rotation. MathGL has a set of functions: [subplot], page 24, [inplot], page 25, [title], page 25, [aspect], page 26, and [rotate], page 25, and so on (see Section 3.4 [Subplots and rotation], page 24). The order of their calling is strictly determined. First, one changes the position of plot in image area (functions [subplot], page 24, [inplot], page 25, and [multiplot], page 25). Secondly, you can add the title of plot by [title], page 25, function. After that one may rotate the plot (command [rotate], page 25). Finally, one may change aspects of axes (command [aspect], page 26). The following code illustrates the aforesaid it:

```
subplot 2 2 0
box:text -1 1.1 'Just box' ':L'
inplot 0.2 0.5 0.7 1 off
box:text 0 1.2 'InPlot example'

subplot 2 2 1:title 'Rotate only'
rotate 50 60:box

subplot 2 2 2:title 'Rotate and Aspect'
rotate 50 60:aspect 1 1 2:box

subplot 2 2 3:title 'Shear'
box 'c':shear 0.2 0.1:box
```

Here I used function `Puts` for printing the text in arbitrary position of picture (see Section 3.8 [Text printing], page 30). Text coordinates and size are connected with axes. However, text coordinates may be everywhere, including the outside the bounding box. I'll show its features later in Section 5.2.7 [Text features], page 81.

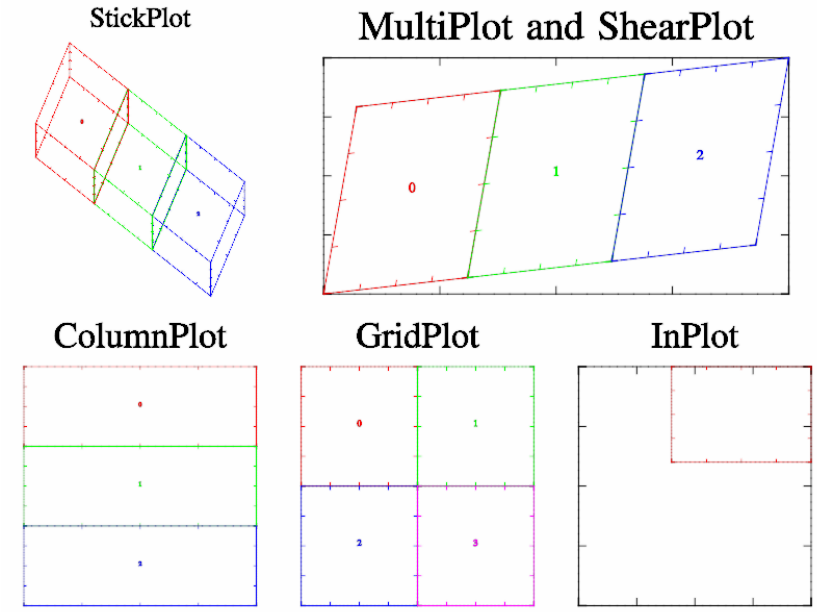
Note that several commands can be placed in a string if they are separated by ':' symbol.



More complicated sample show how to use most of positioning functions:

```
subplot 3 2 0:title 'StickPlot'
stickplot 3 0 20 30:box 'r':text 0 0 0 '0' 'r'
stickplot 3 1 20 30:box 'g':text 0 0 0 '1' 'g'
stickplot 3 2 20 30:box 'b':text 0 0 0 '2' 'b'

subplot 3 2 3 '':title 'ColumnPlot'
columnplot 3 0:box 'r':text 0 0 '0' 'r'
columnplot 3 1:box 'g':text 0 0 '1' 'g'
```

```
columnplot 3 2:box 'b':text 0 0 '2' 'b'

subplot 3 2 4 '':title 'GridPlot'
gridplot 2 2 0:box 'r':text 0 0 '0' 'r'
gridplot 2 2 1:box 'g':text 0 0 '1' 'g'
gridplot 2 2 2:box 'b':text 0 0 '2' 'b'
gridplot 2 2 3:box 'm':text 0 0 '3' 'm'

subplot 3 2 5 '':title 'InPlot':box
inplot 0.4 1 0.6 1 on:box 'r'

multiplot 3 2 1 2 1 '':title 'MultiPlot and ShearPlot':box
shearplot 3 0 0.2 0.1:box 'r':text 0 0 '0' 'r'
shearplot 3 1 0.2 0.1:box 'g':text 0 0 '1' 'g'
shearplot 3 2 0.2 0.1:box 'b':text 0 0 '2' 'b'
```



5.2.2 Axis and ticks

MathGL library can draw not only the bounding box but also the axes, grids, labels and so on. The ranges of axes and their origin (the point of intersection) are determined by functions `SetRange()`, `SetRanges()`, `SetOrigin()` (see Section 3.3.1 [Ranges (bounding box)], page 21). Ticks on axis are specified by function `SetTicks`, `SetTicksVal`, `SetTicksTime` (see Section 3.3.3 [Ticks], page 23). But usually

Command [axis], page 31, draws axes. Its textual string shows in which directions the axis or axes will be drawn (by default `"xyz"`, function draws axes in all directions). Command [grid], page 32, draws grid perpendicularly to specified directions. Example of axes and grid drawing is:

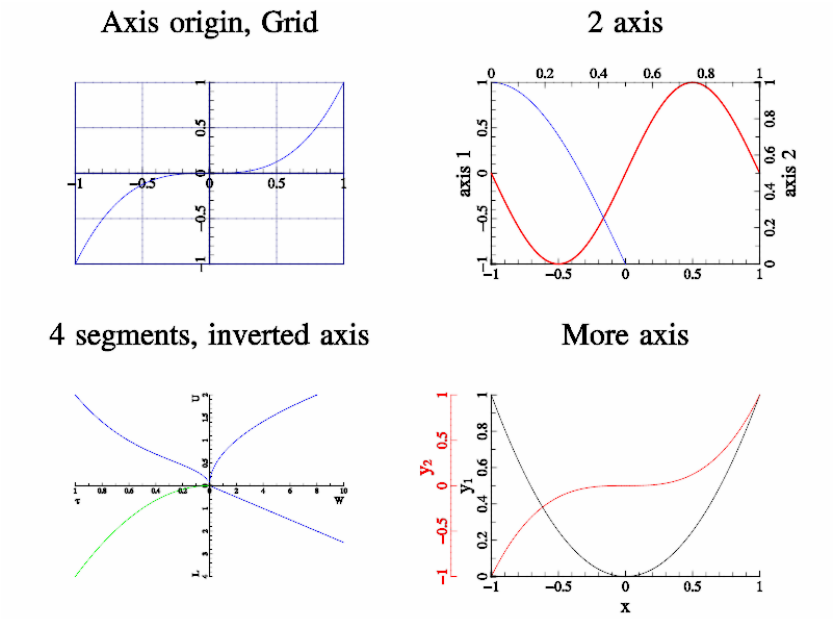
```
subplot 2 2 0:title 'Axis origin, Grid'
```

```
origin 0 0:axis:grid:fplot 'x^3'

subplot 2 2 1:title '2 axis'
ranges -1 1 -1 1:origin -1 -1:axis
ylabel 'axis_1':fplot 'sin(pi*x)' 'r2'
ranges 0 1 0 1:origin 1 1:axis
ylabel 'axis_2':fplot 'cos(pi*x)'

subplot 2 2 3:title 'More axis'
origin nan nan:xrange -1 1:axis
xlabel 'x' 0:ylabel 'y_1' 0:fplot 'x^2' 'k'
yrange -1 1:origin -1.3 -1:axis 'y' 'r'
ylabel '#r{y_2}' 0.2:fplot 'x^3' 'r'

subplot 2 2 2:title '4 segments, inverted axis':origin 0 0:
inplot 0.5 1 0.5 1 on:ranges 0 10 0 2:axis
fplot 'sqrt(x/2)':xlabel 'W' 1:ylabel 'U' 1
inplot 0 0.5 0.5 1 on:ranges 1 0 0 2:axis 'x'
fplot 'sqrt(x)+x^3':xlabel '\tau' 1
inplot 0.5 1 0 0.5 on:ranges 0 10 4 0:axis 'y'
fplot 'x/4':ylabel 'L' -1
inplot 0 0.5 0 0.5 on:ranges 1 0 4 0:fplot '4*x^2'
```

Note, that MathGL can draw not only single axis (which is default). But also several axis on the plot (see right plots). The idea is that the change of settings does not influence on the already drawn graphics. So, for 2-axes I setup the first axis and draw everything concerning it. Then I setup the second axis and draw things for the second axis. Generally, the similar idea allows one to draw rather complicated plot of 4 axis with different ranges (see bottom left plot).

At this inverted axis can be created by 2 methods. First one is used in this sample – just specify minimal axis value to be large than maximal one. This method work well for 2D axis, but can wrongly place labels in 3D case. Second method is more general and work in 3D case too – just use [aspect], page 26, function with negative arguments. For example, following code will produce exactly the same result for 2D case, but 2nd variant will look better in 3D.

```
# variant 1
ranges 0 10 4 0:axis

# variant 2
ranges 0 10 0 4:aspect 1 -1:axis
```



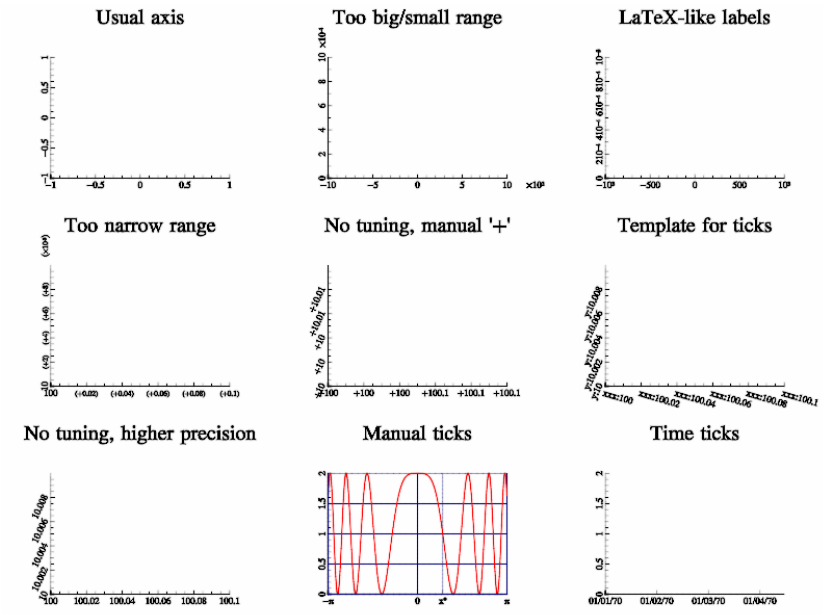
Another MathGL feature is fine ticks tunning. By default (if it is not changed by `SetTicks` function), MathGL try to adjust ticks positioning, so that they looks most human readable. At this, MathGL try to extract common factor for too large or too small axis ranges, as well as for too narrow ranges. Last one is non-common notation and can be disabled by `SetTuneTicks` function.

Also, one can specify its own ticks with arbitrary labels by help of `SetTicksVal` function. Or one can set ticks in time format. In last case MathGL will try to select optimal format for labels with automatic switching between years, months/days, hours/minutes/seconds or microseconds. However, you can specify its own time representation using formats described in `http://www.manpagez.com/man/3/strftime/`. Most common variants are '`%X`' for national representation of time, '`%x`' for national representation of date, '`%Y`' for year with century.

The sample code, demonstrated ticks feature is

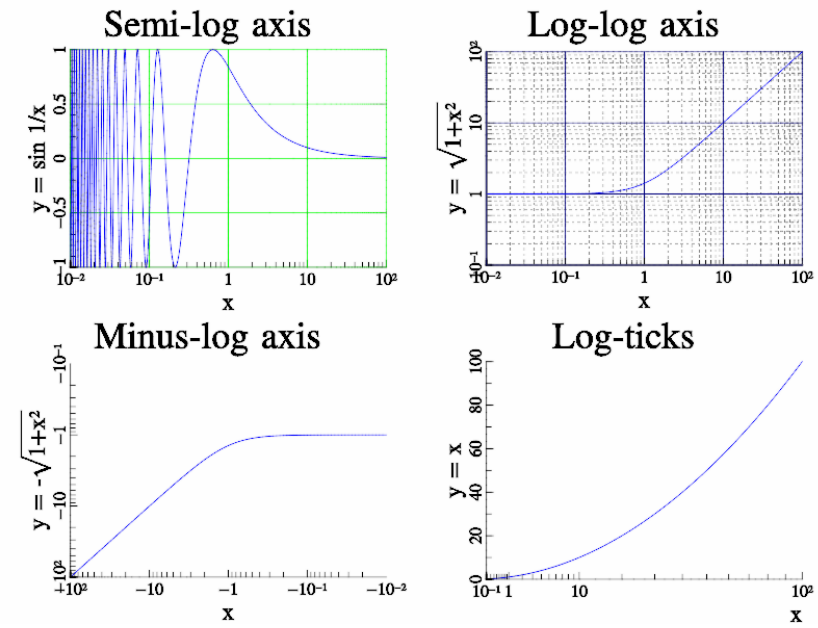
```
subplot 3 3 0:title 'Usual axis'
axis

subplot 3 3 1:title 'Too big/small range'
ranges -1000 1000 0 0.001:axis

subplot 3 3 2:title 'LaTeX-like labels'
axis 'F!'

subplot 3 3 3:title 'Too narrow range'
ranges 100 100.1 10 10.01:axis

subplot 3 3 4:title 'No tuning, manual "+"'
axis '+!'
# for version <2.3 you can use
#tuneticks off:axis
```

```
subplot 3 3 5:title 'Template for ticks'
xtick 'xxx:%g':ytick 'y:%g'
axis

xtick '':ytick '' # switch it off for other plots

subplot 3 3 6:title 'No tuning, higher precision'
axis '!4'

subplot 3 3 7:title 'Manual ticks'
ranges -pi pi 0 2
xtick pi 3 '\pi'
xtick 0.886 'x^*' on # note this will disable subticks drawing
# or you can use
#xtick -pi '\pi' -pi/2 '-\pi/2' 0 '0' 0.886 'x^*' pi/2 '\pi/2' pi 'pi'
# or you can use
#list v -pi -pi/2 0 0.886 pi/2 pi:xtick v '-\pi\n-\pi/2\n{}0\n{}x^*\n\pi/2\n\pi'
axis:grid:fplot '2*cos(x^2)^2' 'r2'

subplot 3 3 8:title 'Time ticks'
xrange 0 3e5:ticktime 'x':axis
```



The last sample I want to show in this subsection is Log-axis. From MathGL's point of view, the log-axis is particular case of general curvilinear coordinates. So, we need first define new coordinates (see also Section 5.2.3 [Curvilinear coordinates], page 77) by help of `SetFunc` or `SetCoor` functions. At this one should wary about proper axis range. So the code looks as following:

```
subplot 2 2 0 '<_':title 'Semi-log axis'
ranges 0.01 100 -1 1:axis 'lg(x)' '' ''
axis:grid 'xy' 'g':fplot 'sin(1/x)'
xlabel 'x' 0:ylabel 'y = sin 1/x' 0

subplot 2 2 1 '<_':title 'Log-log axis'
ranges 0.01 100 0.1 100:axis 'lg(x)' 'lg(y)' ''
axis:grid '!' 'h=':grid:fplot 'sqrt(1+x^2)'
xlabel 'x' 0:ylabel 'y = \sqrt{1+x^2}' 0

subplot 2 2 2 '<_':title 'Minus-log axis'
ranges -100 -0.01 -100 -0.1:axis '-lg(-x)' '-lg(-y)' ''
axis:fplot '-sqrt(1+x^2)'
xlabel 'x' 0:ylabel 'y = -\sqrt{1+x^2}' 0

subplot 2 2 3 '<_':title 'Log-ticks'
ranges 0.01 100 0 100:axis 'sqrt(x)' '' ''
axis:fplot 'x'
xlabel 'x' 1:ylabel 'y = x' 0
```

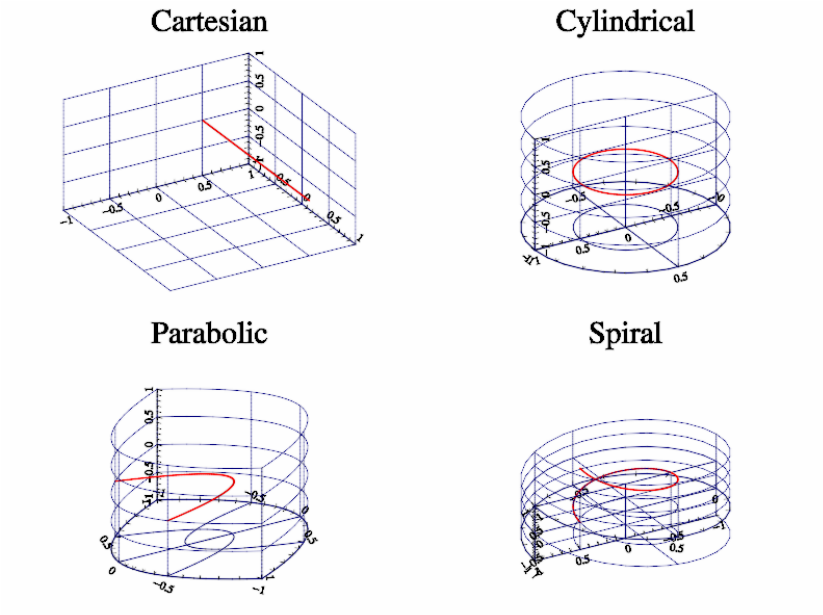


You can see that MathGL automatically switch to log-ticks as we define log-axis formula (in difference from v.1.*). Moreover, it switch to log-ticks for any formula if axis range will be large enough (see right bottom plot). Another interesting feature is that you not necessary define usual log-axis (i.e. when coordinates are positive), but you can define "minus-log" axis when coordinate is negative (see left bottom plot).

5.2.3 Curvilinear coordinates

As I noted in previous subsection, MathGL support curvilinear coordinates. In difference from other plotting programs and libraries, MathGL uses textual formulas for connection of the old (data) and new (output) coordinates. This allows one to plot in arbitrary coordinates. The following code plots the line y=0, z=0 in Cartesian, polar, parabolic and spiral coordinates:

```
origin -1 1 -1
subplot 2 2 0:title 'Cartesian':rotate 50 60
fplot '2*t-1' '0.5' '0' '2r':axis:grid

axis 'y*sin(pi*x)' 'y*cos(pi*x)' '':
subplot 2 2 1:title 'Cylindrical':rotate 50 60
fplot '2*t-1' '0.5' '0' '2r':axis:grid

axis '2*y*x' 'y*y - x*x' ''
subplot 2 2 2:title 'Parabolic':rotate 50 60
fplot '2*t-1' '0.5' '0' '2r':axis:grid

axis 'y*sin(pi*x)' 'y*cos(pi*x)' 'x+z'
subplot 2 2 3:title 'Spiral':rotate 50 60
fplot '2*t-1' '0.5' '0' '2r':axis:grid
```

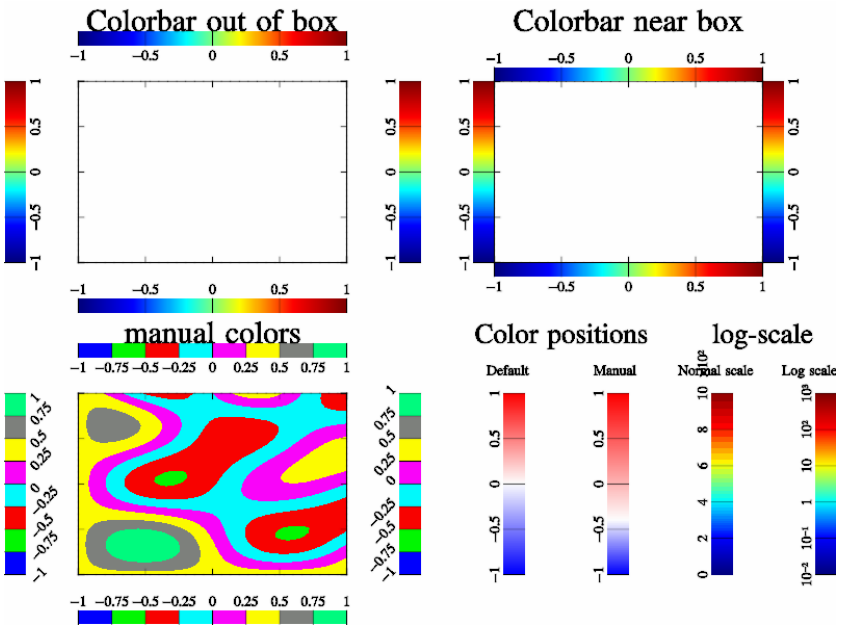


5.2.4 Colorbars

MathGL handle [colorbar], page 32, as special kind of axis. So, most of functions for axis and ticks setup will work for colorbar too. Colorbars can be in log-scale, and generally as arbitrary function scale; common factor of colorbar labels can be separated; and so on.

But of course, there are differences – colorbars usually located out of bounding box. At this, colorbars can be at subplot boundaries (by default), or at bounding box (if symbol 'I' is specified). Colorbars can handle sharp colors. And they can be located at arbitrary position too. The sample code, which demonstrate colorbar features is:

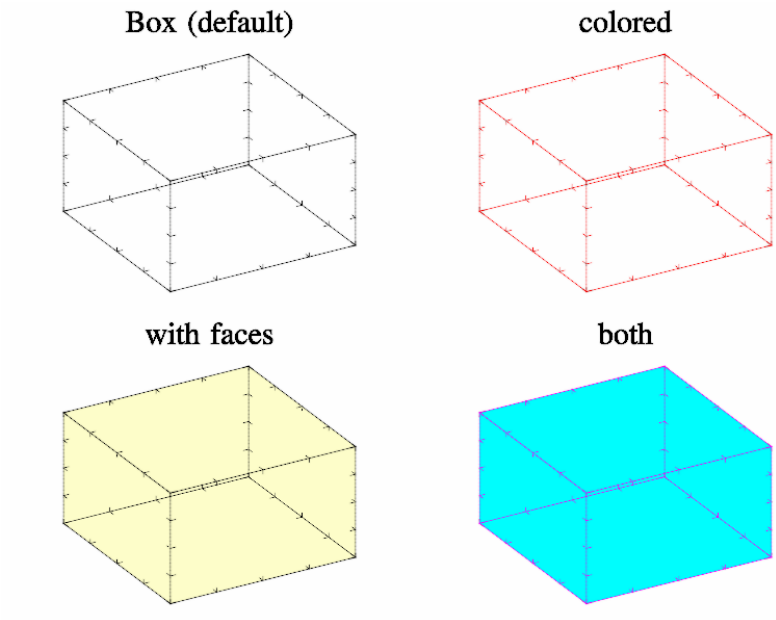
```
call 'prepare2d'
new v 9 'x'

subplot 2 2 0:title 'Colorbar out of box':box
colorbar '<':colorbar '>':colorbar '_':colorbar '^'

subplot 2 2 1:title 'Colorbar near box':box
colorbar '<I':colorbar '>I':colorbar '_I':colorbar '^I'

subplot 2 2 2:title 'manual colors':box:contd v a
colorbar v '<':colorbar v '>':colorbar v '_':colorbar v '^'

subplot 2 2 3:title '':text -0.5 1.55 'Color positions' ':C' -2

colorbar 'bwr>' 0.25 0:text -0.9 1.2 'Default'
colorbar 'b{w,0.3}r>' 0.5 0:text -0.1 1.2 'Manual'

crange 0.01 1e3
colorbar '>' 0.75 0:text 0.65 1.2 'Normal scale'
colorbar '>':text 1.35 1.2 'Log scale'
```



5.2.5 Bounding box

Box around the plot is rather useful thing because it allows one to: see the plot boundaries, and better estimate points position since box contain another set of ticks. MathGL provide special function for drawing such box – [box], page 33, function. By default, it draw black or white box with ticks (color depend on transparency type, see Section 5.9.3 [Types of transparency], page 147). However, you can change the color of box, or add drawing of rectangles at rear faces of box. Also you can disable ticks drawing, but I don't know why anybody will want it. The sample code, which demonstrate [box], page 33, features is:

```
subplot 2 2 0:title 'Box (default)':rotate 50 60:box

subplot 2 2 1:title 'colored':rotate 50 60:box 'r'

subplot 2 2 2:title 'with faces':rotate 50 60:box '@'

subplot 2 2 3:title 'both':rotate 50 60:box '@cm'
```



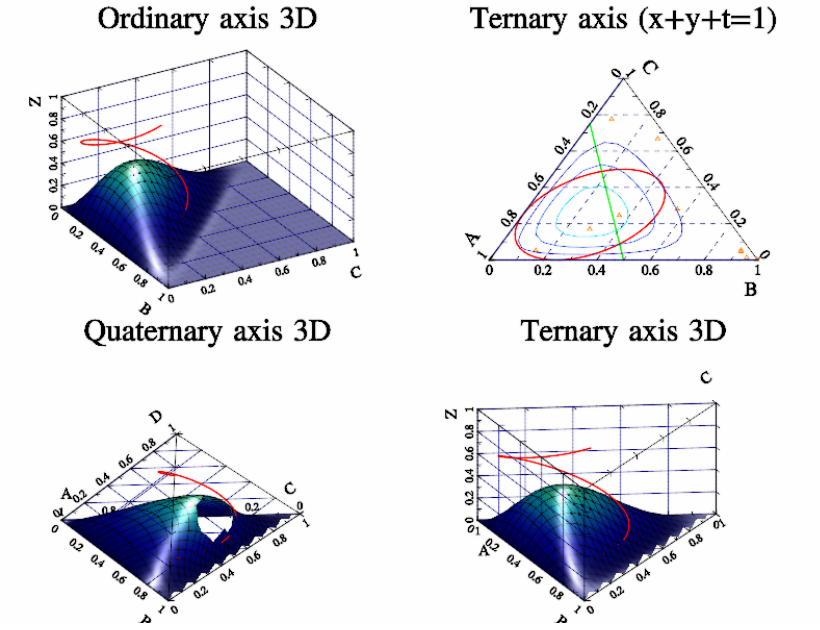
5.2.6 Ternary axis

There are another unusual axis types which are supported by MathGL. These are ternary and quaternary axis. Ternary axis is special axis of 3 coordinates a, b, c which satisfy relation $a+b+c=1$. Correspondingly, quaternary axis is special axis of 4 coordinates a, b, c, d which satisfy relation $a+b+c+d=1$.

Generally speaking, only 2 of coordinates (3 for quaternary) are independent. So, MathGL just introduce some special transformation formulas which treat a as '`x`', b as '`y`' (and c as '`z`' for quaternary). As result, all plotting functions (curves, surfaces, contours and so on) work as usual, but in new axis. You should use [ternary], page 22, function for switching to ternary/quaternary coordinates. The sample code is:

```
ranges 0 1 0 1 0 1
```

```
new x 50 '0.25*(1+cos(2*pi*x))'
new y 50 '0.25*(1+sin(2*pi*x))'
new z 50 'x'
new a 20 30 '30*x*y*(1-x-y)^2*(x+y<1)'
new rx 10 'rnd':copy ry (1-rx)*rnd
light on

subplot 2 2 0:title 'Ordinary axis 3D':rotate 50 60
box:axis:grid
plot x y z 'r2':surf a '#'
xlabel 'B':ylabel 'C':zlabel 'Z'

subplot 2 2 1:title 'Ternary axis (x+y+t=1)':ternary 1
box:axis:grid 'xyz' 'B;'
plot x y 'r2':plot rx ry 'q^ ':cont a:line 0.5 0 0 0.75 'g2'
xlabel 'B':ylabel 'C':tlabel 'A'

subplot 2 2 2:title 'Quaternary axis 3D':rotate 50 60:ternary 2
box:axis:grid 'xyz' 'B;'
plot x y z 'r2':surf a '#'
xlabel 'B':ylabel 'C':tlabel 'A':zlabel 'D'

subplot 2 2 3:title 'Ternary axis 3D':rotate 50 60:ternary 1
box:axis:grid 'xyz' 'B;'
plot x y z 'r2':surf a '#'
xlabel 'B':ylabel 'C':tlabel 'A':zlabel 'Z'
```



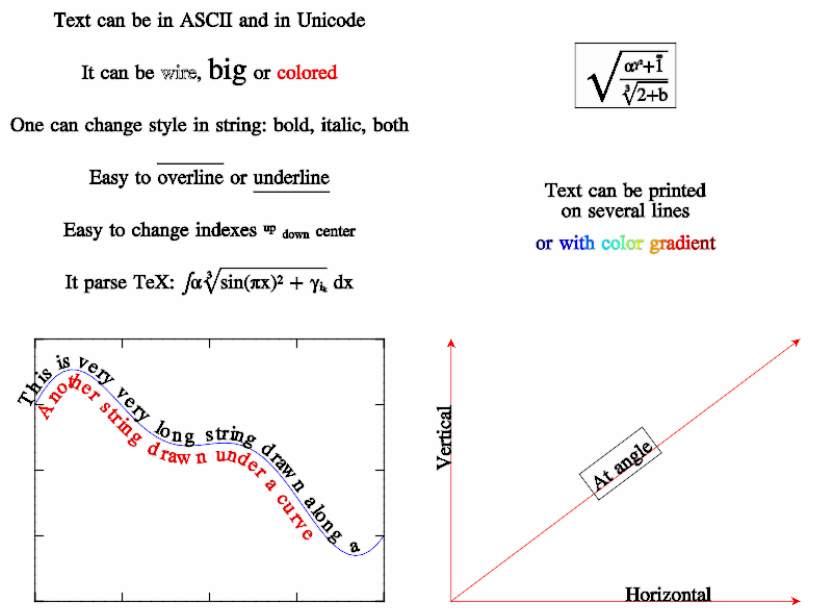
5.2.7 Text features

MathGL prints text by vector font. There are functions for manual specifying of text position (like `Puts`) and for its automatic selection (like `Label`, `Legend` and so on). MathGL prints text always in specified position even if it lies outside the bounding box. The default size of font is specified by functions *SetFontSize** (see Section 3.2.6 [Font settings], page 20). However, the actual size of output string depends on subplot size (depends on functions `SubPlot`, `InPlot`). The switching of the font style (italic, bold, wire and so on) can be done for the whole string (by function parameter) or inside the string. By default MathGL parses TeX-like commands for symbols and indexes (see Section 2.5 [Font styles], page 13).

Text can be printed as usual one (from left to right), along some direction (rotated text), or along a curve. Text can be printed on several lines, divided by new line symbol '`\n`'.

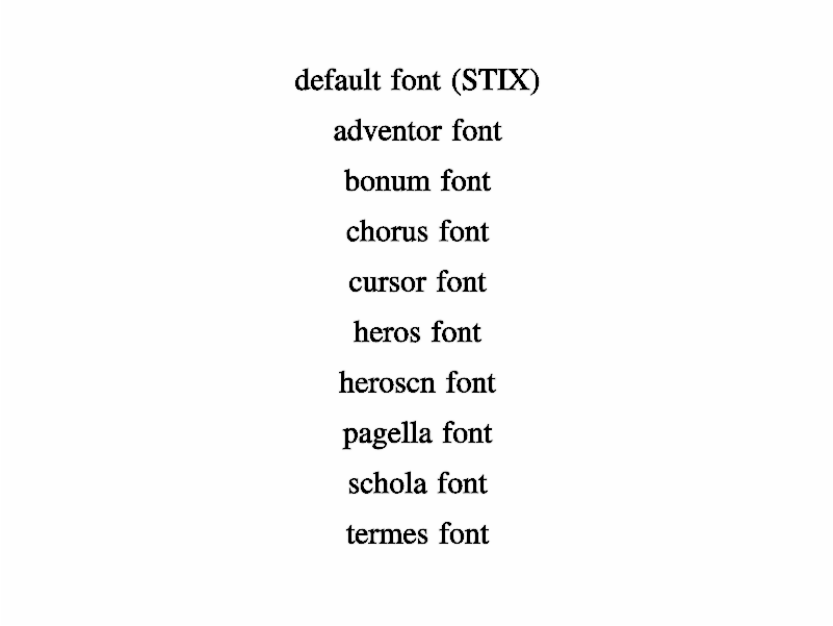
Example of MathGL font drawing is:

```
call 'prepare1d'

subplot 2 2 0 ''
text 0 1 'Text can be in ASCII and in Unicode'
text 0 0.6 'It can be \wire{wire}, \big{big} or #r{colored}'
text 0 0.2 'One can change style in string: \b{bold}, \i{italic, \b{both}}'
text 0 -0.2 'Easy to \a{overline} or \u{underline}'
text 0 -0.6 'Easy to change indexes ^{up} _{down} @{center}'
text 0 -1 'It parse TeX: \int \alpha \cdot \
\sqrt3{sin(\pi x)^2 + \gamma_{i_k}} dx'

subplot 2 2 1 ''
 text 0 0.5 '\sqrt{\frac{\alpha^{\gamma^2}+\overset 1{\big\infty}}{\sqrt3{2+b}}}' '@' -2
text 0 -0.5 'Text can be printed\n{}on several lines'

subplot 2 2 2 '':box:plot y(:,0)
text y 'This is very very long string drawn along a curve' 'k'
text y 'Another string drawn under a curve' 'Tr'

subplot 2 2 3 '':line -1 -1 1 -1 'rA':text 0 -1 1 -1 'Horizontal'
line -1 -1 1 1 'rA':text 0 0 1 1 'At angle' '@'
line -1 -1 -1 1 'rA':text -1 0 -1 1 'Vertical'
```



You can change font faces by loading font files by function [loadfont], page 20. Note, that this is long-run procedure. Font faces can be downloaded from MathGL website (`http://mathgl.sourceforge.net/download.html`) or from here (`http://sourceforge.net/project/showfiles.php?group_id=152187&package_id=267177`). The sample code is:

```
define d 0.25
loadfont 'STIX':text 0 1.1 'default font (STIX)'
loadfont 'adventor':text 0 1.1-d 'adventor font'
loadfont 'bonum':text 0 1.1-2*d 'bonum font'
loadfont 'chorus':text 0 1.1-3*d 'chorus font'
loadfont 'cursor':text 0 1.1-4*d 'cursor font'
loadfont 'heros':text 0 1.1-5*d 'heros font'
loadfont 'heroscn':text 0 1.1-6*d 'heroscn font'
loadfont 'pagella':text 0 1.1-7*d 'pagella font'
loadfont 'schola':text 0 1.1-8*d 'schola font'
loadfont 'termes':text 0 1.1-9*d 'termes font'
```



5.2.8 Legend sample

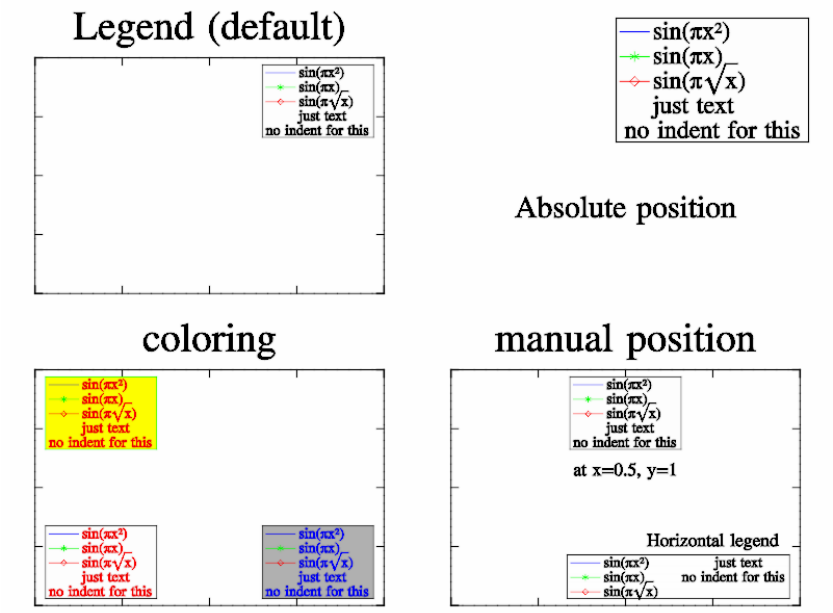
Legend is one of standard ways to show plot annotations. Basically you need to connect the plot style (line style, marker and color) with some text. In MathGL, you can do it by 2 methods: manually using [addlegend], page 34, function; or use '`legend`' option (see Section 2.7 [Command options], page 15), which will use last plot style. In both cases, legend entries will be added into internal accumulator, which later used for legend drawing itself. [clearlegend], page 34, function allow you to remove all saved legend entries.

There are 2 features. If plot style is empty then text will be printed without indent. If you want to plot the text with indent but without plot sample then you need to use space ' ' as plot style. Such style ' ' will draw a plot sample (line with marker(s)) which is invisible line (i.e. nothing) and print the text with indent as usual one.

Command [legend], page 33, draw legend on the plot. The position of the legend can be selected automatic or manually. You can change the size and style of text labels, as well as setup the plot sample. The sample code demonstrating legend features is:

```
addlegend 'sin(\pi {x^2})' 'b'
addlegend 'sin(\pi x)' 'g*'
addlegend 'sin(\pi \sqrt{x})' 'rd'
addlegend 'jsut text' ' '
addlegend 'no indent for this' ''

subplot 2 2 0 '':title 'Legend (default)':box
legend

text 0.75 0.65 'Absolute position' 'A'
legend 3 'A#'

subplot 2 2 2 '':title 'coloring':box
legend 0 'r#':legend 1 'Wb#':legend 2 'ygr#'
```

```
subplot 2 2 3 '':title 'manual position':box
legend 0.5 1:text 0.5 0.55 'at x=0.5, y=1' 'a'
legend 1 '#-':text 0.75 0.25 'Horizontal legend' 'a'
```



5.2.9 Cutting sample

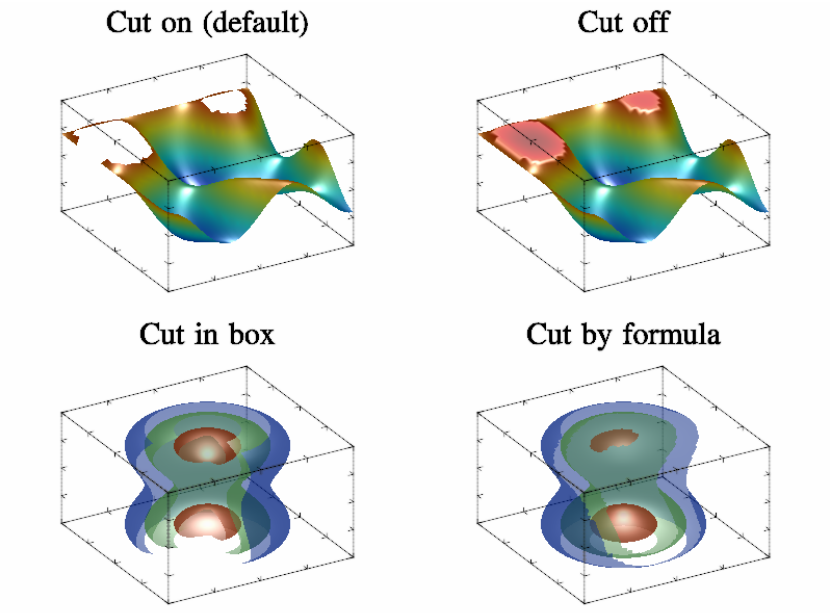
The last common thing which I want to show in this section is how one can cut off points from plot. There are 4 mechanism for that.

- You can set one of coordinate to NAN value. All points with NAN values will be omitted.
- You can enable cutting at edges by `SetCut` function. As result all points out of bounding box will be omitted.
- You can set cutting box by `SetCutBox` function. All points inside this box will be omitted.
- You can define cutting formula by `SetCutOff` function. All points for which the value of formula is nonzero will be omitted. Note, that this is the slowest variant.

Below I place the code which demonstrate last 3 possibilities:

```
call 'prepare2d'
call 'prepare3d'

subplot 2 2 0:title 'Cut on (default)':rotate 50 60
light on:box:surf a; zrange -1 0.5

subplot 2 2 1:title 'Cut off':rotate 50 60
box:surf a; zrange -1 0.5; cut off
```

```
subplot 2 2 2:title 'Cut in box':rotate 50 60:box:alpha on
cut 0 -1 -1 1 0 1.1:surf3 c
cut 0 0 0 0 0 0 # restore back

subplot 2 2 3:title 'Cut by formula':rotate 50 60:box
cut '(z>(x+0.5*y-1)^2-1) & (z>(x-0.5*y-1)^2-1)':surf3 c
```



5.3 Data handling

Class `mglData` contains all functions for the data handling in MathGL (see Chapter 4 [Data processing], page 54). There are several matters why I use class `mglData` but not a single array: it does not depend on type of data (mreal or double), sizes of data arrays are kept with data, memory working is simpler and safer.

5.3.1 Array creation

One can put numbers into the data instance by several ways. Let us do it for square function:

- one can create array by `list` command

  ```
  list a 0 0.04 0.16 0.36 0.64 1
  ```

- another way is to copy from "inline" array

  ```
  copy a [0,0.04,0.16,0.36,0.64,1]
  ```

- next way is to fill the data by textual formula with the help of `modify` function

  ```
  new a 6
  modify a 'x^2'
  ```

- or one may fill the array in some interval and modify it later

  ```
  new a 6
  ```

```
fill a 0 1
modify a 'u^2'
```

- or fill the array using current axis range

```
new a 6
fill a '(x+1)^2/4'
```

or use single line

```
new a 6 '(x+1)^2/4'
```

- finally it can be loaded from file

```
new s 6 '(x+1)^2/4'
save s 'sqr.dat'    # create file first
read a 'sqr.dat'    # load it
```

- at this one can read only part of data

```
new s 6 '(x+1)^2/4'
save s 'sqr.dat'    # create file first
read a 'sqr.dat' 5  # load it
```

Creation of 2d- and 3d-arrays is mostly the same. One can use direct data filling by `list` command

```
list a 11 12 13 | 21 22 23 | 31 32 33
```

or by inline arrays

```
copy a [[11,12,13],[21,22,23],[31,32,33]]
```

Also data can be filled by formula

```
new z 30 40 'sin(pi*x)*cos(pi*y)'
```

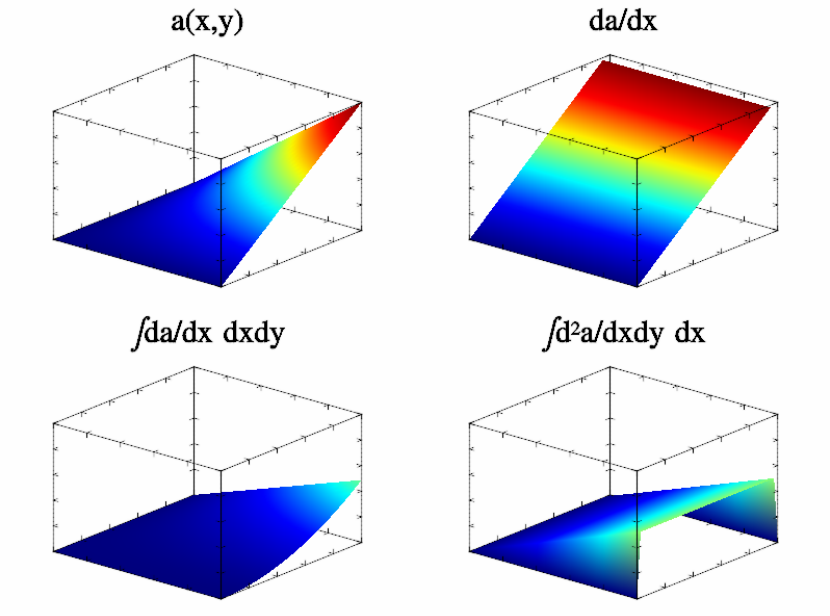
or loaded from a file.

5.3.2 Change data

MathGL has functions for data processing: differentiating, integrating, smoothing and so on (for more detail, see Chapter 4 [Data processing], page 54). Let us consider some examples. The simplest ones are integration and differentiation. The direction in which operation will be performed is specified by textual string, which may contain symbols '`x`', '`y`' or '`z`'. For example, the call of `diff 'x'` will differentiate data along '`x`' direction; the call of `integrate 'xy'` perform the double integration of data along '`x`' and '`y`' directions; the call of `diff2 'xyz'` will apply 3d Laplace operator to data and so on. Example of this operations on 2d array a=x*y is presented in code:

```
ranges 0 1 0 1 0 1:new a 30 40 'x*y'
subplot 2 2 0:title 'a(x,y)':rotate 60 40
surf a:box

subplot 2 2 1:title 'da/dx':rotate 60 40
diff a 'x':surf a:box

subplot 2 2 2:title '\int da/dx dxdy':rotate 60 40
integrate a 'xy':surf a:box
```

```
subplot 2 2 3:title '\int {d^2}a/dxdy dx':rotate 60 40
diff2 a 'y':surf a:box
```

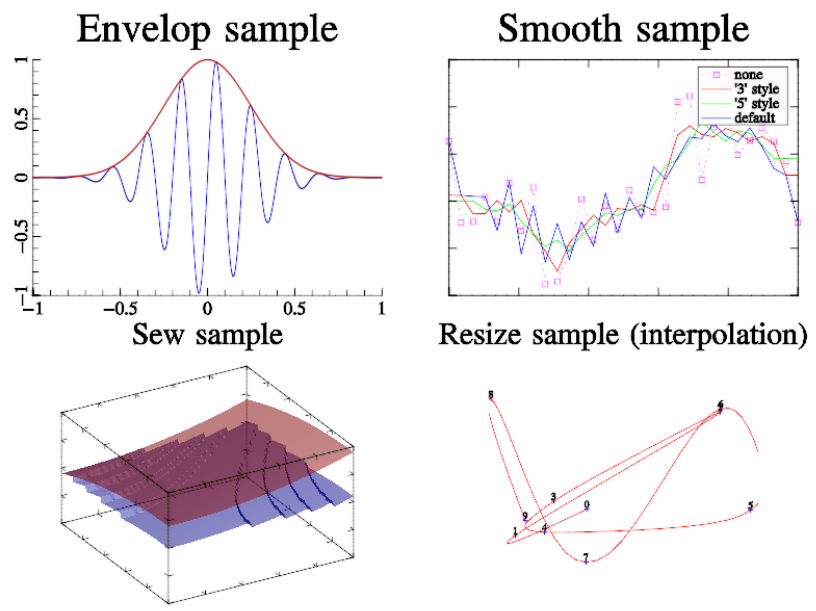


Data smoothing (command [smooth], page 62) is more interesting and important. This function has single argument which define type of smoothing and its direction. Now 3 methods are supported: '3' – linear averaging by 3 points, '5' – linear averaging by 5 points, and default one – quadratic averaging by 5 points.

MathGL also have some amazing functions which is not so important for data processing as useful for data plotting. There are functions for finding envelope (useful for plotting rapidly oscillating data), for data sewing (useful to removing jumps on the phase), for data resizing (interpolation). Let me demonstrate it:

```
subplot 2 2 0 '':title 'Envelop sample'
new d1 1000 'exp(-8*x^2)*sin(10*pi*x)'
axis:plot d1 'b'
envelop d1 'x'
plot d1 'r'

subplot 2 2 1 '':title 'Smooth sample':ranges 0 1 0 1
new y0 30 '0.4*sin(pi*x) + 0.3*cos(1.5*pi*x) - 0.4*sin(2*pi*x)+0.5*rnd'
copy y1 y0:smooth y1 'x3':plot y1 'r';legend '"3" style'
copy y2 y0:smooth y2 'x5':plot y2 'g';legend '"5" style'
copy y3 y0:smooth y3 'x':plot y3 'b';legend 'default'
plot y0 '{m7}:s';legend 'none':legend:box

subplot 2 2 2:title 'Sew sample':rotate 50 60:light on:alpha on
new d2 100 100 'mod((y^2-(1-x)^2)/2,0.1)'
box:surf d2 'b'
sew d2 'xy' 0.1
```

```
surf d2 'r'

subplot 2 2 3:title 'Resize sample (interpolation)'
new x0 10 'rnd':new v0 10 'rnd'
resize x1 x0 100:resize v1 v0 100
plot x0 v0 'b+ ':plot x1 v1 'r-':label x0 v0 '%n'
```



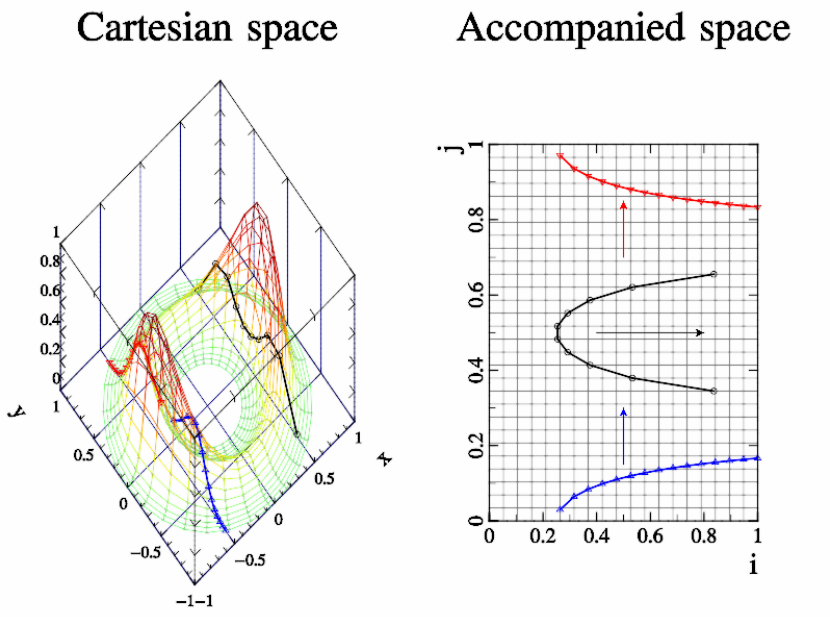
Finally one can create new data arrays on base of the existing one: extract slice, row or column of data ([subdata], page 58), summarize along a direction(s) ([sum], page 60), find distribution of data elements ([hist], page 59) and so on.

Another interesting feature of MathGL is interpolation and root-finding. There are several functions for linear and cubic spline interpolation (see Section 4.8 [Interpolation], page 63). Also there is a function [evaluate], page 59, which do interpolation of data array for values of each data element of index data. It look as indirect access to the data elements.

This function have inverse function [solve], page 59, which find array of indexes at which data array is equal to given value (i.e. work as root finding). But [solve], page 59, function have the issue – usually multidimensional data (2d and 3d ones) have an infinite number of indexes which give some value. This is contour lines for 2d data, or isosurface(s) for 3d data. So, [solve], page 59, function will return index only in given direction, assuming that other index(es) are the same as equidistant index(es) of original data. Let me demonstrate this on the following sample.

```
zrange 0 1
new x 20 30 '(x+2)/3*cos(pi*y)'
new y 20 30 '(x+2)/3*sin(pi*y)'
new z 20 30 'exp(-6*x^2-2*sin(pi*y)^2)'

subplot 2 1 0:title 'Cartesian space':rotate 30 -40
axis 'xyzU':box
```

```
xlabel 'x':ylabel 'y'origin 1 1:grid 'xy'
mesh x y z

# section along 'x' direction
solve u x 0.5 'x'
var v u.nx 0 1
evaluate yy y u v
evaluate xx x u v
evaluate zz z u v
plot xx yy zz 'k2o'

# 1st section along 'y' direction
solve u1 x -0.5 'y'
var v1 u1.nx 0 1
evaluate yy y v1 u1
evaluate xx x v1 u1
evaluate zz z v1 u1
plot xx yy zz 'b2^'

# 2nd section along 'y' direction
solve u2 x -0.5 'y' u1
evaluate yy y v1 u2
evaluate xx x v1 u2
evaluate zz z v1 u2
plot xx yy zz 'r2v'

subplot 2 1 1:title 'Accompanied space'
ranges 0 1 0 1:origin 0 0
axis:box:xlabel 'i':ylabel 'j':grid2 z 'h'

plot u v 'k2o':line 0.4 0.5 0.8 0.5 'kA'
plot v1 u1 'b2^':line 0.5 0.15 0.5 0.3 'bA'
plot v1 u2 'r2v':line 0.5 0.7 0.5 0.85 'rA'
```



5.4 Data plotting

Let me now show how to plot the data. Next section will give much more examples for all plotting functions. Here I just show some basics. MathGL generally has 2 types of plotting functions. Simple variant requires a single data array for plotting, other data (coordinates) are considered uniformly distributed in axis range. Second variant requires data arrays for all coordinates. It allows one to plot rather complex multivalent curves and surfaces (in case of parametric dependencies). Usually each function have one textual argument for plot style and accept options (see Section 2.7 [Command options], page 15).

Note, that the call of drawing function adds something to picture but does not clear the previous plots (as it does in Matlab). Another difference from Matlab is that all setup (like transparency, lightning, axis borders and so on) must be specified **before** plotting functions.

Let start for plots for 1D data. Term "1D data" means that data depend on single index (parameter) like curve in parametric form {x(i),y(i),z(i)}, i=1...n. The textual argument allow you specify styles of line and marks (see Section 2.3 [Line styles], page 9). If this parameter is empty '' then solid line with color from palette is used (see Section 3.2.7 [Palette and colors], page 20).

Below I shall show the features of 1D plotting on base of [plot], page 34, function. Let us start from sinus plot:

```
new y0 50 'sin(pi*x)'
subplot 2 2 0
plot y0:box
```

Style of line is not specified in [plot], page 34, function. So MathGL uses the solid line with first color of palette (this is blue). Next subplot shows array *y1* with 2 rows:

```
subplot 2 2 1
new y1 50 2
fill y1 'cos(pi*(x+y/4))*2/(y+3)'
plot y1:box
```

As previously I did not specify the style of lines. As a result, MathGL again uses solid line with next colors in palette (there are green and red). Now let us plot a circle on the same subplot. The circle is parametric curve $x = cos(\pi t), y = sin(\pi t)$. I will set the color of the circle (dark yellow, '`Y`') and put marks '`+`' at point position:

```
new x 50 'cos(pi*x)'
plot x y0 'Y+'
```

Note that solid line is used because I did not specify the type of line. The same picture can be achieved by [plot], page 34, and [subdata], page 58, functions. Let us draw ellipse by orange dash line:

```
plot y1(:,0) y1(:,1) 'q|'
```

Drawing in 3D space is mostly the same. Let us draw spiral with default line style. Now its color is 4-th color from palette (this is cyan):

```
subplot 2 2 2:rotate 60 40
new z 50 'x'
plot x y0 z:box
```

Functions [plot], page 34, and [subdata], page 58, make 3D curve plot but for single array. Use it to put circle marks on the previous plot:

```
new y2 10 3 'cos(pi*(x+y/2))'
modify y2 '2*x-1' 2
plot y2(:,0) y2(:,1) y2(:,2) 'bo '
```

Note that line style is empty ' ' here. Usage of other 1D plotting functions looks similar:

```
subplot 2 2 3:rotate 60 40
bars x y0 z 'r':box
```

Surfaces [surf], page 40, and other 2D plots (see Section 3.12 [2D plotting], page 39) are drown the same simpler as 1D one. The difference is that the string parameter specifies not the line style but the color scheme of the plot (see Section 2.4 [Color scheme], page 11). Here I draw attention on 4 most interesting color schemes. There is gray scheme where color is changed from black to white (string '`kw`') or from white to black (string '`wk`'). Another scheme is useful for accentuation of negative (by blue color) and positive (by red color) regions on plot (string '`"BbwrR"`'). Last one is the popular "jet" scheme (string '`"BbcyrR"`').

Now I shall show the example of a surface drawing. At first let us switch lightning on

```
light on
```

and draw the surface, considering coordinates x,y to be uniformly distributed in axis range

```
new a0 50 40 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
subplot 2 2 0:rotate 60 40
surf a0:box
```

Color scheme was not specified. So previous color scheme is used. In this case it is default color scheme ("jet") for the first plot. Next example is a sphere. The sphere is parametrically specified surface:

```
new x 50 40 '0.8*sin(pi*x)*cos(pi*y/2)'
new y 50 40 '0.8*cos(pi*x)*cos(pi*y/2)'
```

```
new z 50 40 '0.8*sin(pi*y/2)'
subplot 2 2 1:rotate 60 40
surf x y z 'BbwrR':box
```

I set color scheme to "BbwrR" that corresponds to red top and blue bottom of the sphere.

Surfaces will be plotted for each of slice of the data if *nz*>1. Next example draws surfaces for data arrays with *nz*=3:

```
new a1 50 40 3
modify a1 '0.6*sin(2*pi*x)*sin(3*pi*y)+0.4*cos(3*pi*(x*y))'
modify a1 '0.6*cos(2*pi*x)*cos(3*pi*y)+0.4*sin(3*pi*(x*y))' 1
modify a1 '0.6*cos(2*pi*x)*cos(3*pi*y)+0.4*cos(3*pi*(x*y))' 2
subplot 2 2 2:rotate 60 40
alpha on
surf a1:box
```

Note, that it may entail a confusion. However, if one will use density plot then the picture will look better:

```
subplot 2 2 3:rotate 60 40
dens a1:box
```

Drawing of other 2D plots is analogous. The only peculiarity is the usage of flag '#'. By default this flag switches on the drawing of a grid on plot ([grid], page 32, or [mesh], page 40, for plots in plain or in volume). However, for isosurfaces (including surfaces of rotation [axial], page 42) this flag switches the face drawing off and figure becomes wired.

5.5 1D samples

This section is devoted to visualization of 1D data arrays. 1D means the data which depend on single index (parameter) like curve in parametric form {x(i),y(i),z(i)}, i=1...n. Most of samples will use the same data for plotting. So, I put its initialization in separate function

```
func 'prepare1d'
new y 50 3
modify y '0.7*sin(2*pi*x)+0.5*cos(3*pi*x)+0.2*sin(pi*x)'
modify y 'sin(2*pi*x)' 1
modify y 'cos(2*pi*x)' 2
new x1 50 'x'
new x2 50 '0.05-0.03*cos(pi*x)'
new y1 50 '0.5-0.3*cos(pi*x)'
new y2 50 '-0.3*sin(pi*x)'
return
```

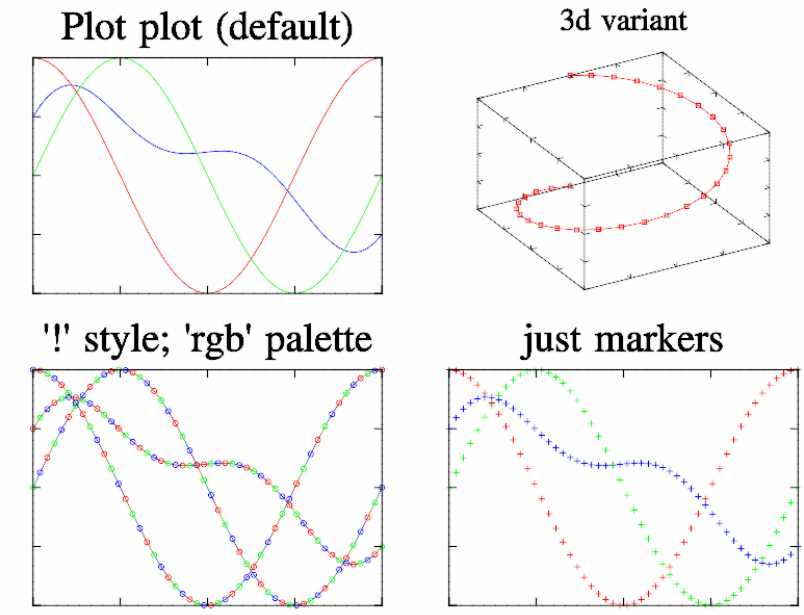
Basically, you can put this text after the script. Note, that you need to terminate main script by [stop], page 4, command before defining a function.

5.5.1 Plot sample

Command [plot], page 34, is most standard way to visualize 1D data array. By default, Plot use colors from palette. However, you can specify manual color/palette, and even set

to use new color for each points by using '!' style. Another feature is ' ' style which draw only markers without line between points. The sample code is:

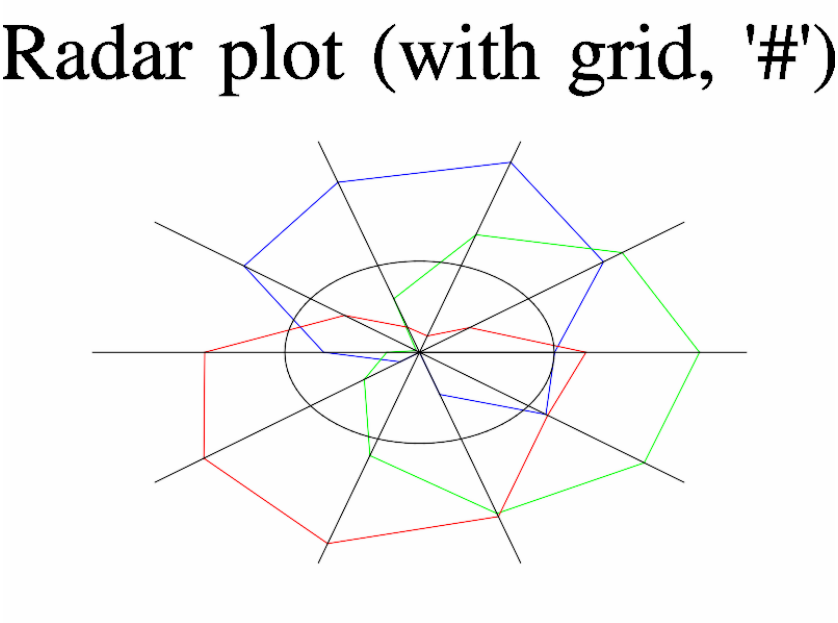
```
call 'prepare1d'
subplot 2 2 0 '':title 'Plot plot (default)':box
plot y

subplot 2 2 2 '':title ''!' style; 'rgb' palette':box
plot y 'o!rgb'

subplot 2 2 3 '':title 'just markers':box
plot y ' +'

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 2 2 1:title '3d variant':rotate 50 60:box
plot xc yc z 'rs'
```



5.5.2 Radar sample

Command [radar], page 34, plot is variant of `Plot` one, which make plot in polar coordinates and draw radial rays in point directions. If you just need a plot in polar coordinates then I recommend to use Section 5.2.3 [Curvilinear coordinates], page 77, or `Plot` in parabolic form with `x=r*cos(fi); y=r*sin(fi);`. The sample code is:

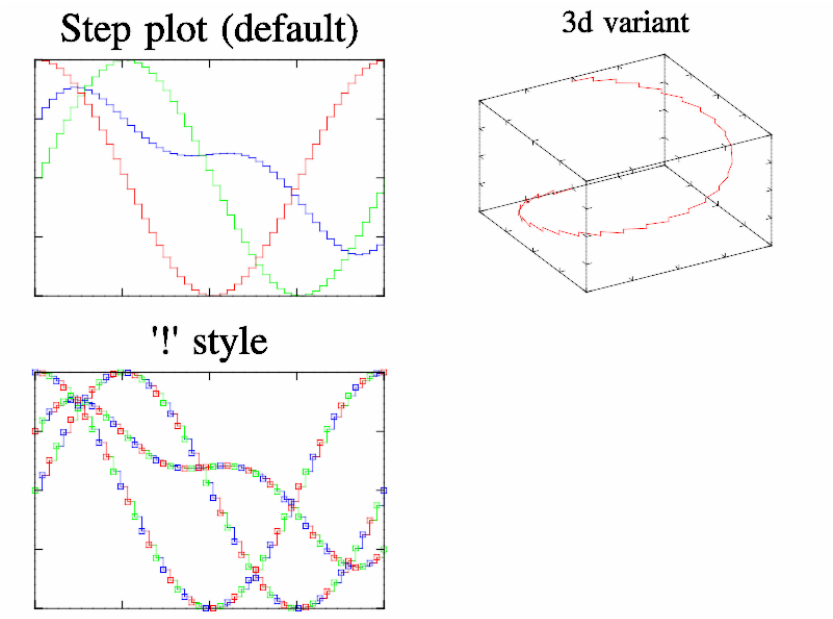
```
new yr 10 3 '0.4*sin(pi*(x+1.5+y/2)+0.1*rnd)'
subplot 1 1 0 '':title 'Radar plot (with grid, "\#")'
radar yr '#'
```



5.5.3 Step sample

Command [step], page 34, plot data as stairs. It have the same options as `Plot`. The sample code is:

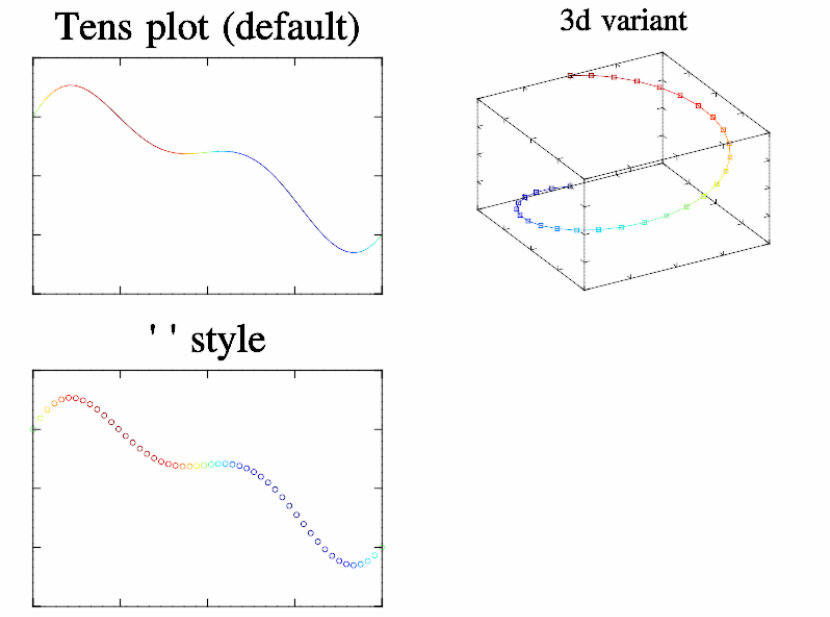
```
call 'prepare1d'
origin 0 0 0:subplot 2 2 0 '':title 'Step plot (default)':box
step y

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 2 2 1:title '3d variant':rotate 50 60:box
step xc yc z 'r'

subplot 2 2 2 '':title '"!" style':box
step y 's!rgb'
```



5.5.4 Tens sample

Command [tens], page 34, is variant of [plot], page 34, with smooth coloring along the curves. At this, color is determined as for surfaces (see Section 2.4 [Color scheme], page 11). The sample code is:

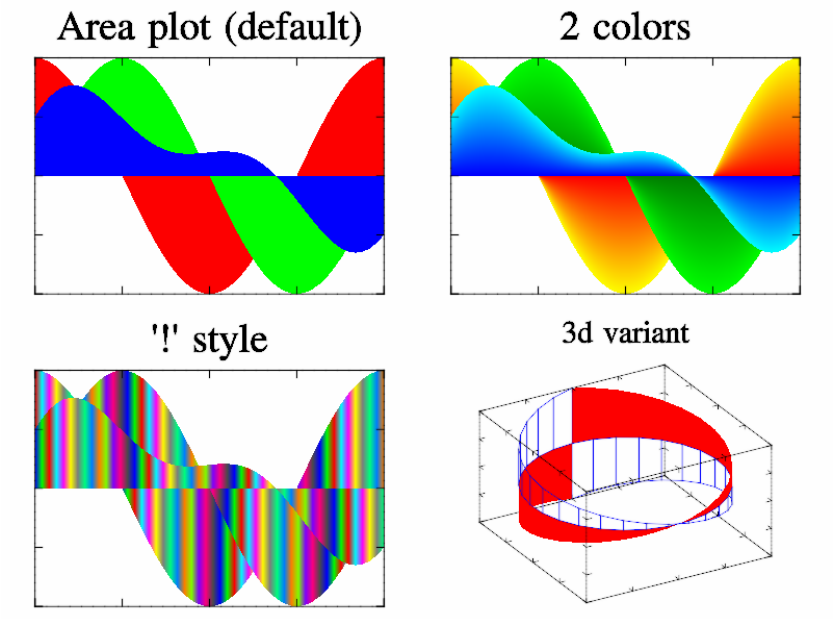
```
call 'prepare1d'
subplot 2 2 0 '':title 'Tens plot (default)':box
tens y(:,0) y(:,1)

subplot 2 2 2 '':title ' style':box
tens y(:,0) y(:,1) 'o '

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 2 2 1:title '3d variant':rotate 50 60:box
tens xc yc z z 's'
```



5.5.5 Area sample

Command [area], page 35, fill the area between curve and axis plane. It support gradient filling if 2 colors per curve is specified. The sample code is:

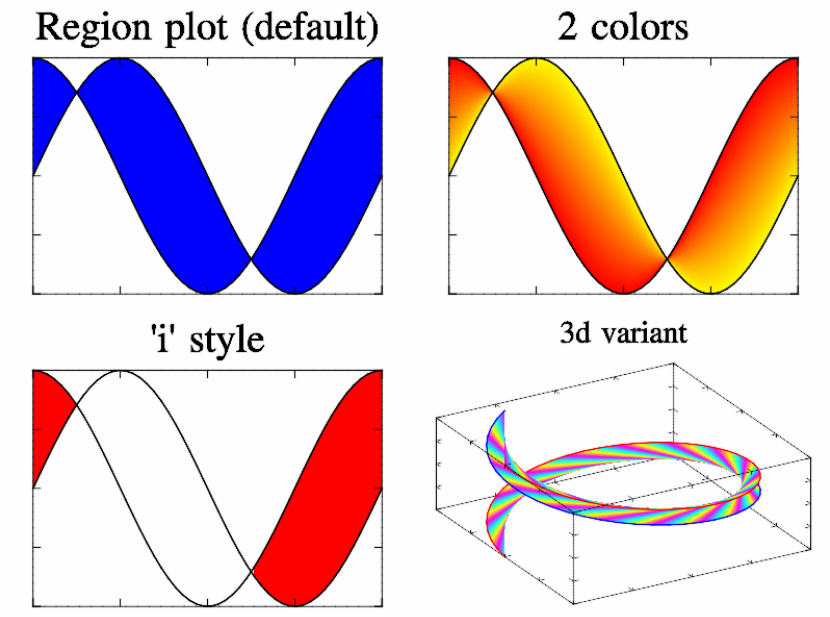
```
call 'prepare1d'
origin 0 0 0
subplot 2 2 0 '':title 'Area plot (default)':box
area y

subplot 2 2 1 '':title '2 colors':box
area y 'cbgGyr'

subplot 2 2 2 '':title '"!" style':box
area y '!'

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 2 2 3:title '3d variant':rotate 50 60:box
area xc yc z 'r':area xc -yc z 'b#'
```



5.5.6 Region sample

Command [region], page 35, fill the area between 2 curves. It support gradient filling if 2 colors per curve is specified. Also it can fill only the region y1<y<y2 if style 'i' is used. The sample code is:

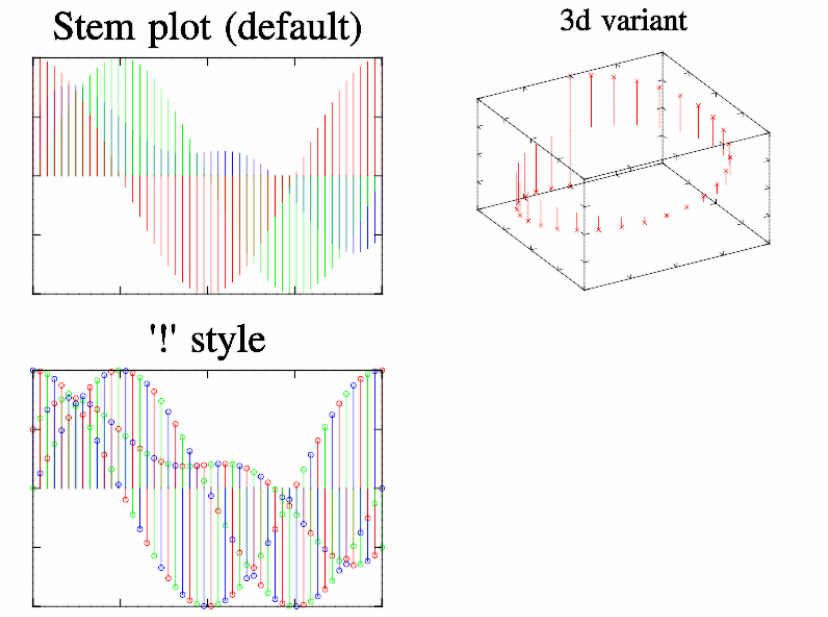
```
call 'prepare1d'
copy y1 y(:,1):copy y2 y(:,2)

subplot 2 2 0 '':title 'Region plot (default)':box
region y1 y2:plot y1 'k2':plot y2 'k2'

subplot 2 2 1 '':title '2 colors':box
region y1 y2 'yr':plot y1 'k2':plot y2 'k2'

subplot 2 2 2 '':title '"i" style':box
region y1 y2 'ir':plot y1 'k2':plot y2 'k2'

subplot 2 2 3 '^_':title '3d variant':rotate 40 60:box
new x1 100 'sin(pi*x)':new y1 100 'cos(pi*x)':new z 100 'x'
new x2 100 'sin(pi*x+pi/3)':new y2 100 'cos(pi*x+pi/3)'
plot x1 y1 z 'r2':plot x2 y2 z 'b2'
region x1 y1 z x2 y2 z 'cmy!'
```



5.5.7 Stem sample

Command [stem], page 35, draw vertical bars. It is most attractive if markers are drawn too. The sample code is:

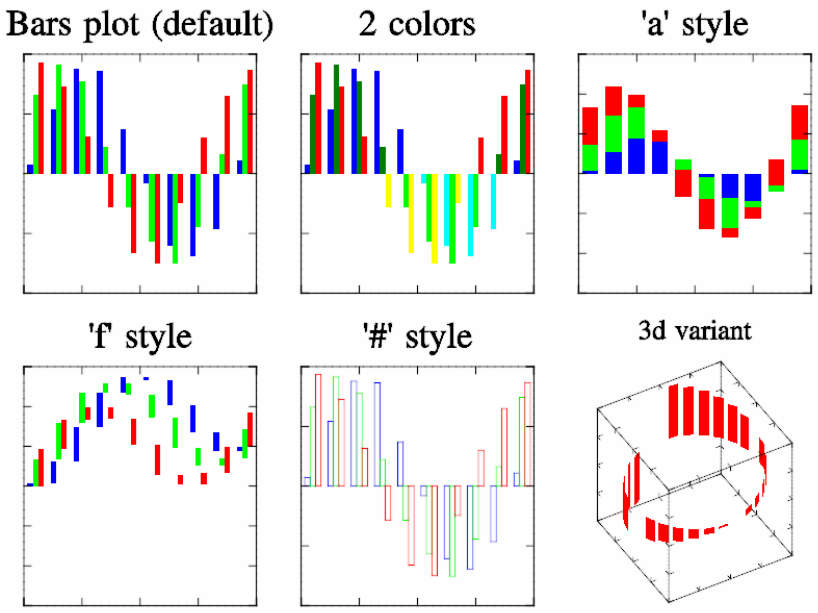
```
call 'prepare1d'
origin 0 0 0:subplot 2 2 0 '':title 'Stem plot (default)':box
stem y

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 2 2 1:title '3d variant':rotate 50 60:box
stem xc yc z 'rx'

subplot 2 2 2 '':title '"!" style':box
stem y 'o!rgb'
```



5.5.8 Bars sample

Command [bars], page 35, draw vertical bars. It have a lot of options: bar-above-bar (‘`a`’ style), fall like (‘`f`’ style), 2 colors for positive and negative values, wired bars (‘`#`’ style), 3D variant. The sample code is:

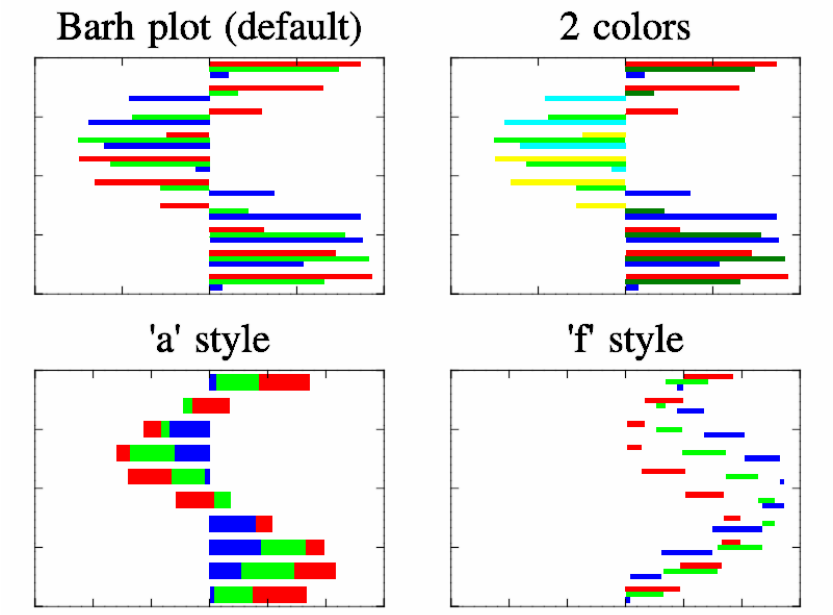
```
new ys 10 3 '0.8*sin(pi*(x+y/4+1.25))+0.2*rnd':origin 0 0 0
subplot 3 2 0 '':title 'Bars plot (default)':box
bars ys

subplot 3 2 1 '':title '2 colors':box
bars ys 'cbgGyr'

subplot 3 2 4 '':title '"\#" style':box
bars ys '#'

new yc 30 'sin(pi*x)':new xc 30 'cos(pi*x)':new z 30 'x'
subplot 3 2 5:title '3d variant':rotate 50 60:box
bars xc yc z 'r'

subplot 3 2 2 '':title '"a" style':ranges -1 1 -3 3:box
bars ys 'a'

subplot 3 2 3 '':title '"f" style':box
bars ys 'f'
```



5.5.9 Barh sample

Command [barh], page 36, is the similar to `Bars` but draw horizontal bars. The sample code is:

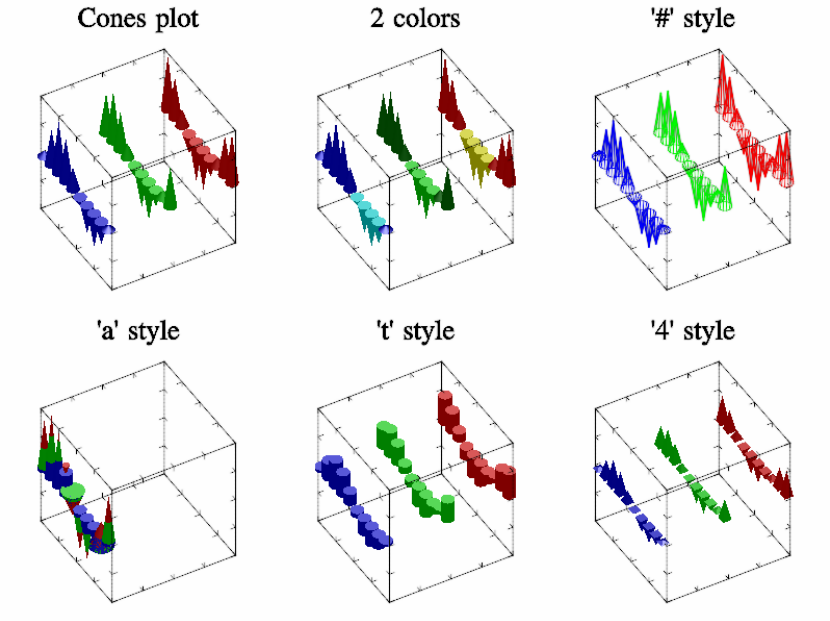
```
new ys 10 3 '0.8*sin(pi*(x+y/4+1.25))+0.2*rnd':origin 0 0 0
subplot 2 2 0 '':title 'Barh plot (default)':box
barh ys

subplot 2 2 1 '':title '2 colors':box
barh ys 'cbgGyr'

ranges -3 3 -1 1:subplot 2 2 2 '':title '"a" style':box:barh ys 'a'
subplot 2 2 3 '': title '"f" style':box
barh ys 'f'
```



5.5.10 Cones sample

Command [cones], page 36, is similar to `Bars` but draw cones. The sample code is:

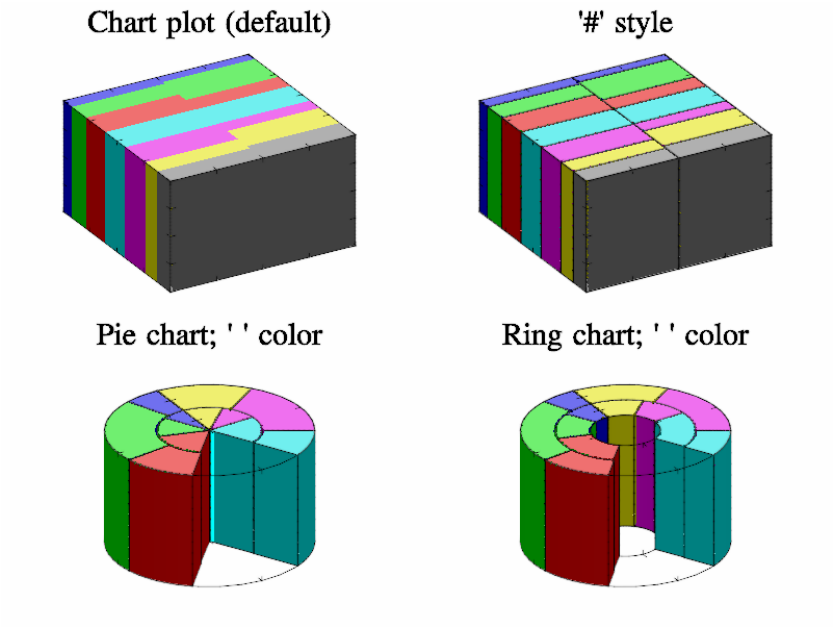
```
new ys 10 3 '0.8*sin(pi*(x+y/4+1.25))+0.2*rnd'
origin 0 0 0:light on
subplot 3 2 0:title 'Cones plot':rotate 50 60:box
cones ys

subplot 3 2 1:title '2 colors':rotate 50 60:box
cones ys 'cbgGyr'

subplot 3 2 2:title '"\#" style':rotate 50 60:box
cones ys '#'

subplot 3 2 3:title '"a" style':rotate 50 60:zrange -2 2:box
cones ys 'a'

subplot 3 2 4:title '"t" style':rotate 50 60:box
cones ys 't'

subplot 3 2 5:title '"4" style':rotate 50 60:box
cones ys '4'
```



5.5.11 Chart sample

Command [chart], page 36, draw colored boxes with width proportional to data values. Use ‘ ’ for empty box. Plot looks most attractive in polar coordinates – well known pie chart. The sample code is:

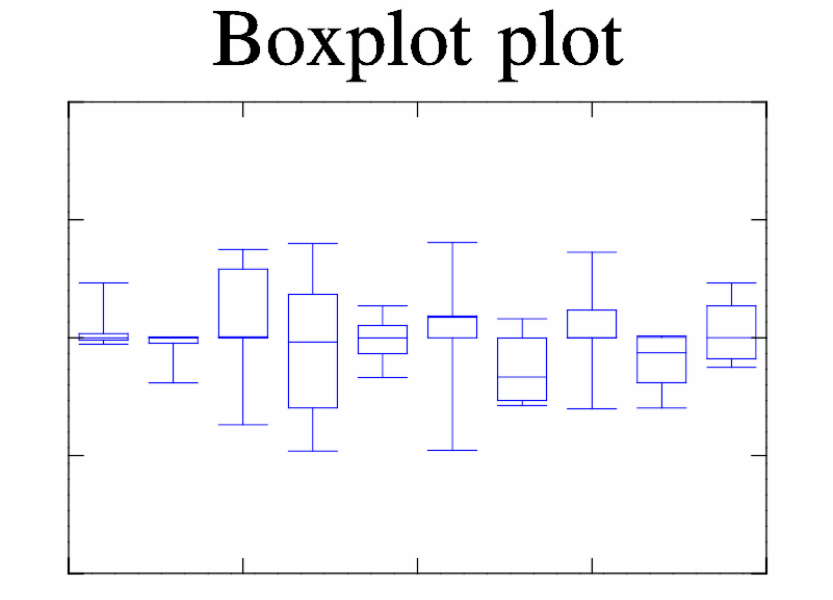
```
new ch 7 2 'rnd+0.1':light on
subplot 2 2 0:title 'Chart plot (default)':rotate 50 60:box
chart ch

subplot 2 2 1:title '"\#" style':rotate 50 60:box
chart ch '#'

subplot 2 2 2:title 'Pie chart; " " color':rotate 50 60:
axis '(y+1)/2*cos(pi*x)' '(y+1)/2*sin(pi*x)' '':box
chart ch 'bgr cmy#'

subplot 2 2 3:title 'Ring chart; " " color':rotate 50 60:
axis '(y+2)/3*cos(pi*x)' '(y+2)/3*sin(pi*x)' '':box
chart ch 'bgr cmy#'
```



5.5.12 BoxPlot sample

Command [boxplot], page 36, draw box-and-whisker diagram. The sample code is:

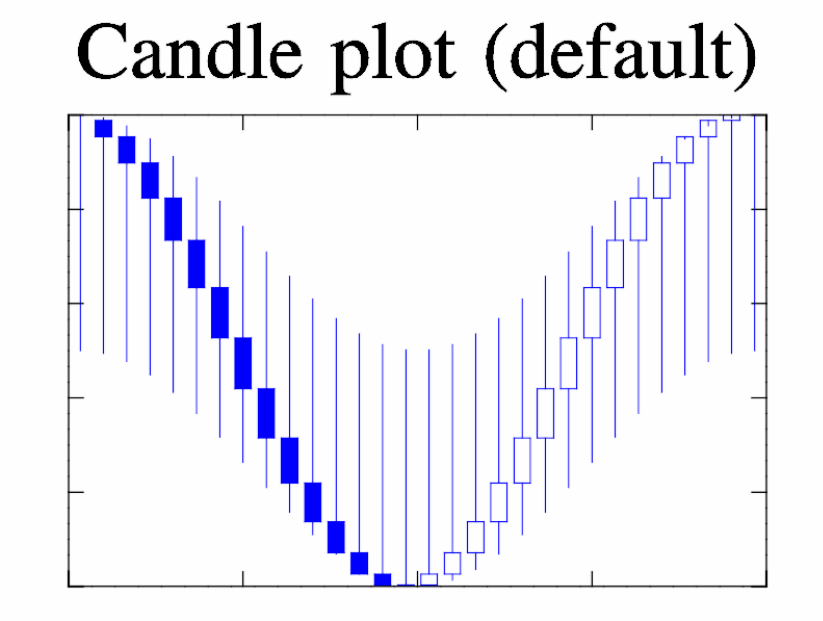
```
new a 10 7 '(2*rnd-1)^3/2'
subplot 1 1 0 '':title 'Boxplot plot':box
boxplot a
```



5.5.13 Candle sample

Command [candle], page 37, draw candlestick chart. This is a combination of a line-chart and a bar-chart, in that each bar represents the range of price movement over a given time interval. The sample code is:

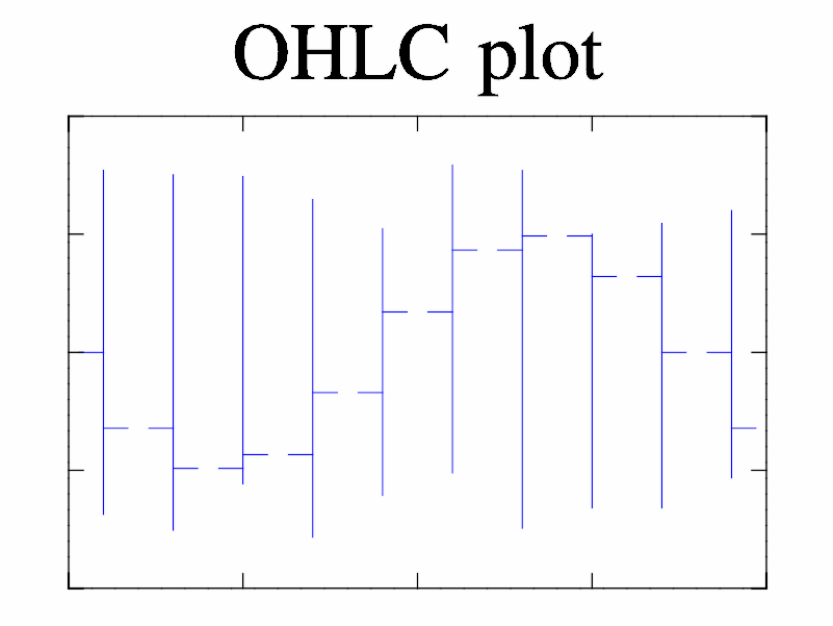
```
new y 30 'sin(pi*x/2)^2':copy y1 y/2:copy y2 (y+1)/2
subplot 1 1 0 '':title 'Candle plot (default)':yrange 0 1:box
candle y y1 y2
```



5.5.14 OHLC sample

Command [ohlc], page 37, draw Open-High-Low-Close diagram. This diagram show vertical line for between maximal(high) and minimal(low) values, as well as horizontal lines before/after vertical line for initial(open)/final(close) values of some process. The sample code is:

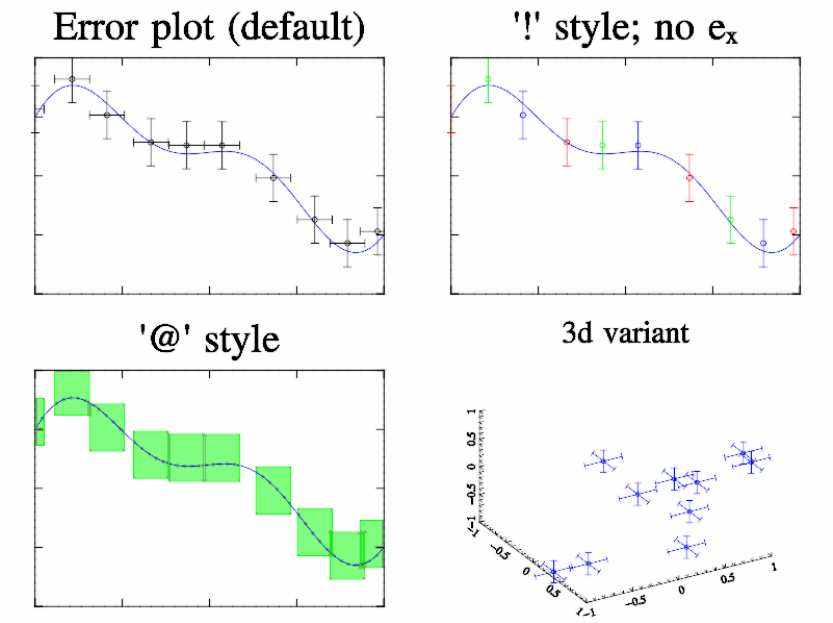
```
new o 10 '0.5*sin(pi*x)'
new c 10 '0.5*sin(pi*(x+2/9))'
new l 10 '0.3*rnd-0.8'
new h 10 '0.3*rnd+0.5'
subplot 1 1 0 '':title 'OHLC plot':box
ohlc o h l c
```



5.5.15 Error sample

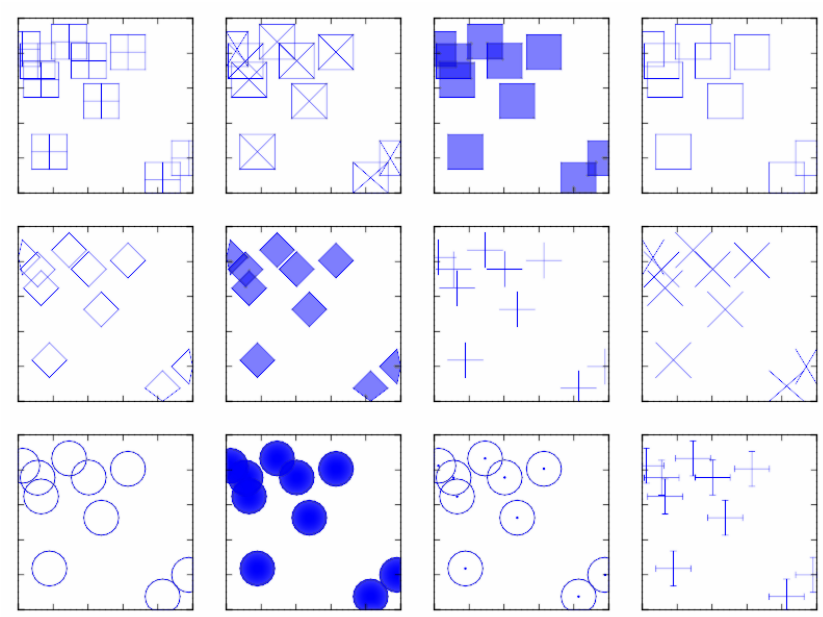
Command [error], page 37, draw error boxes around the points. You can draw default boxes or semi-transparent symbol (like marker, see Section 2.3 [Line styles], page 9). Also you can set individual color for each box. The sample code is:

```
call 'prepare1d'
new y 50 '0.7*sin(pi*x-pi) + 0.5*cos(3*pi*(x+1)/2) + 0.2*sin(pi*(x+1)/2)'
new x0 10 'x + 0.1*rnd-0.05':new ex 10 '0.1':new ey 10 '0.2'
new y0 10 '0.7*sin(pi*x-pi) + 0.5*cos(3*pi*(x+1)/2) + 0.2*sin(pi*(x+1)/2) + 0.2*rnd-0.1'

subplot 2 2 0 '':title 'Error plot (default)':box:plot y
error x0 y0 ex ey 'k'

subplot 2 2 1 '':title '"!" style; no e_x':box:plot y
error x0 y0 ey 'o!rgb'

subplot 2 2 2 '':title '"\@" style':box:plot y
error x0 y0 ex ey '@'; alpha 0.5

subplot 2 2 3:title '3d variant':rotate 50 60:axis
for $1 0 9
  errbox 2*rnd-1 2*rnd-1 2*rnd-1 0.2 0.2 0.2 'bo'
next
```



Additionally, you can use solid large "marks" instead of error boxes by selecting proper style.

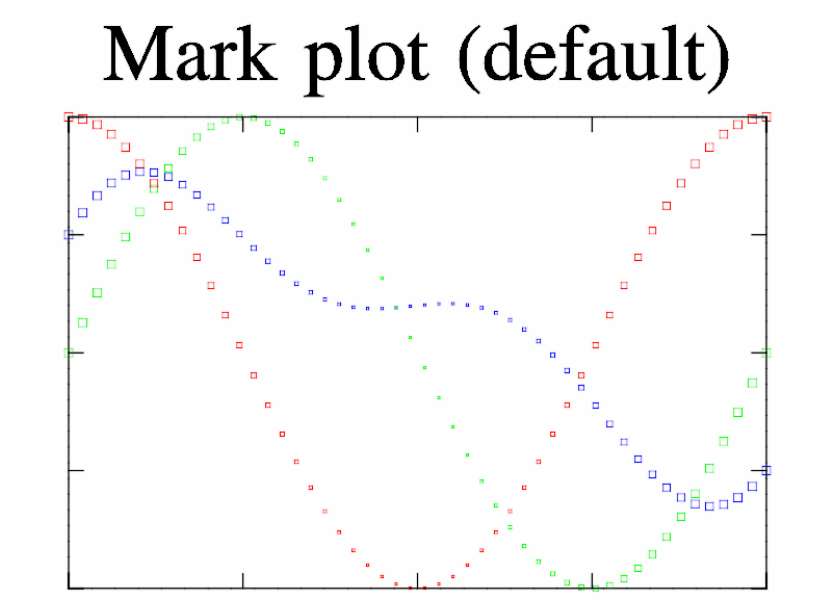
```
new x0 10 'rnd':new ex 10 '0.1'
new y0 10 'rnd':new ey 10 '0.1'
ranges 0 1 0 1
subplot 4 3 0 '':box:error x0 y0 ex ey '#+@'
subplot 4 3 1 '':box:error x0 y0 ex ey '#x@'
subplot 4 3 2 '':box:error x0 y0 ex ey '#s@'; alpha 0.5
subplot 4 3 3 '':box:error x0 y0 ex ey 's@'
subplot 4 3 4 '':box:error x0 y0 ex ey 'd@'
subplot 4 3 5 '':box:error x0 y0 ex ey '#d@'; alpha 0.5
subplot 4 3 6 '':box:error x0 y0 ex ey '+@'
subplot 4 3 7 '':box:error x0 y0 ex ey 'x@'
subplot 4 3 8 '':box:error x0 y0 ex ey 'o@'
subplot 4 3 9 '':box:error x0 y0 ex ey '#o@'; alpha 0.5
subplot 4 3 10 '':box:error x0 y0 ex ey '#.@'
subplot 4 3 11 '':box:error x0 y0 ex ey; alpha 0.5
```



5.5.16 Mark sample

Command [mark], page 37, draw markers at points. It is mostly the same as `Plot` but marker size can be variable. The sample code is:

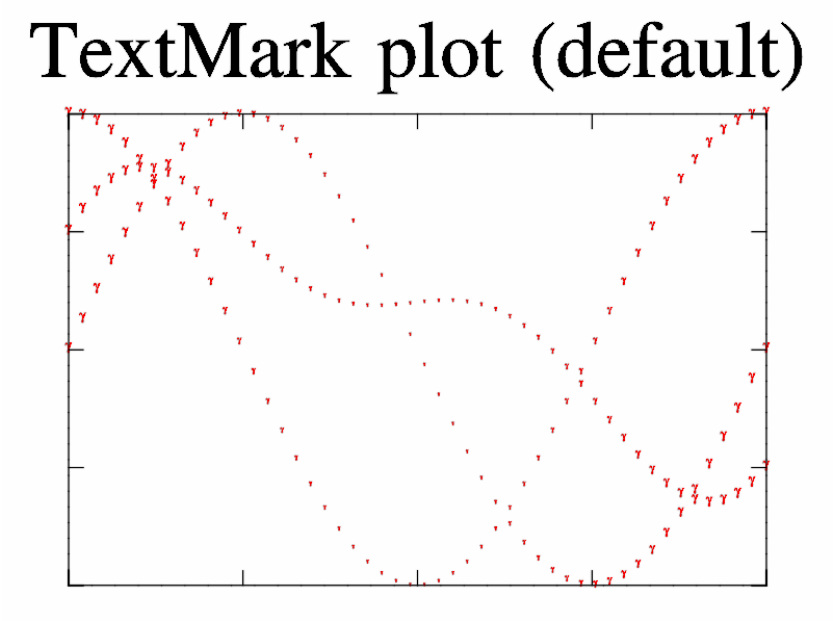
```
call 'prepare1d'
subplot 1 1 0 '':title 'Mark plot (default)':box
mark y y1 's'
```



5.5.17 TextMark sample

Command [textmark], page 38, like `Mark` but draw text instead of markers. The sample code is:

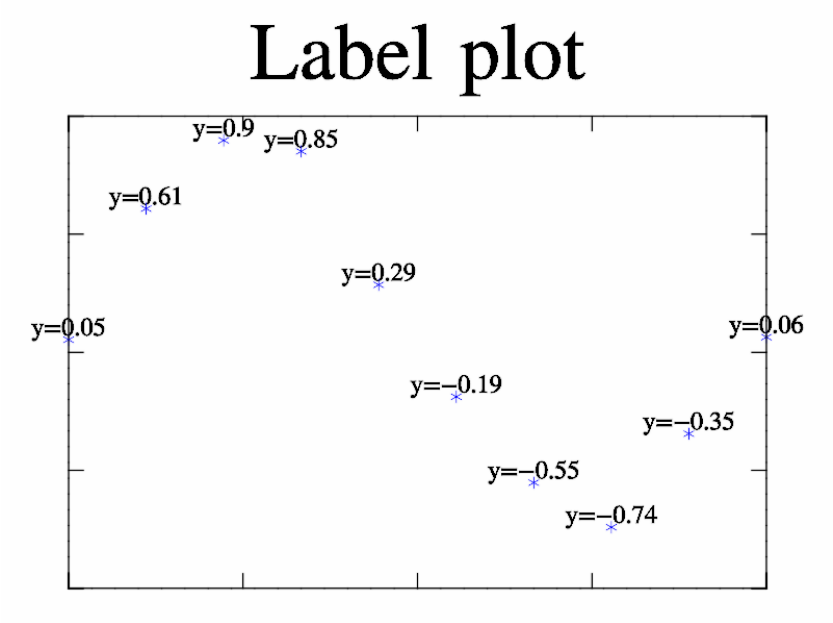
```
call 'prepare1d'
subplot 1 1 0 '':title 'TextMark plot (default)':box
textmark y y1 '\gamma' 'r'
```



5.5.18 Label sample

Command [label], page 38, print text at data points. The string may contain '`%x`', '`%y`', '`%z`' for x-, y-, z-coordinates of points, '`%n`' for point index. The sample code is:

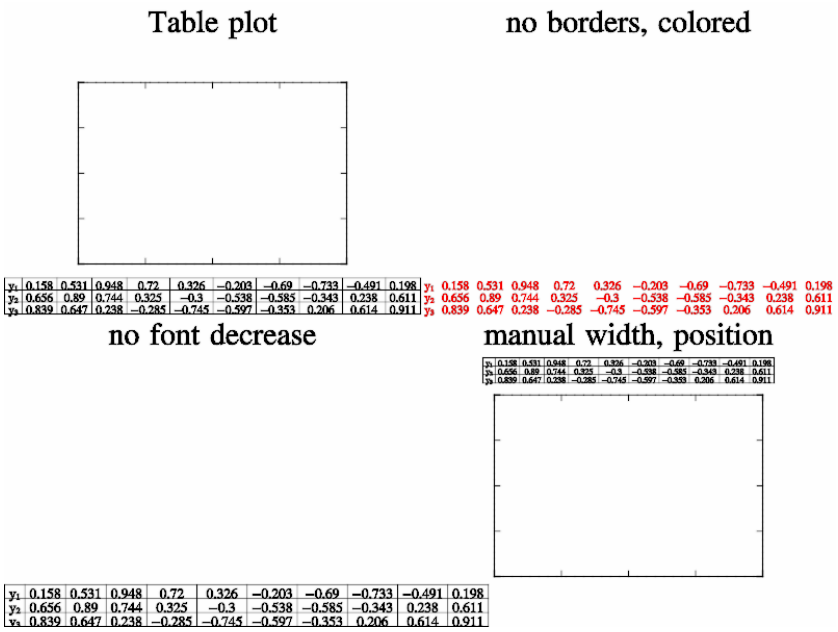
```
new ys 10 '0.2*rnd-0.8*sin(pi*x)'
subplot 1 1 0 '':title 'Label plot':box
plot ys ' *':label ys 'y=%y'
```



5.5.19 Table sample

Command [table], page 38, draw table with data values. The sample code is:

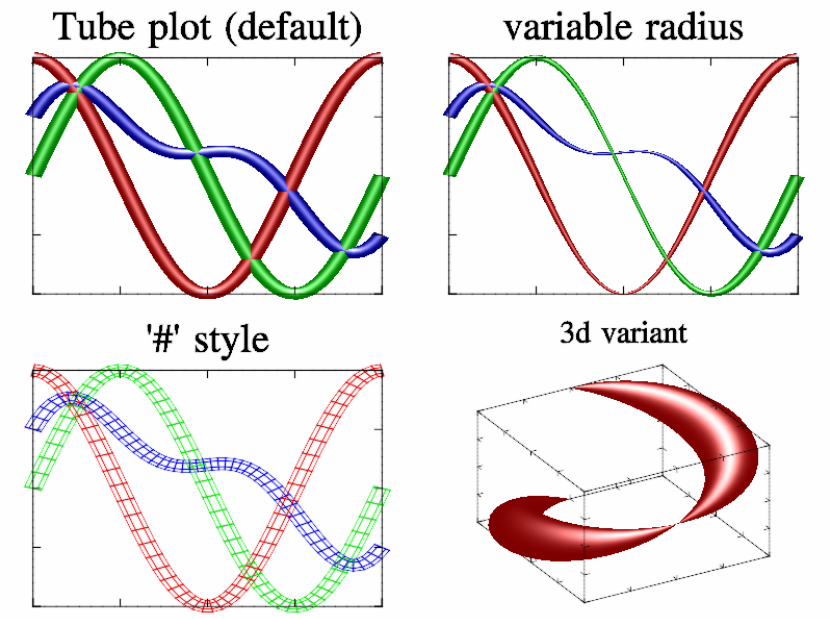
```
new ys 10 3 '0.8*sin(pi*(x+y/4+1.25))+0.2*rnd'
subplot 2 2 0:title 'Table sample':box
table ys 'y_1\n{}y_2\n{}y_3'

subplot 2 2 1:title 'no borders, colored'
table ys 'y_1\n{}y_2\n{}y_3' 'r|'

subplot 2 2 2:title 'no font decrease'
table ys 'y_1\n{}y_2\n{}y_3' '#'

subplot 2 2 3:title 'manual width and position':box
table 0.5 0.95 ys 'y_1\n{}y_2\n{}y_3' '#';value 0.7
```



5.5.20 Tube sample

Command [tube], page 39, draw tube with variable radius. The sample code is:

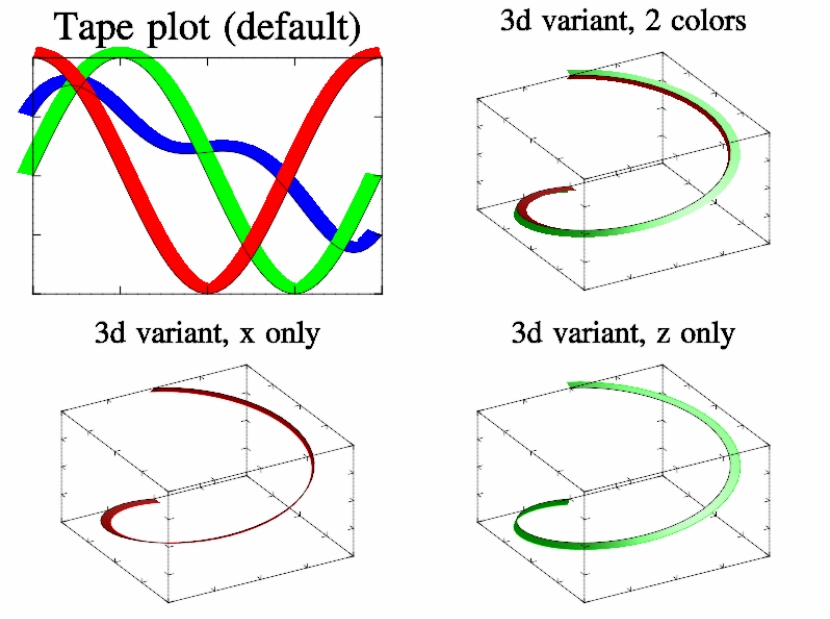
```
light on:call 'prepare1d'
new yc 50 'sin(pi*x)':new xc 50 'cos(pi*x)':new z 50 'x':divto y1 20

subplot 2 2 0 '':title 'Tube plot (default)':box
tube y 0.05

subplot 2 2 1 '':title 'variable radius':box
tube y y1

subplot 2 2 2 '':title '"\#" style':box
tube y 0.05 '#'

subplot 2 2 3:title '3d variant':rotate 50 60:box
tube xc yc z y2 'r'
```



5.5.21 Tape sample

Command [tape], page 35, draw tapes which rotate around the curve as normal and binormal. The sample code is:

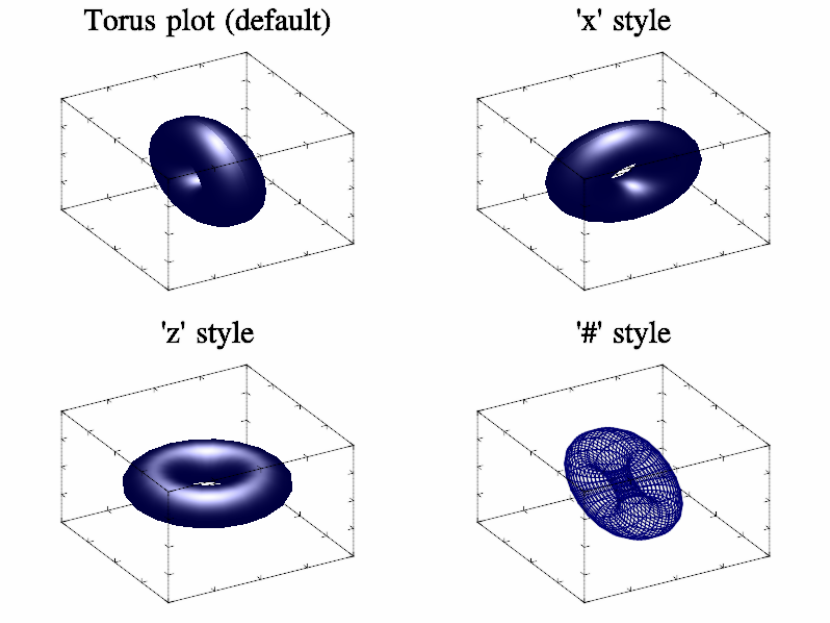
```
call 'prepare1d'
new yc 50 'sin(pi*x)':new xc 50 'cos(pi*x)':new z 50 'x'

subplot 2 2 0 '':title 'Tape plot (default)':box
tape y:plot y 'k'

subplot 2 2 1:title '3d variant, 2 colors':rotate 50 60:light on:box
plot xc yc z 'k':tape xc yc z 'rg'

subplot 2 2 2:title '3d variant, x only':rotate 50 60:box
plot xc yc z 'k':tape xc yc z 'xr':tape xc yc z 'xr#'

subplot 2 2 3:title '3d variant, z only':rotate 50 60:box
plot xc yc z 'k':tape xc yc z 'zg':tape xc yc z 'zg#'
```



5.5.22 Torus sample

Command [torus], page 39, draw surface of the curve rotation. The sample code is:

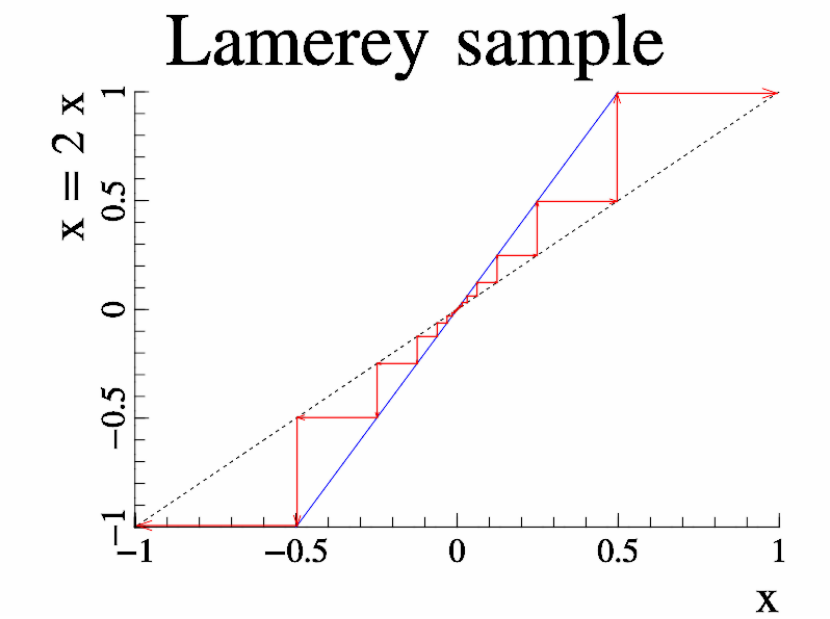
```
call 'prepare1d'
subplot 2 2 0:title 'Torus plot (default)':light on:rotate 50 60:box
torus y1 y2

subplot 2 2 1:title '"x" style':light on:rotate 50 60:box
torus y1 y2 'x'

subplot 2 2 2:title '"z" style':light on:rotate 50 60:box
torus y1 y2 'z'

subplot 2 2 3:title '"\#" style':light on:rotate 50 60:box
torus y1 y2 '#'
```



5.5.23 Lamerey sample

Function [lamerey], page 39, draw Lamerey diagram. The sample code is:

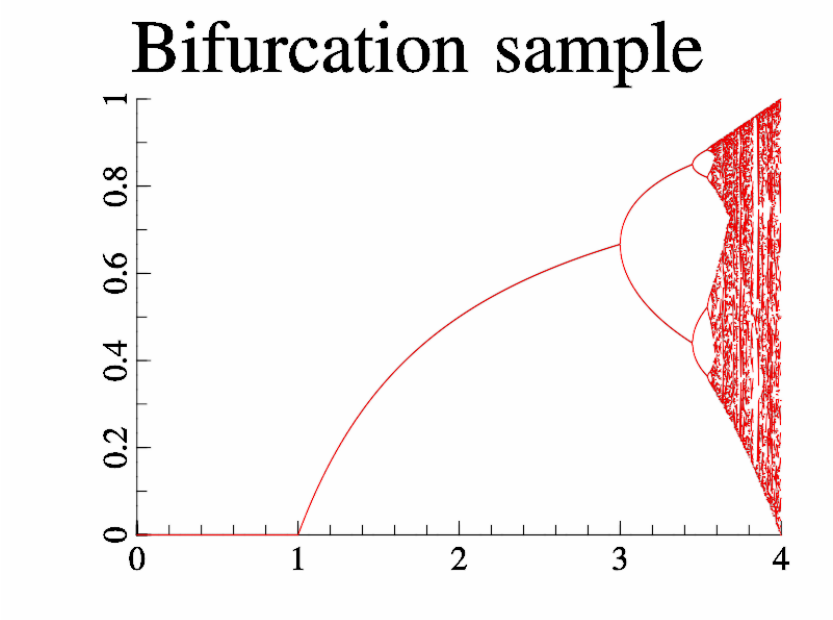
```
subplot 1 1 0 '<_':title 'Lamerey sample'
axis:xlabel '\i x':ylabel '\bar{\i x} = 2 \i{x}'
fplot 'x' 'k='
fplot '2*x' 'b'

lamerey 0.00097 '2*x' 'rv~';size 2
lamerey -0.00097 '2*x' 'rv~';size 2
```



5.5.24 Bifurcation sample

Function [bifurcation], page 39, draw Bifurcation diagram for logistic map. The sample code is:

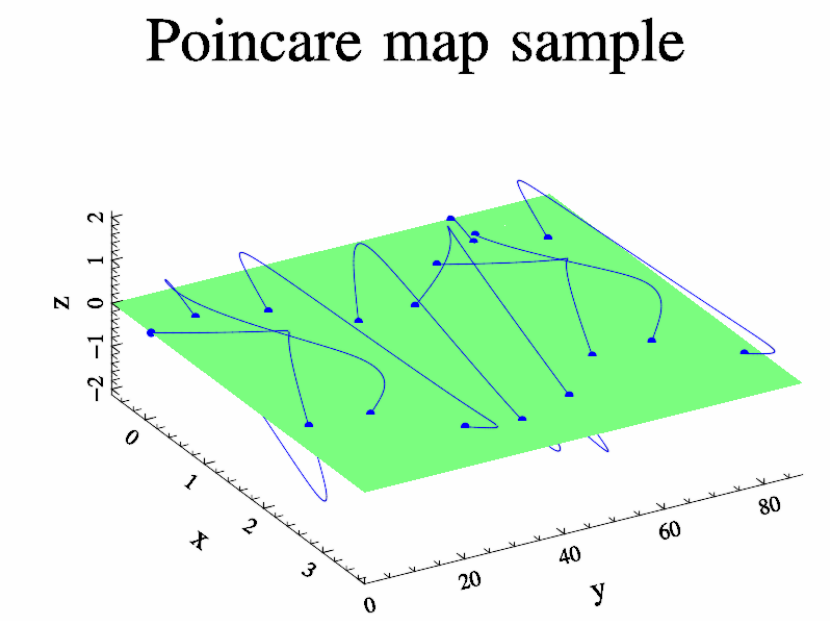
```
subplot 1 1 0 '<_':title 'Bifurcation sample'
ranges 0 4 0 1:axis
bifurcation 0.005 'x*y*(1-y)' 'r'
```



5.5.25 Pmap sample

Function [pmap], page 39, draw Poincare map – show intersections of the curve and the surface. The sample code is:

```
subplot 1 1 0 '<_^':title 'Poincare map sample'
ode r 'cos(y)+sin(z);cos(z)+sin(x);cos(x)+sin(y)' 'xyz' [0.1,0,0] 0.1 100
rotate 40 60:copy x r(0):copy y r(1):copy z r(2)
ranges x y z
axis:plot x y z 'b':fsurf '0'
xlabel '\i x' 0:ylabel '\i y' 0:zlabel '\i z'

pmap x y z z 'b#o'
```



5.6 2D samples

This section is devoted to visualization of 2D data arrays. 2D means the data which depend on 2 indexes (parameters) like matrix z(i,j)=z(x(i),y(j)), i=1...n, j=1...m or in parametric form {x(i,j),y(i,j),z(i,j)}. Most of samples will use the same data for plotting. So, I put its initialization in separate function

```
func 'prepare2d'
new a 50 40 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
new b 50 40 '0.6*cos(pi*(x+1))*cos(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
return
```

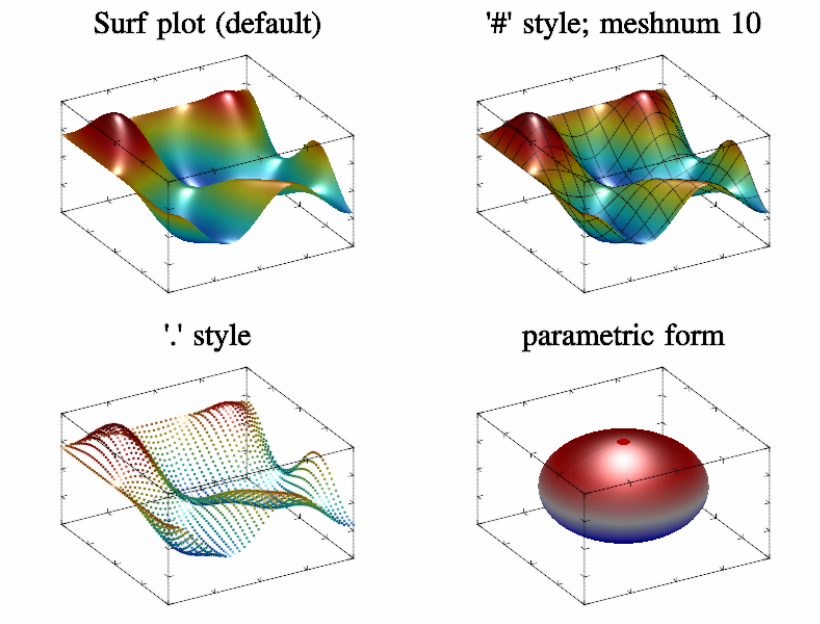
Basically, you can put this text after the script. Note, that you need to terminate main script by [stop], page 4, command before defining a function.

5.6.1 Surf sample

Command [surf], page 40, is most standard way to visualize 2D data array. `Surf` use color scheme for coloring (see Section 2.4 [Color scheme], page 11). You can use '`#`' style for drawing black meshes on the surface. The sample code is:

```
call 'prepare2d'
subplot 2 2 0:title 'Surf plot (default)':rotate 50 60:light on:box:surf a

subplot 2 2 1:title '"\#" style; meshnum 10':rotate 50 60:box
surf a '#'; meshnum 10

subplot 2 2 2:title '"." style':rotate 50 60:box
surf a '.'

new x 50 40 '0.8*sin(pi*x)*sin(pi*(y+1)/2)'
new y 50 40 '0.8*cos(pi*x)*sin(pi*(y+1)/2)'
```

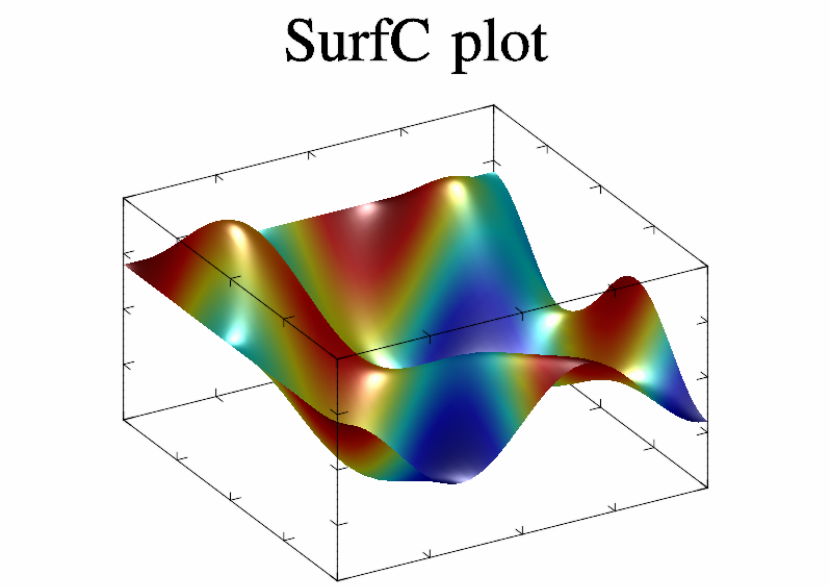
```
new z 50 40 '0.8*cos(pi*(y+1)/2)'
subplot 2 2 3:title 'parametric form':rotate 50 60:box
surf x y z 'BbwrR'
```



5.6.2 SurfC sample

Command [surfc], page 44, is similar to [surf], page 40, but its coloring is determined by another data. The sample code is:

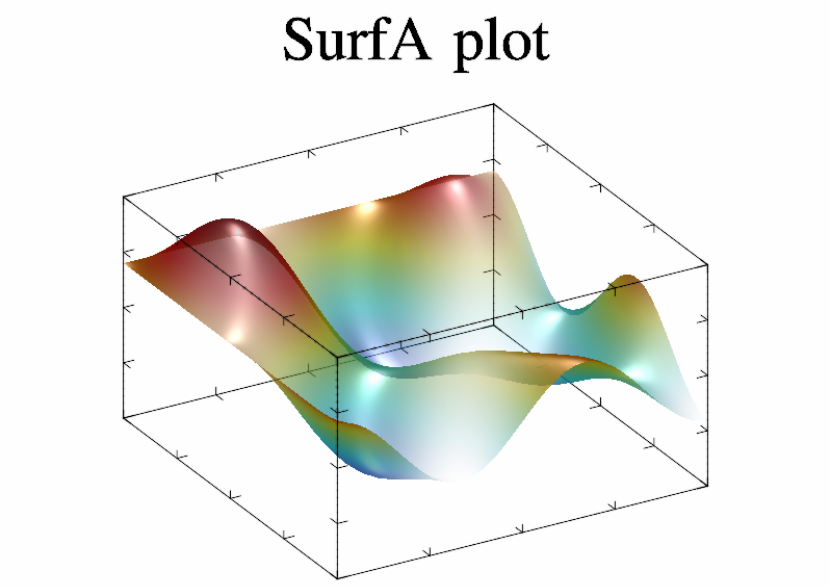
```
call 'prepare2d'
title 'SurfC plot':rotate 50 60:light on:box
surfc a b
```



5.6.3 SurfA sample

Command [surfa], page 45, is similar to [surf], page 40, but its transparency is determined by another data. The sample code is:

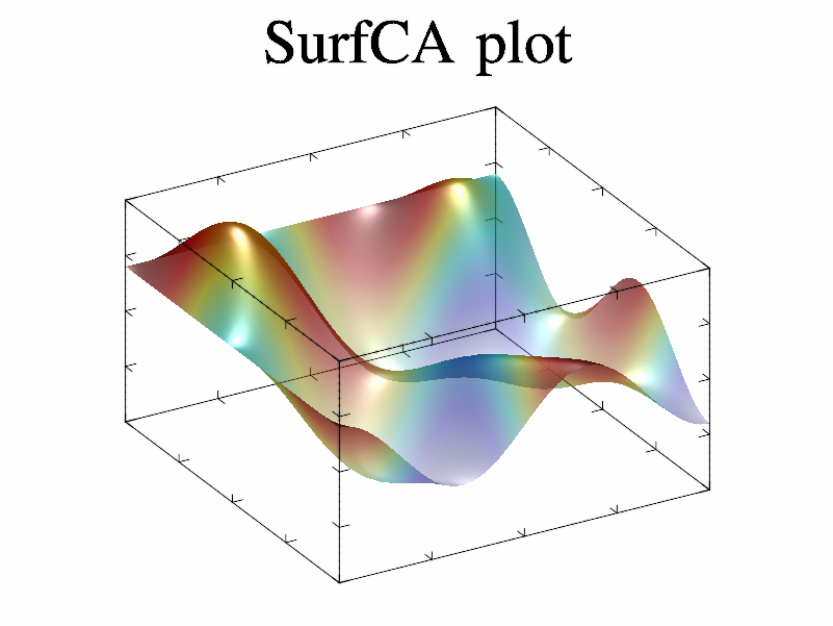
```
call 'prepare2d'
title 'SurfA plot':rotate 50 60:light on:alpha on:box
surfa a b
```



5.6.4 SurfCA sample

Command [surfca], page 45, is similar to [surf], page 40, but its color and transparency is determined by another data. The sample code is:

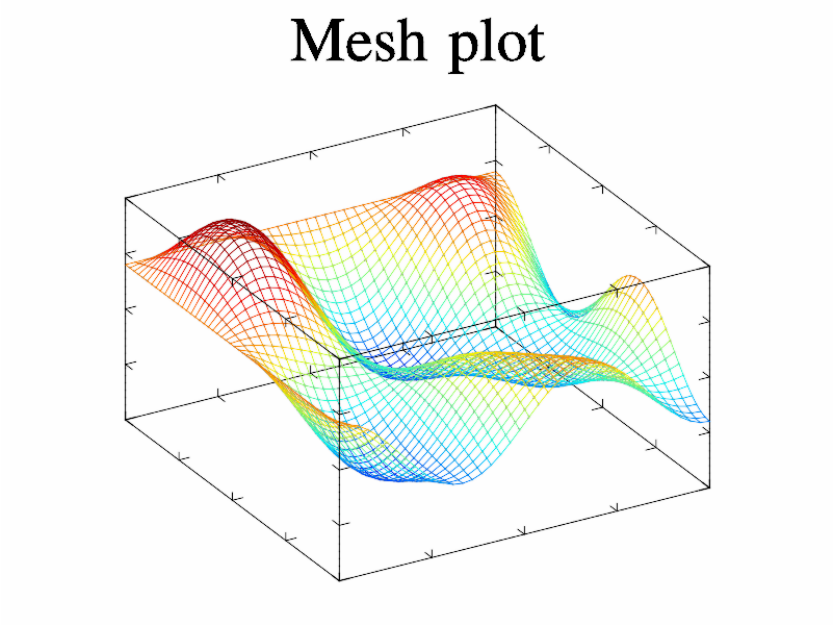
```
call 'prepare2d'
title 'SurfCA plot':rotate 50 60:light on:alpha on:box
surfa a b a
```



5.6.5 Mesh sample

Command [mesh], page 40, draw wired surface. You can use [meshnum], page 19, for changing number of lines to be drawn. The sample code is:

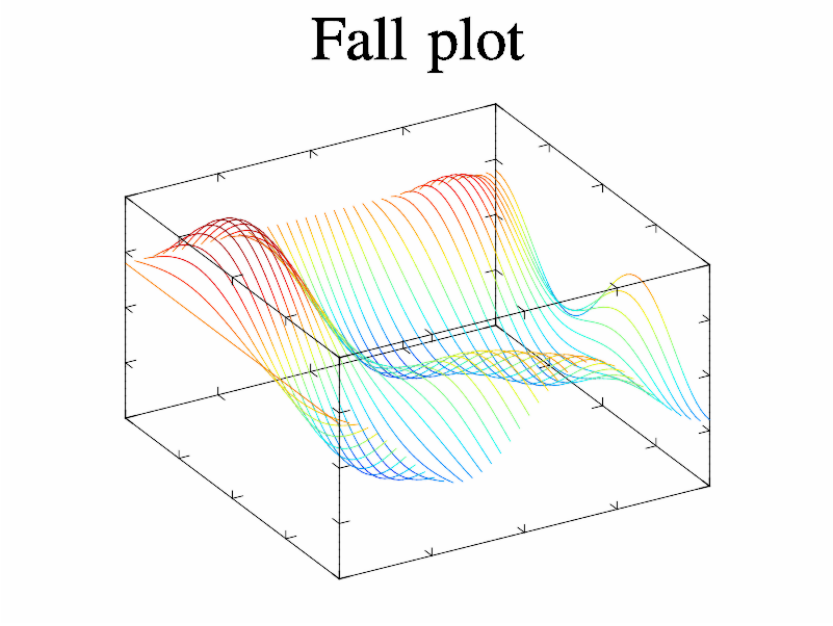
```
call 'prepare2d'
title 'Mesh plot':rotate 50 60:box
mesh a
```



5.6.6 Fall sample

Command [fall], page 40, draw waterfall surface. You can use [meshnum], page 19, for changing number of lines to be drawn. Also you can use '`x`' style for drawing lines in other direction. The sample code is:

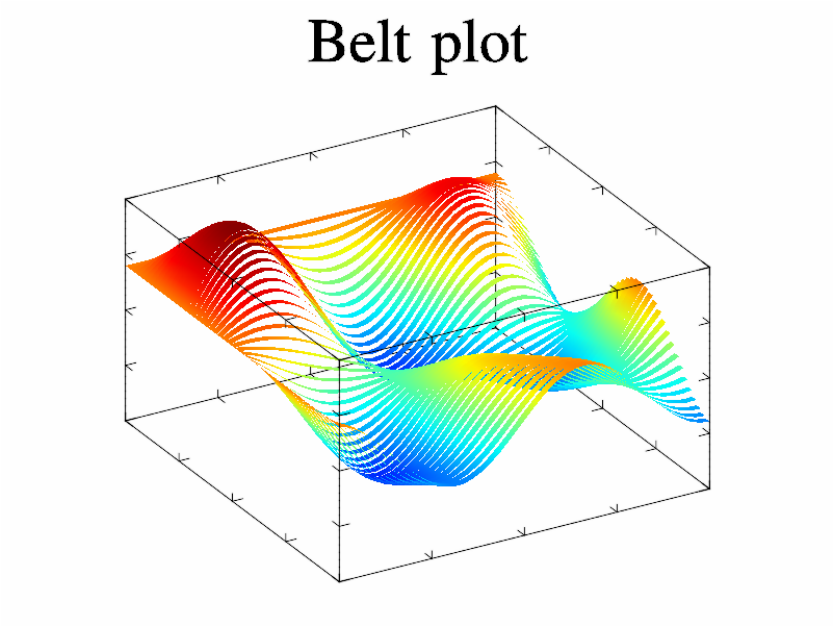
```
call 'prepare2d'
title 'Fall plot':rotate 50 60:box
fall a
```



5.6.7 Belt sample

Command [belt], page 40, draw surface by belts. You can use '`x`' style for drawing lines in other direction. The sample code is:

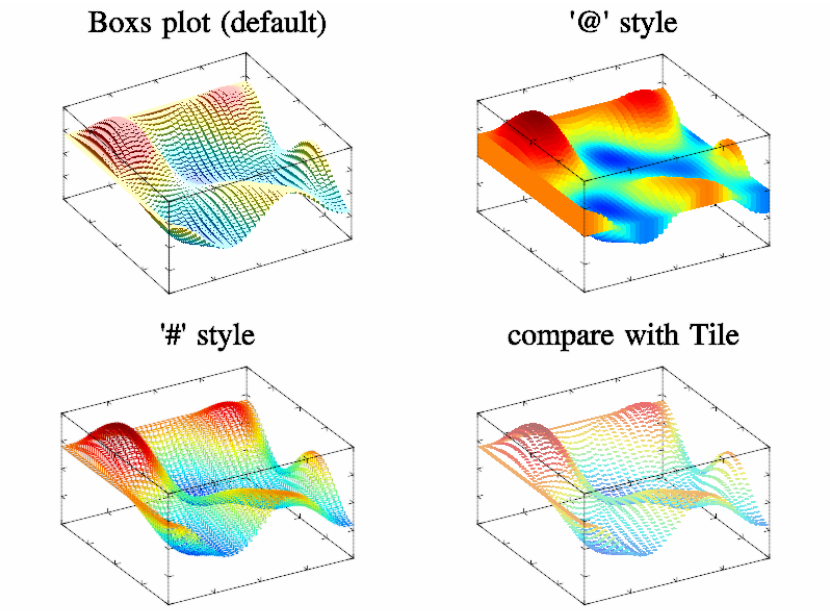
```
call 'prepare2d'
title 'Belt plot':rotate 50 60:box
belt a
```



5.6.8 Boxs sample

Command [boxs], page 40, draw surface by boxes. You can use '`#`' for drawing wire plot. The sample code is:

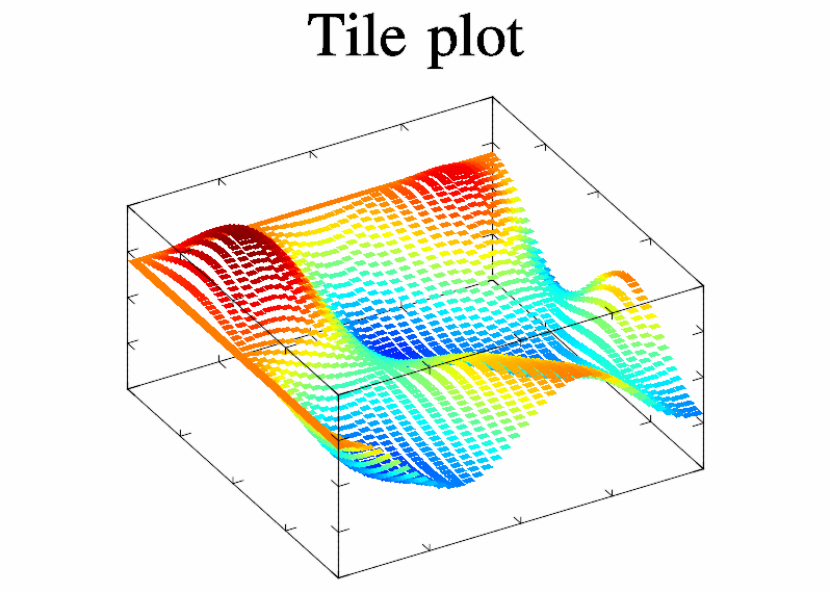
```
call 'prepare2d'
origin 0 0 0
subplot 2 2 0:title 'Boxs plot (default)':rotate 40 60:light on:box
boxs a

subplot 2 2 1:title '"\@" style':rotate 50 60:box
boxs a '@'

subplot 2 2 2:title '"\#" style':rotate 50 60:box
boxs a '#'

subplot 2 2 3:title 'compare with Tile':rotate 50 60:box
tile a
```



5.6.9 Tile sample

Command [tile], page 40, draw surface by tiles. The sample code is:

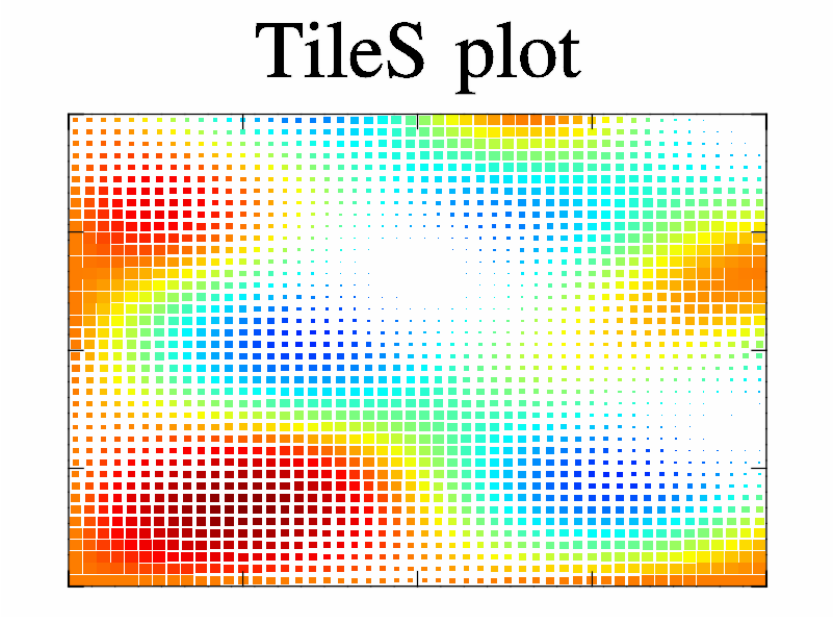
```
call 'prepare2d'
subplot 1 1 0 '':title 'Tiles plot':box
tile a
```



5.6.10 TileS sample

Command [tiles], page 46, is similar to [tile], page 40, but tile sizes is determined by another data. This allows one to simulate transparency of the plot. The sample code is:

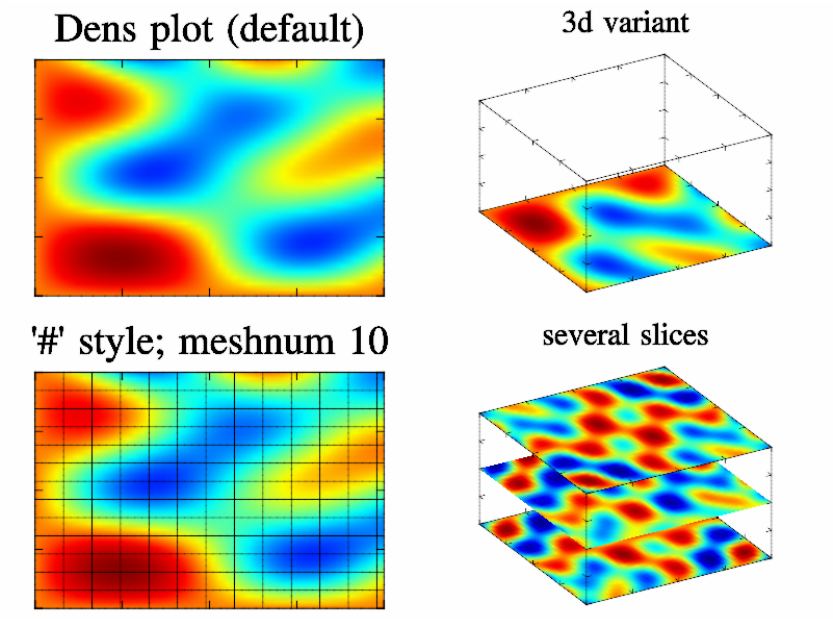
```
call 'prepare2d'
subplot 1 1 0 '':title 'Tiles plot':box
tiles a b
```



5.6.11 Dens sample

Command [dens], page 41, draw density plot for surface. The sample code is:

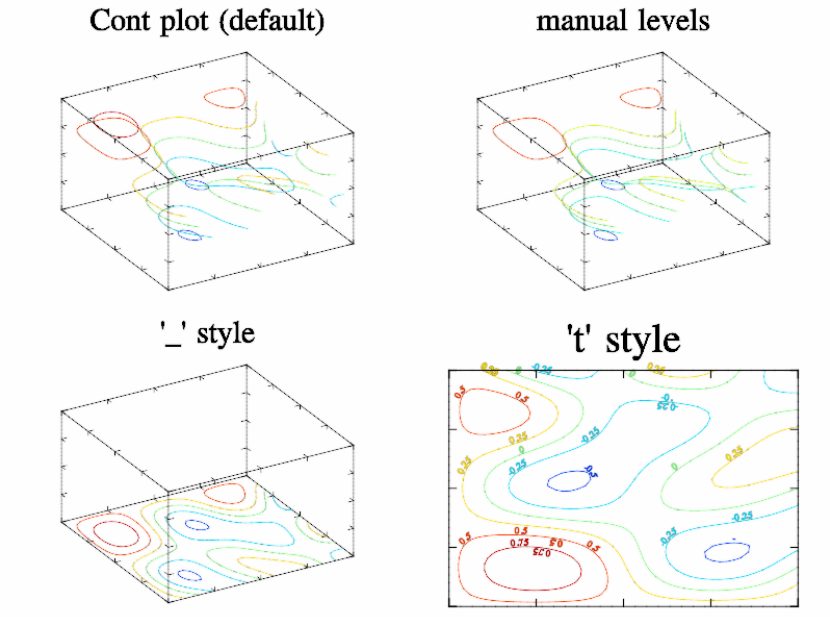
```
call 'prepare2d'
subplot 2 2 0 '':title 'Dens plot (default)':box
dens a

subplot 2 2 1:title '3d variant':rotate 50 60:box
dens a

subplot 2 2 2 '':title '"\#" style; meshnum 10':box
dens a '#'; meshnum 10

new a1 30 40 3 '0.6*sin(2*pi*x+pi*(z+1)/2)*sin(3*pi*y+pi*z) +\
 0.4*cos(3*pi*(x*y)+pi*(z+1)^2/2)'
subplot 2 2 3:title 'several slices':rotate 50 60:box
dens a1
```



5.6.12 Cont sample

Command [cont], page 41, draw contour lines for surface. You can select automatic (default) or manual levels for contours, print contour labels, draw it on the surface (default) or at plane (as `Dens`). The sample code is:

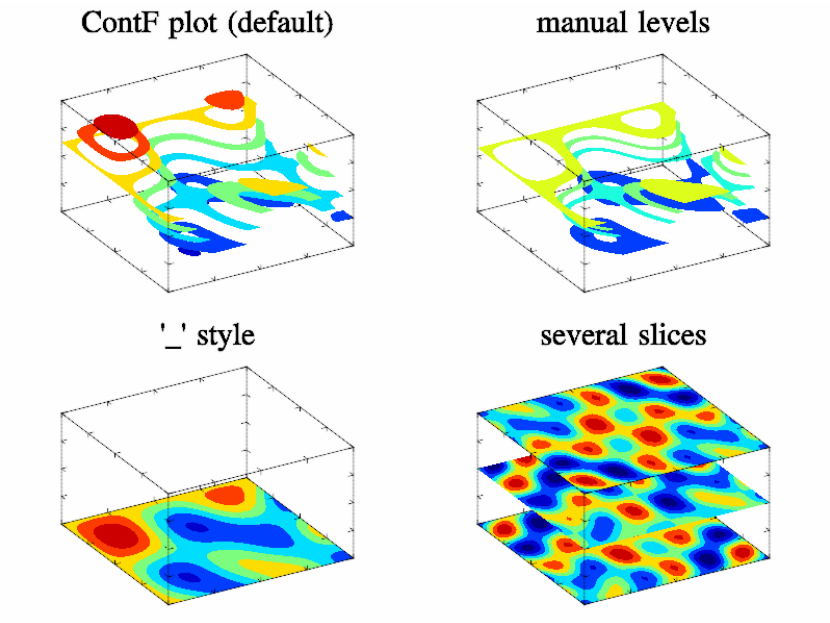
```
call 'prepare2d'
list v -0.5 -0.15 0 0.15 0.5
subplot 2 2 0:title 'Cont plot (default)':rotate 50 60:box
cont a

subplot 2 2 1:title 'manual levels':rotate 50 60:box
cont v a

subplot 2 2 2:title '"\_" style':rotate 50 60:box
cont a '_'

subplot 2 2 3 '':title '"t" style':box
cont a 't'
```



5.6.13 ContF sample

Command [contf], page 41, draw filled contours. You can select automatic (default) or manual levels for contours. The sample code is:

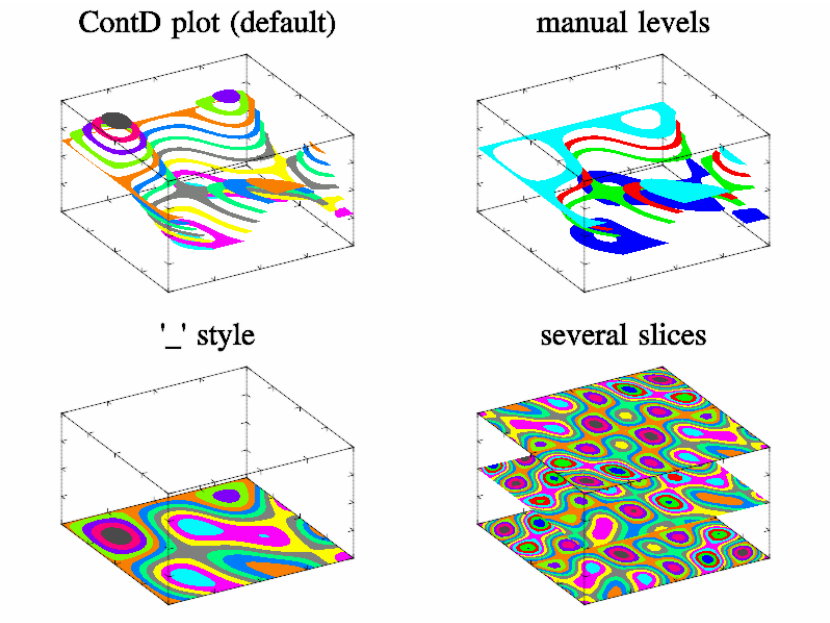
```
call 'prepare2d'
list v -0.5 -0.15 0 0.15 0.5
subplot 2 2 0:title 'ContF plot (default)':rotate 50 60:box
contf a

subplot 2 2 1:title 'manual levels':rotate 50 60:box
contf v a

subplot 2 2 2:title '"\_" style':rotate 50 60:box
contf a '_'

new a1 30 40 3 '0.6*sin(2*pi*x+pi*(z+1)/2)*sin(3*pi*y+pi*z) +\
 0.4*cos(3*pi*(x*y)+pi*(z+1)^2/2)'
subplot 2 2 3:title 'several slices':rotate 50 60:box
contf a1
```



5.6.14 ContD sample

Command [contd], page 41, is similar to `ContF` but with manual contour colors. The sample code is:

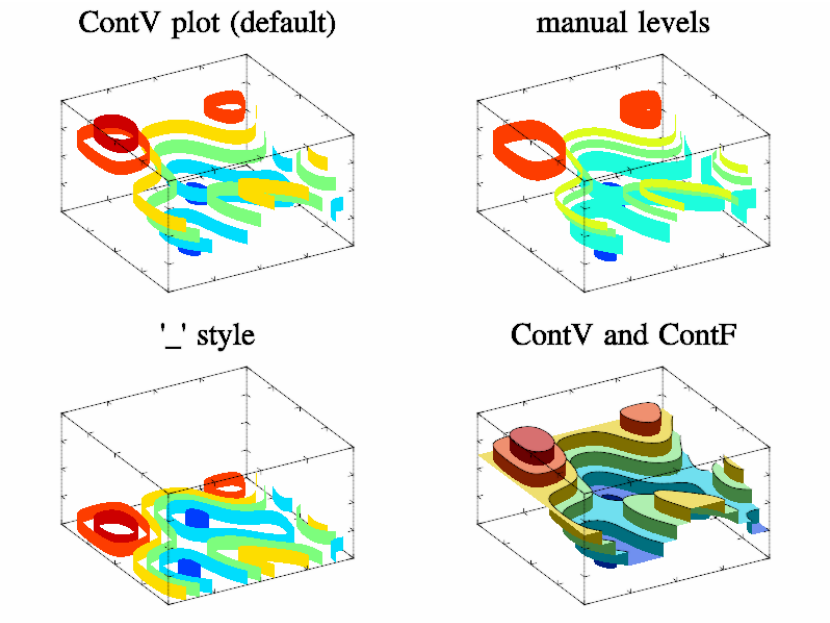
```
call 'prepare2d'
list v -0.5 -0.15 0 0.15 0.5
subplot 2 2 0:title 'ContD plot (default)':rotate 50 60:box
contd a

subplot 2 2 1:title 'manual levels':rotate 50 60:box
contd v a

subplot 2 2 2:title '"\_" style':rotate 50 60:box
contd a '_'

new a1 30 40 3 '0.6*sin(2*pi*x+pi*(z+1)/2)*sin(3*pi*y+pi*z) +\
 0.4*cos(3*pi*(x*y)+pi*(z+1)^2/2)'
subplot 2 2 3:title 'several slices':rotate 50 60:box
contd a1
```



5.6.15 ContV sample

Command [contv], page 42, draw vertical cylinders (belts) at contour lines. The sample code is:

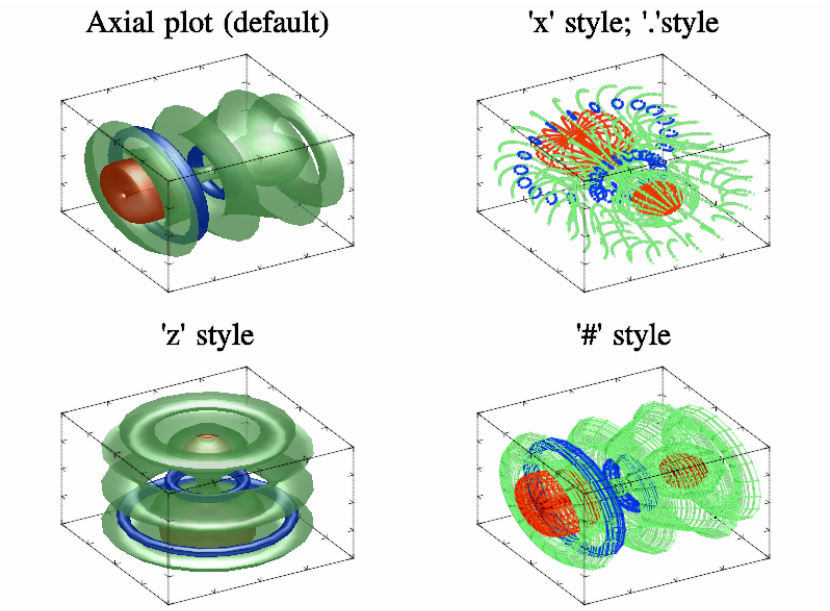
```
call 'prepare2d'
list v -0.5 -0.15 0 0.15 0.5
subplot 2 2 0:title 'ContV plot (default)':rotate 50 60:box
contv a

subplot 2 2 1:title 'manual levels':rotate 50 60:box
contv v a

subplot 2 2 2:title '"\_" style':rotate 50 60:box
contv a '_'

subplot 2 2 3:title 'ContV and ContF':rotate 50 60:light on:box
contv a:contf a:cont a 'k'
```



5.6.16 Axial sample

Command [axial], page 42, draw surfaces of rotation for contour lines. You can draw wire surfaces (‘#’ style) or ones rotated in other directions (‘x’, ‘z’ styles). The sample code is:

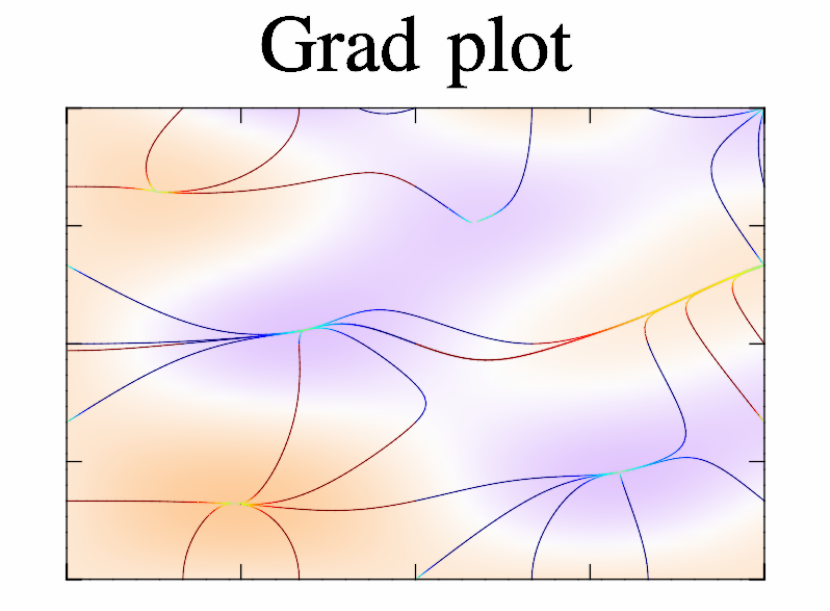
```
light on:alpha on:call 'prepare2d'
subplot 2 2 0:title 'Axial plot (default)':rotate 50 60:box
axial a

subplot 2 2 1:title '"x" style;\".\" style':light on:rotate 50 60:box
axial a 'x.'

subplot 2 2 2:title '"z" style':light on:rotate 50 60:box
axial a 'z'

subplot 2 2 3:title '"\#" style':light on:rotate 50 60:box
axial a '#'
```



5.6.17 Grad sample

Command [grad], page 48, draw gradient lines for matrix. The sample code is:

```
call 'prepare2d'
subplot 1 1 0 '':title 'Grad plot':box
grad a:dens a '{u8}w{q8}'
```



5.7 3D samples

This section is devoted to visualization of 3D data arrays. 3D means the data which depend on 3 indexes (parameters) like tensor a(i,j,k)=a(x(i),y(j),x(k)), i=1...n, j=1...m, k=1...l or in parametric form {x(i,j,k),y(i,j,k),z(i,j,k),a(i,j,k)}. Most of samples will use the same data for plotting. So, I put its initialization in separate function

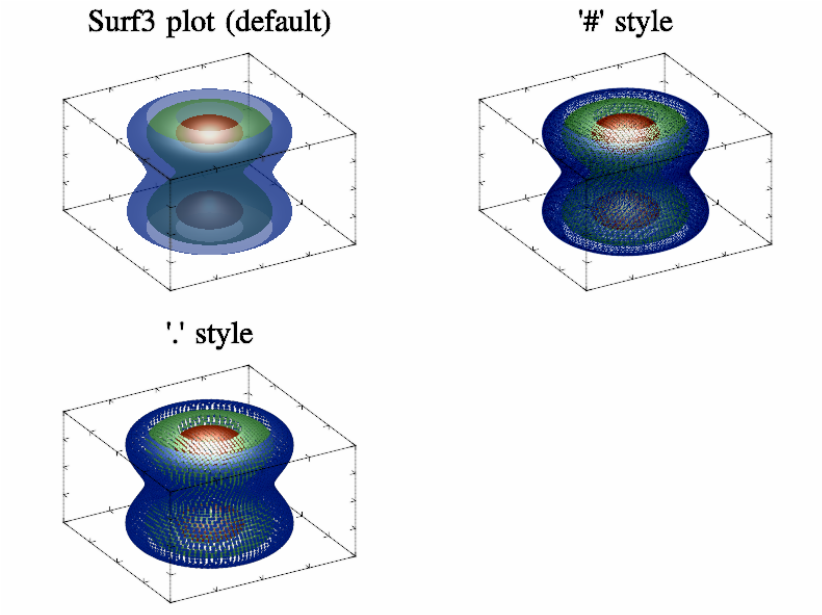
```
func 'prepare3d'
new c 61 50 40 '-2*(x^2+y^2+z^4-z^2)+0.2'
new d 61 50 40 '1-2*tanh((x+y)*(x+y))'
return
```

Basically, you can put this text after the script. Note, that you need to terminate main script by [stop], page 4, command before defining a function.

5.7.1 Surf3 sample

Command [surf3], page 42, is one of most suitable (for my opinion) functions to visualize 3D data. It draw the isosurface(s) – surface(s) of constant amplitude (3D analogue of contour lines). You can draw wired isosurfaces if specify '#' style. The sample code is:

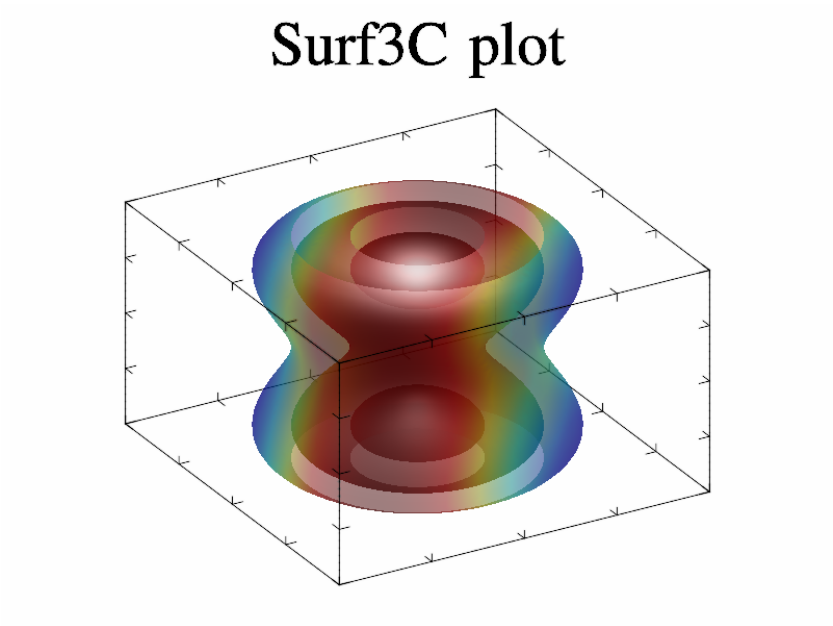
```
call 'prepare3d'
light on:alpha on
subplot 2 2 0:title 'Surf3 plot':rotate 50 60:box
surf3 c

subplot 2 2 1:title '"\#" style':rotate 50 60:box
surf3 c '#'

subplot 2 2 2:title '"." style':rotate 50 60:box
surf3 c '.'
```



5.7.2 Surf3C sample

Command [surf3c], page 45, is similar to [surf3], page 42, but its coloring is determined by another data. The sample code is:

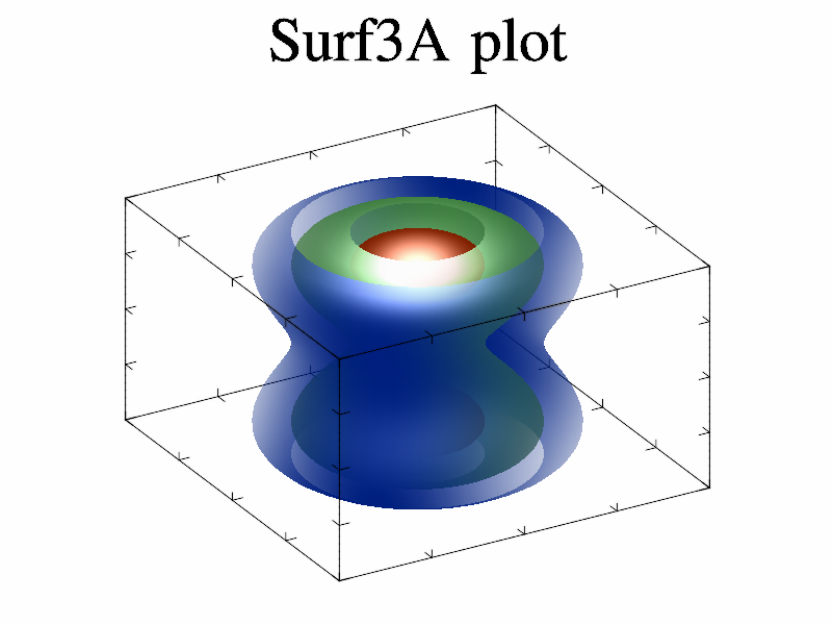
```
call 'prepare3d'
title 'Surf3C plot':rotate 50 60:light on:alpha on:box
surf3c c d
```



5.7.3 Surf3A sample

Command [surf3a], page 45, is similar to [surf3], page 42, but its transparency is determined by another data. The sample code is:

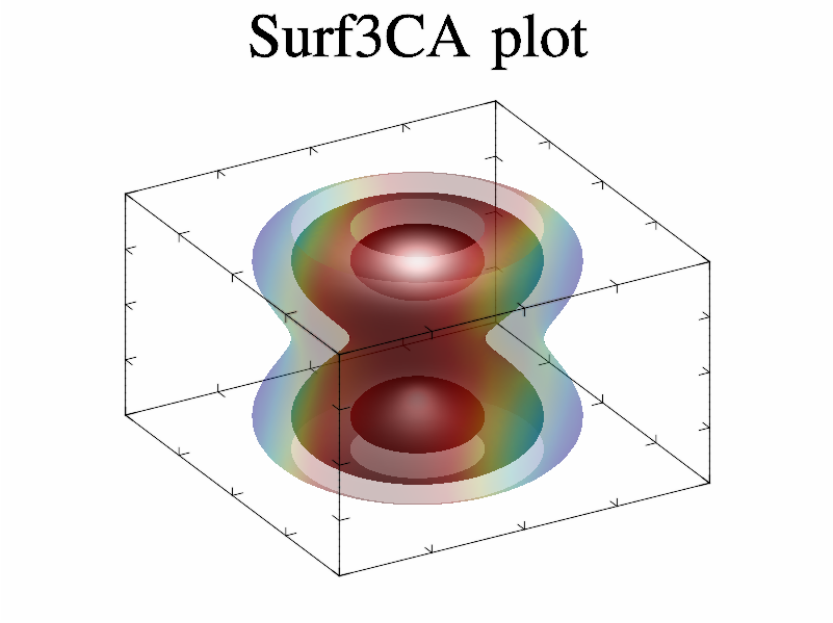
```
call 'prepare3d'
title 'Surf3A plot':rotate 50 60:light on:alpha on:box
surf3a c d
```



5.7.4 Surf3CA sample

Command [surf3ca], page 46, is similar to [surf3], page 42, but its color and transparency is determined by another data. The sample code is:

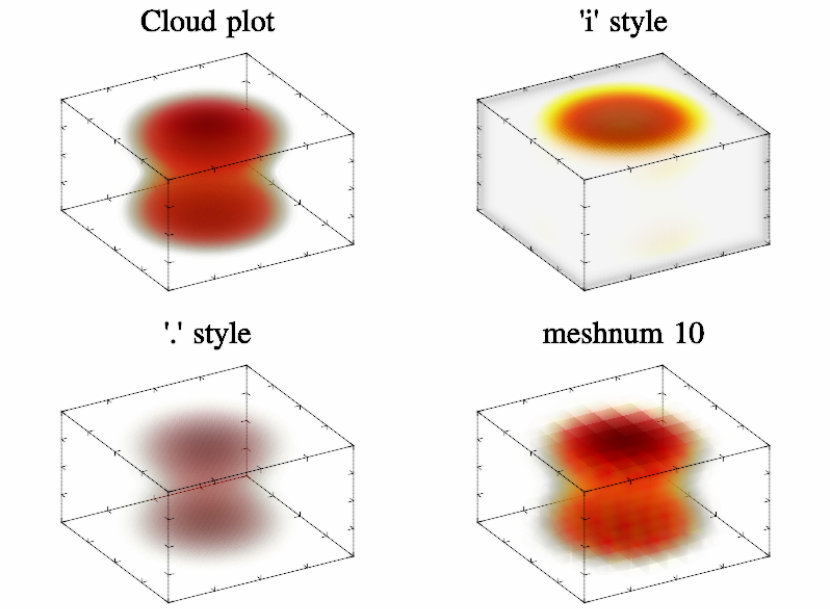
```
call 'prepare3d'
title 'Surf3CA plot':rotate 50 60:light on:alpha on:box
surf3a c d c
```



5.7.5 Cloud sample

Command [cloud], page 43, draw cloud-like object which is less transparent for higher data values. Similar plot can be created using many (about 10-20) `Surf3A(a,a)` isosurfaces. The sample code is:

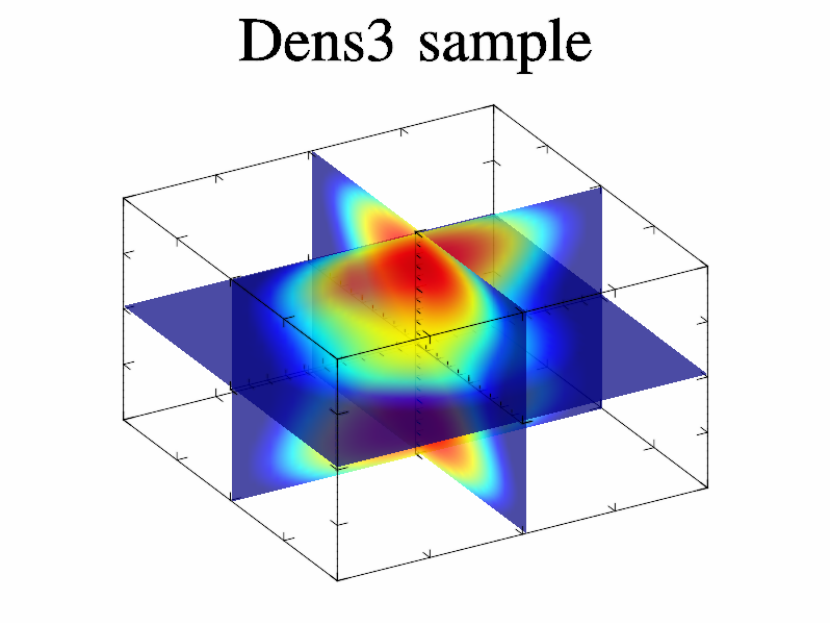
```
call 'prepare3d'
subplot 2 2 0:title 'Cloud plot':rotate 50 60:alpha on:box
cloud c 'wyrRk'

subplot 2 2 1:title '"i" style':rotate 50 60:box
cloud c 'iwyrRk'

subplot 2 2 2:title '"." style':rotate 50 60:box
cloud c '.wyrRk'

subplot 2 2 3:title 'meshnum 10':rotate 50 60:box
cloud c 'wyrRk'; meshnum 10
```



5.7.6 Dens3 sample

Command [dens3], page 43, draw just usual density plot but at slices of 3D data. The sample code is:

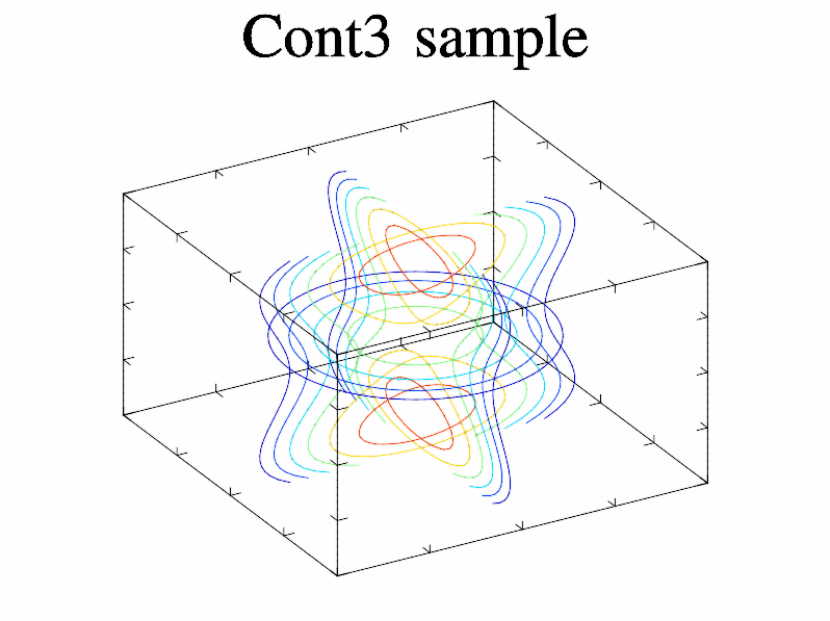
```
call 'prepare3d'
title 'Dens3 sample':rotate 50 60:alpha on:alphadef 0.7
origin 0 0 0:box:axis '_xyz'
dens3 c 'x':dens3 c ':y':dens3 c 'z'
```



5.7.7 Cont3 sample

Command [cont3], page 43, draw just usual contour lines but at slices of 3D data. The sample code is:

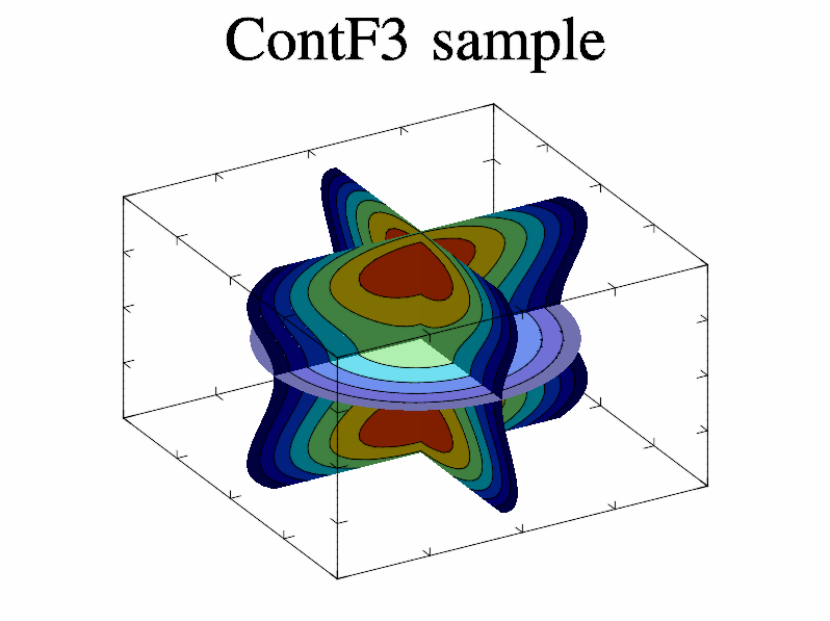
```
call 'prepare3d'
title 'Cont3 sample':rotate 50 60:box
cont3 c 'x':cont3 c:cont3 c 'z'
```



5.7.8 ContF3 sample

Command [contf3], page 44, draw just usual filled contours but at slices of 3D data. The sample code is:

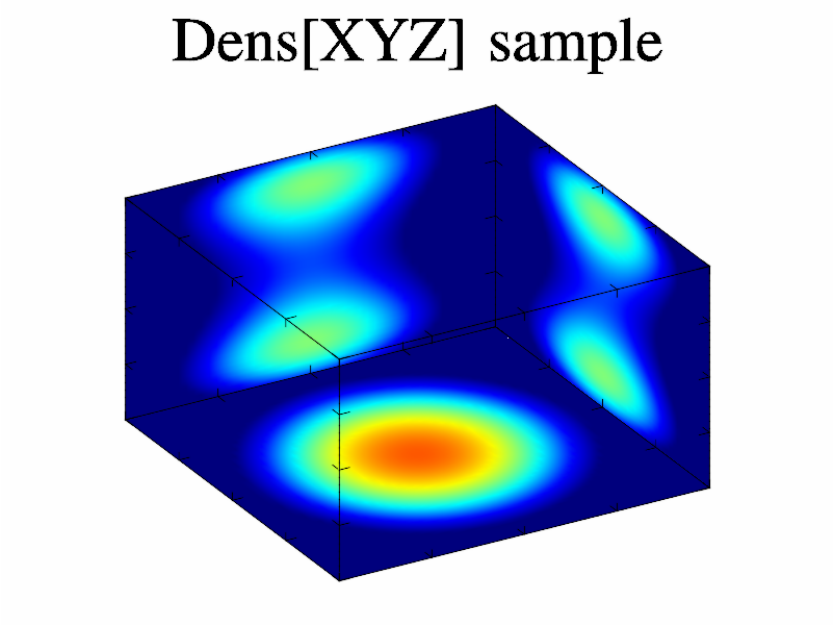
```
call 'prepare3d'
title 'Cont3 sample':rotate 50 60:box:light on
contf3 c 'x':contf3 c:contf3 c 'z'
cont3 c 'xk':cont3 c 'k':cont3 c 'zk'
```



5.7.9 Dens projection sample

Functions [densz], page 49, [densy], page 49, [densx], page 49, draw density plot on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

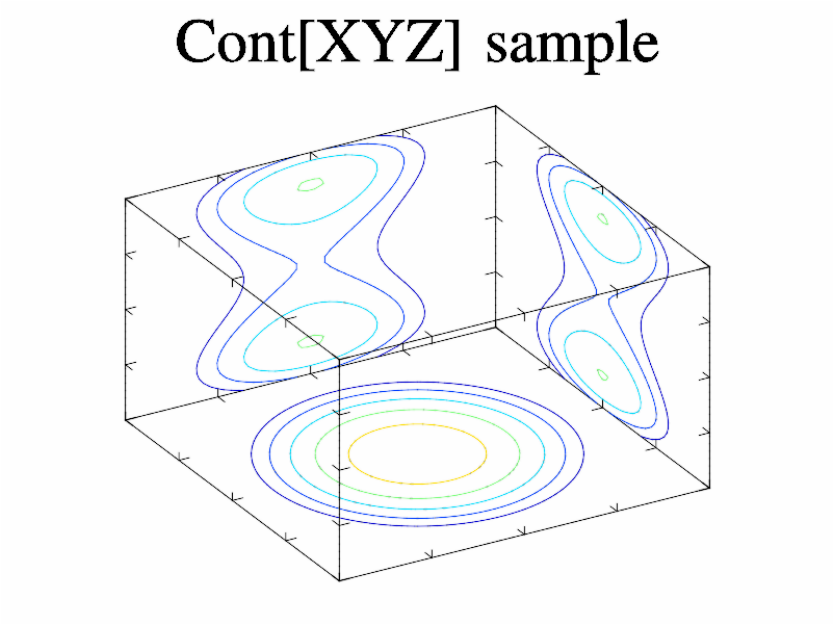
```
call 'prepare3d'
title 'Dens[XYZ] sample':rotate 50 60:box
densx {sum c 'x'} '' -1
densy {sum c 'y'} '' 1
densz {sum c 'z'} '' -1
```



5.7.10 Cont projection sample

Functions [contz], page 49, [conty], page 49, [contx], page 49, draw contour lines on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

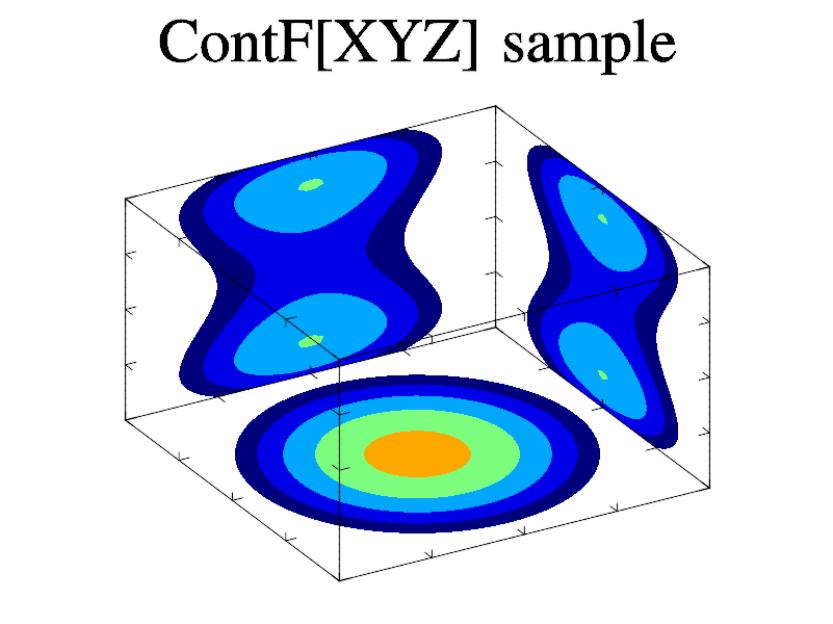
```
call 'prepare3d'
title 'Cont[XYZ] sample':rotate 50 60:box
contx {sum c 'x'} '' -1
conty {sum c 'y'} '' 1
contz {sum c 'z'} '' -1
```



5.7.11 ContF projection sample

Functions [contfz], page 49, [contfy], page 49, [contfx], page 49, draw filled contours on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

```
call 'prepare3d'
title 'ContF[XYZ] sample':rotate 50 60:box
contfx {sum c 'x'} '' -1
contfy {sum c 'y'} '' 1
contfz {sum c 'z'} '' -1
```

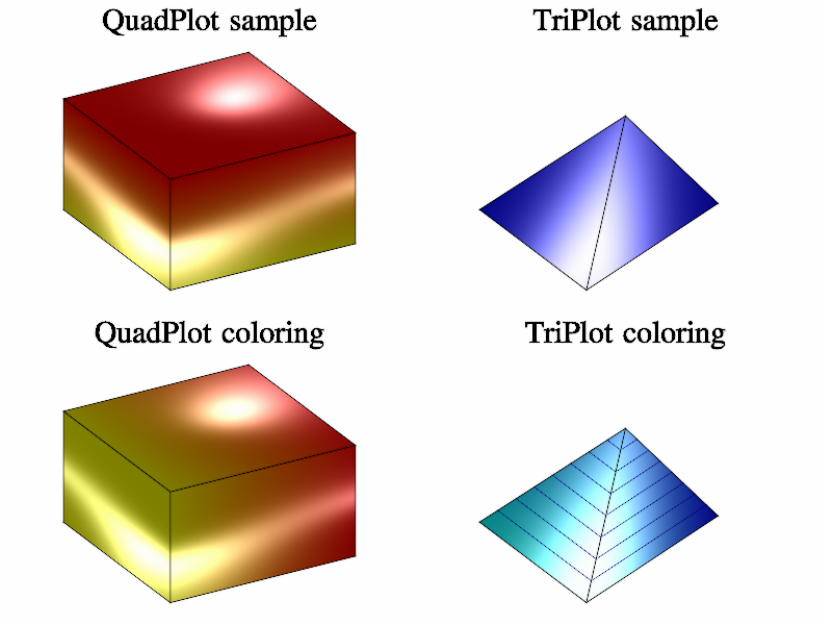


5.7.12 TriPlot and QuadPlot

Command [triplot], page 50, and [quadplot], page 51, draw set of triangles (or quadrangles for `QuadPlot`) for irregular data arrays. Note, that you have to provide not only vertexes, but also the indexes of triangles or quadrangles. I.e. perform triangulation by some other library. The sample code is:

```
list q 0 1 2 3 | 4 5 6 7 | 0 2 4 6 | 1 3 5 7 | 0 4 1 5 | 2 6 3 7
list xq -1 1 -1 1 -1 1 -1 1
list yq -1 -1 1 1 -1 -1 1 1
list zq -1 -1 -1 -1 1 1 1 1
light on
subplot 2 2 0:title 'QuadPlot sample':rotate 50 60
quadplot q xq yq zq 'yr'
quadplot q xq yq zq '#k'

subplot 2 2 2:title 'QuadPlot coloring':rotate 50 60
quadplot q xq yq zq yq 'yr'
quadplot q xq yq zq '#k'
```

```
list t 0 1 2 | 0 1 3 | 0 2 3 | 1 2 3
list xt -1 1 0 0
list yt -1 -1 1 0
list zt -1 -1 -1 1
subplot 2 2 1:title 'TriPlot sample':rotate 50 60
triplot t xt yt zt 'b'
triplot t xt yt zt '#k'

subplot 2 2 3:title 'TriPlot coloring':rotate 50 60
triplot t xt yt zt yt 'cb'
triplot t xt yt zt '#k'
tricont t xt yt zt 'B'
```

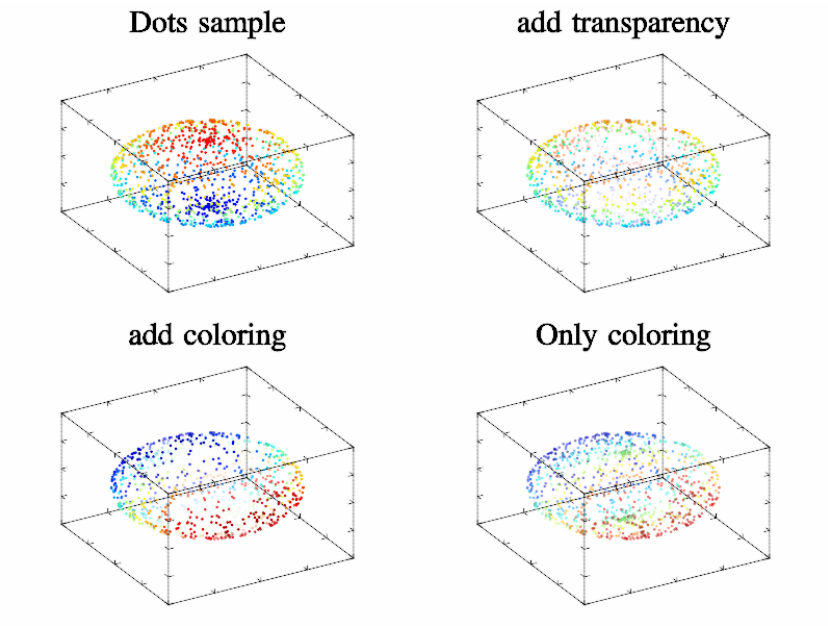


5.7.13 Dots sample

Command [dots], page 51, is another way to draw irregular points. `Dots` use color scheme for coloring (see Section 2.4 [Color scheme], page 11). The sample code is:

```
new t 2000 'pi*(rnd-0.5)':new f 2000 '2*pi*rnd'
copy x 0.9*cos(t)*cos(f):copy y 0.9*cos(t)*sin(f):copy z 0.6*sin(t):copy c cos(2*t)
subplot 2 2 0:title 'Dots sample':rotate 50 60
box:dots x y z
alpha on
subplot 2 2 1:title 'add transparency':rotate 50 60
box:dots x y z c
subplot 2 2 2:title 'add colorings':rotate 50 60
box:dots x y z x c
subplot 2 2 3:title 'Only coloring':rotate 50 60
```

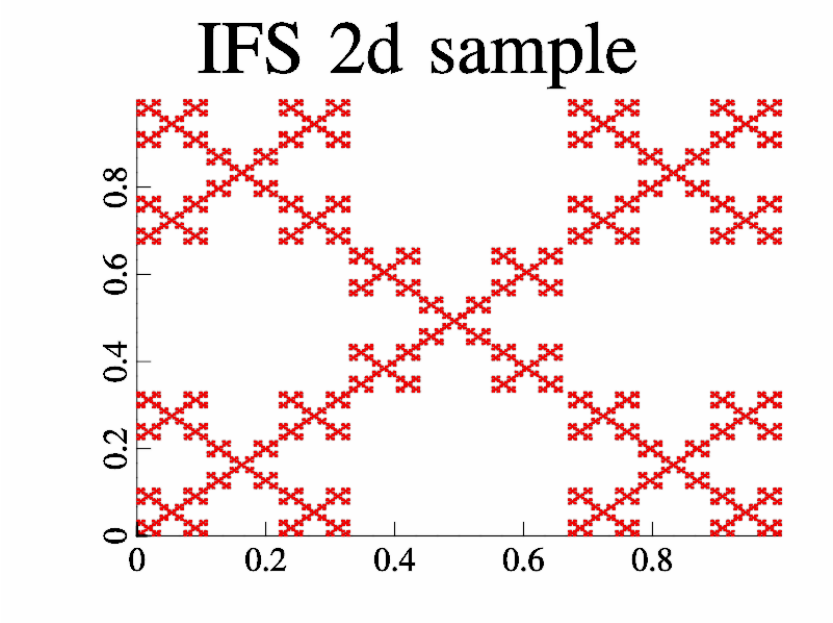
```
box:tens x y z x ' .'
```



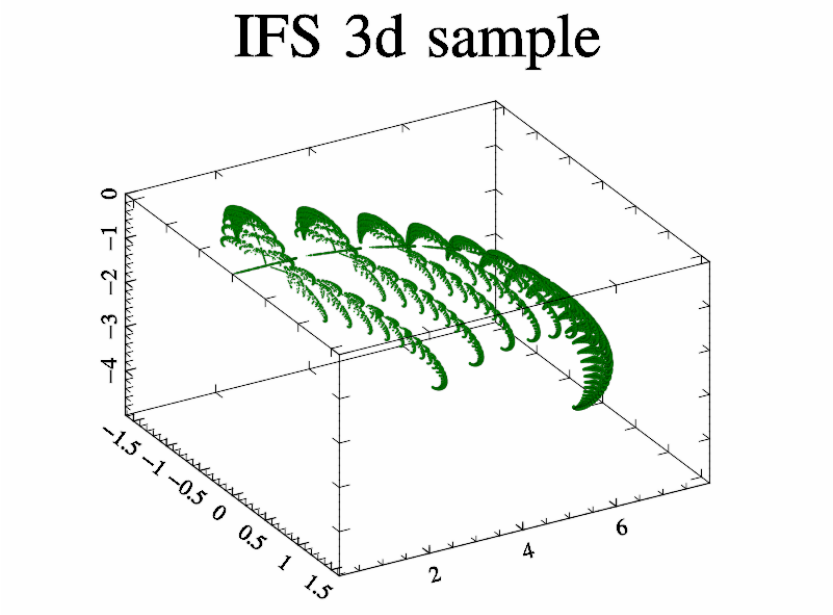
5.7.14 IFS sample

Commands [ifs2d], page 67, and [ifs3d], page 67, generate points for fractals using iterated function system in 2d and 3d cases correspondingly. The sample codes are:

```
list A [0.33,0,0,0.33,0,0,0.2] [0.33,0,0,0.33,0.67,0,0.2] [0.33,0,0,0.33,0.33,0.33,0.2]\
       [0.33,0,0,0.33,0,0.67,0.2] [0.33,0,0,0.33,0.67,0.67,0.2]
ifs2d fx fy A 100000
subplot 1 1 0 '<_':title 'IFS 2d sample'
ranges fx fy:axis
plot fx fy 'r#o ';size 0.05
```




```
list A [0,0,0,0,.18,0,0,0,0,0,0,0,.01] [.85,0,0,0,.85,.1,0,-0.1,0.85,0,1.6,0,.85]\
        [.2,-.2,0,.2,.2,0,0,0,0.3,0,0.8,0,.07] [-.2,.2,0,.2,.2,0,0,0,0.3,0,0.8,0,.07]
ifs3d f A 100000
title 'IFS 3d sample':rotate 50 60
ranges f(0) f(1) f(2):axis:box
dots f(0) f(1) f(2) 'G#o';size 0.05
```



5.8 Vector field samples

Vector field visualization (especially in 3d case) is more or less complex task. MathGL provides 3 general types of plots: vector field itself (`Vect`), flow threads (`Flow`), and flow pipes with radius proportional to field amplitude (`Pipe`).

However, the plot may look tangly – there are too many overlapping lines. I may suggest 2 ways to solve this problem. The first one is to change `SetMeshNum` for decreasing the number of hachures. The second way is to use the flow thread chart `Flow`, or possible many flow thread from manual position (`FlowP`). Unfortunately, I don't know any other methods to visualize 3d vector field. If you know any, e-mail me and I shall add it to MathGL.

Most of samples will use the same data for plotting. So, I put its initialization in separate function

```
func 'prepare2v'
new a 20 30 '0.6*sin(pi*(x+1))*sin(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
new b 20 30 '0.6*cos(pi*(x+1))*cos(1.5*pi*(y+1))+0.4*cos(0.75*pi*(x+1)*(y+1))'
return

func 'prepare3v'
define $1 pow(x*x+y*y+(z-0.3)*(z-0.3)+0.03,1.5)
define $2 pow(x*x+y*y+(z+0.3)*(z+0.3)+0.03,1.5)
new ex 10 10 10 '0.2*x/$1-0.2*x/$2'
new ey 10 10 10 '0.2*y/$1-0.2*y/$2'
new ez 10 10 10 '0.2*(z-0.3)/$1-0.2*(z+0.3)/$2'
return
```

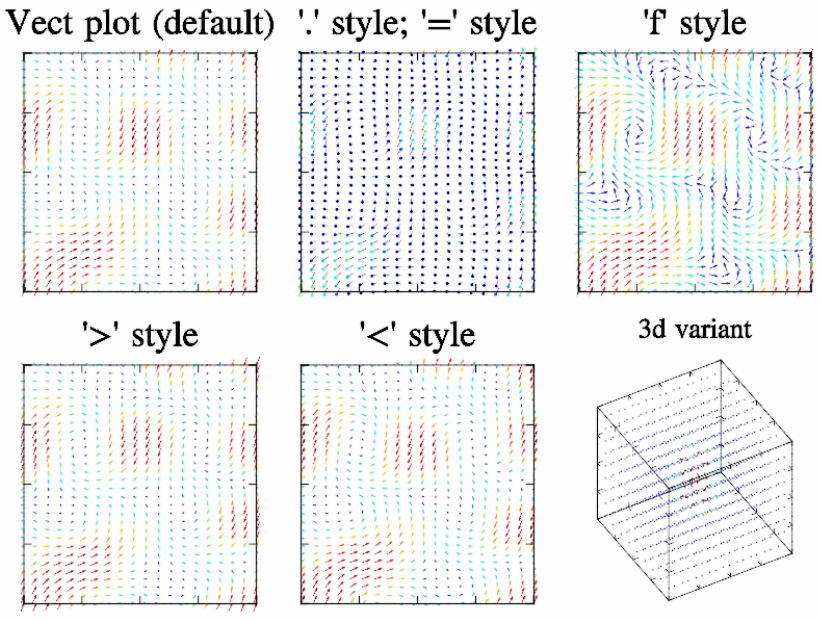
Basically, you can put this text after the script. Note, that you need to terminate main script by [stop], page 4, command before defining a function.

5.8.1 Vect sample

Command [vect], page 47, is most standard way to visualize vector fields – it draw a lot of arrows or hachures for each data cell. It have a lot of options which can be seen on the figure (and in the sample code). `Vect` use color scheme for coloring (see Section 2.4 [Color scheme], page 11). The sample code is:

```
call 'prepare2v'
subplot 3 2 0 '':title 'Vect plot (default)':box
vect a b

subplot 3 2 1 '':title '"." style; "=" style':box
vect a b '.='

subplot 3 2 2 '':title '"f" style':box
vect a b 'f'

subplot 3 2 3 '':title '">" style':box
vect a b '>'

subplot 3 2 4 '':title '"<" style':box
```

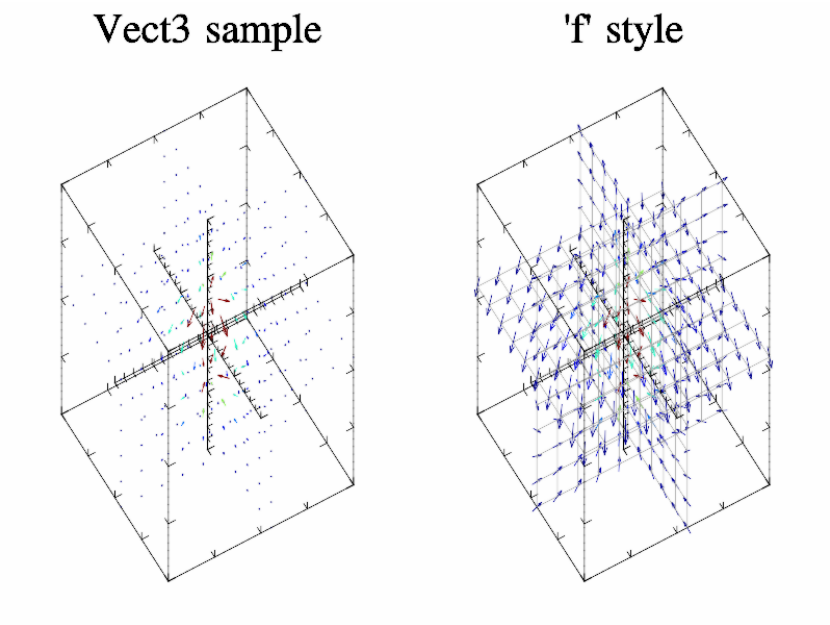
```
vect a b '<'

call 'prepare3v'
subplot 3 2 5:title '3d variant':rotate 50 60:box
vect ex ey ez
```



5.8.2 Vect3 sample

Command [vect3], page 47, draw just usual vector field plot but at slices of 3D data. The sample code is:

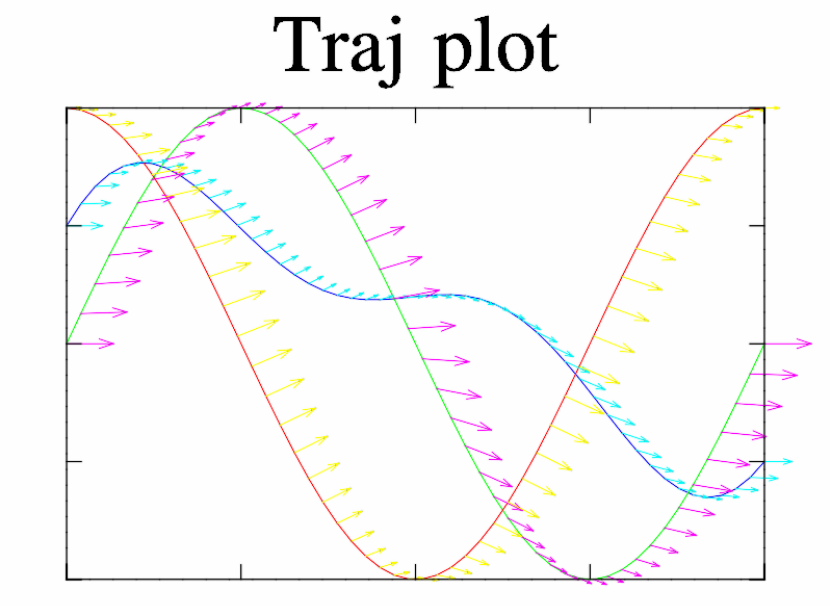
```
origin 0 0 0:call 'prepare3v'
subplot 2 1 0:title 'Vect3 sample':rotate 50 60
box:axis '_xyz'
vect3 ex ey ez 'x':vect3 ex ey ez:vect3 ex ey ez 'z'

subplot 2 1 1:title '"f" style':rotate 50 60
box:axis '_xyz'
vect3 ex ey ez 'fx':vect3 ex ey ez 'f':vect3 ex ey ez 'fz'
grid3 ex 'Wx':grid3 ex 'W':grid3 ex 'Wz'
```



5.8.3 Traj sample

Command [traj], page 47, is 1D analogue of `Vect`. It draw vectors from specified points. The sample code is:

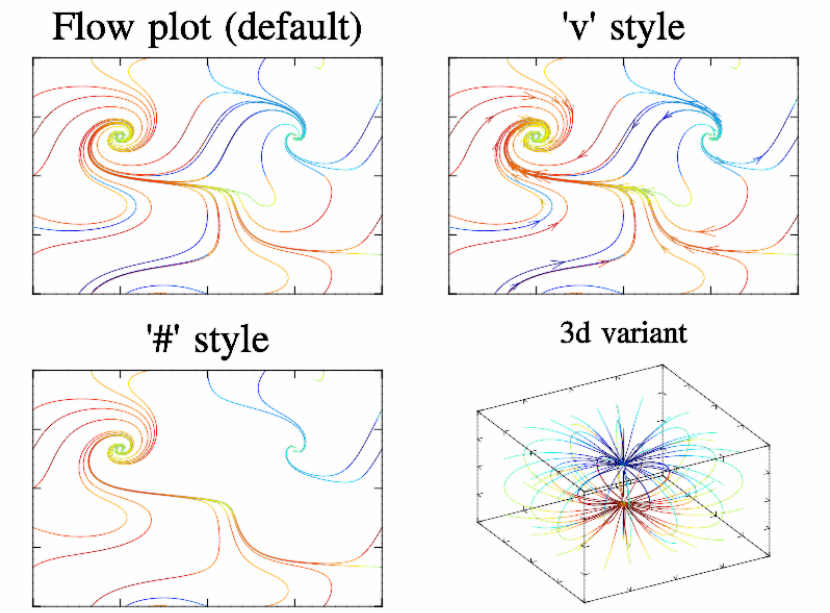
```
call 'prepare1d'
subplot 1 1 0 '':title 'Traj plot':box
plot x1 y:traj x1 y y1 y2
```



5.8.4 Flow sample

Command [flow], page 48, is another standard way to visualize vector fields – it draw lines (threads) which is tangent to local vector field direction. MathGL draw threads from edges of bounding box and from central slices. Sometimes it is not most appropriate variant – you may want to use `FlowP` to specify manual position of threads. `Flow` use color scheme for coloring (see Section 2.4 [Color scheme], page 11). At this warm color corresponds to normal flow (like attractor), cold one corresponds to inverse flow (like source). The sample code is:

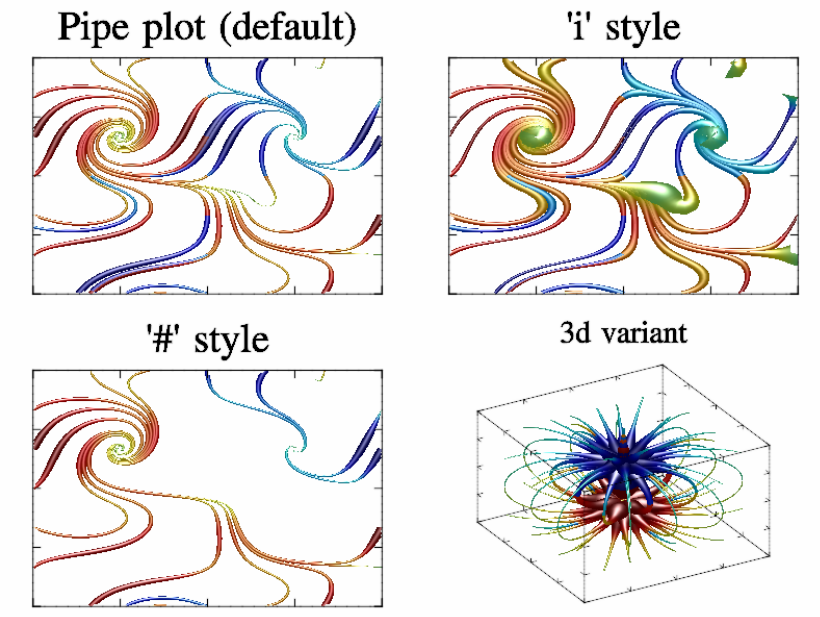
```
call 'prepare2v'
subplot 2 2 0 '':title 'Flow plot (default)':box
flow a b

subplot 2 2 1 '':title '"v" style':box
flow a b 'v'

subplot 2 2 2 '':title 'from edges only':box
flow a b '#'

call 'prepare3v'
subplot 2 2 3:title '3d variant':rotate 50 60:box
flow ex ey ez
```



5.8.5 Pipe sample

Command [pipe], page 49, is similar to [flow], page 48, but draw pipes (tubes) which radius is proportional to the amplitude of vector field. `Pipe` use color scheme for coloring (see Section 2.4 [Color scheme], page 11). At this warm color corresponds to normal flow (like attractor), cold one corresponds to inverse flow (like source). The sample code is:

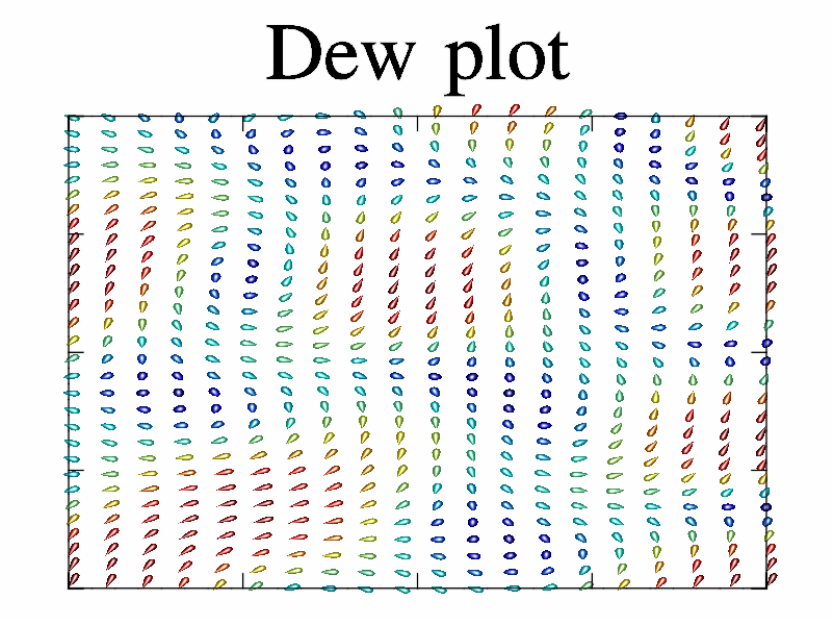
```
call 'prepare2v'
subplot 2 2 0 '':title 'Pipe plot (default)':light on:box
pipe a b

subplot 2 2 1 '':title '"i" style':box
pipe a b 'i'

subplot 2 2 2 '':title 'from edges only':box
pipe a b '#'

call 'prepare3v'
subplot 2 2 3:title '3d variant':rotate 50 60:box
pipe ex ey ez '' 0.1
```



5.8.6 Dew sample

Command [dew], page 48, is similar to `Vect` but use drops instead of arrows. The sample code is:

```
call 'prepare2v'
subplot 1 1 0 '':title 'Dew plot':light on:box
dew a b
```



5.9 Hints

In this section I've included some small hints and advices for the improving of the quality of plots and for the demonstration of some non-trivial features of MathGL library. In contrast to previous examples I showed mostly the idea but not the whole drawing function.

5.9.1 "Compound" graphics

As I noted above, MathGL functions (except the special one, like Clf()) do not erase the previous plotting but just add the new one. It allows one to draw "compound" plots easily. For example, popular Matlab command `surfc` can be emulated in MathGL by 2 calls:

```
Surf(a);
Cont(a, "_");     // draw contours at bottom
```

Here *a* is 2-dimensional data for the plotting, `-1` is the value of z-coordinate at which the contour should be plotted (at the bottom in this example). Analogously, one can draw density plot instead of contour lines and so on.

Another nice plot is contour lines plotted directly on the surface:

```
Light(true);       // switch on light for the surface
Surf(a, "BbcyrR"); // select 'jet' colormap for the surface
Cont(a, "y");      // and yellow color for contours
```

The possible difficulties arise in black&white case, when the color of the surface can be close to the color of a contour line. In that case I may suggest the following code:

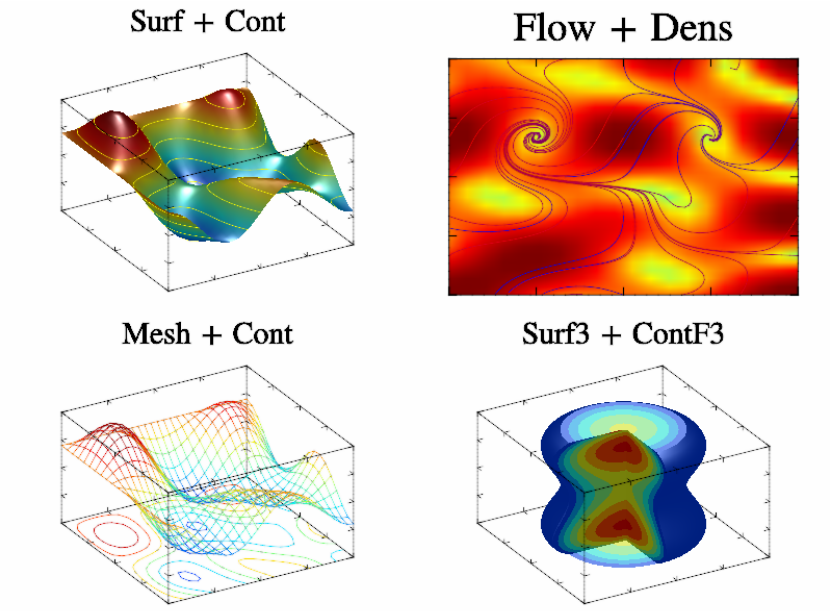
```
Light(true);   // switch on light for the surface
Surf(a, "kw"); // select 'gray' colormap for the surface
CAxis(-1,0);   // first draw for darker surface colors
Cont(a, "w");  // white contours
CAxis(0,1);    // now draw for brighter surface colors
Cont(a, "k");  // black contours
CAxis(-1,1);   // return color range to original state
```

The idea is to divide the color range on 2 parts (dark and bright) and to select the contrasting color for contour lines for each of part.

Similarly, one can plot flow thread over density plot of vector field amplitude (this is another amusing plot from Matlab) and so on. The list of compound graphics can be prolonged but I hope that the general idea is clear.

Just for illustration I put here following sample code:

```
call 'prepare2v'
call 'prepare3d'
new v 10:fill v -0.5 1:copy d sqrt(a^2+b^2)
subplot 2 2 0:title 'Surf + Cont':rotate 50 60:light on:box
surf a:cont a 'y'

subplot 2 2 1 '':title 'Flow + Dens':light off:box
flow a b 'br':dens d

subplot 2 2 2:title 'Mesh + Cont':rotate 50 60:box
mesh a:cont a '_'

subplot 2 2 3:title 'Surf3 + ContF3':rotate 50 60:light on
box:contf3 v c 'z' 0:contf3 v c 'x':contf3 v c
cut 0 -1 -1 1 0 1.1
contf3 v c 'z' c.nz-1:surf3 c -0.5
```

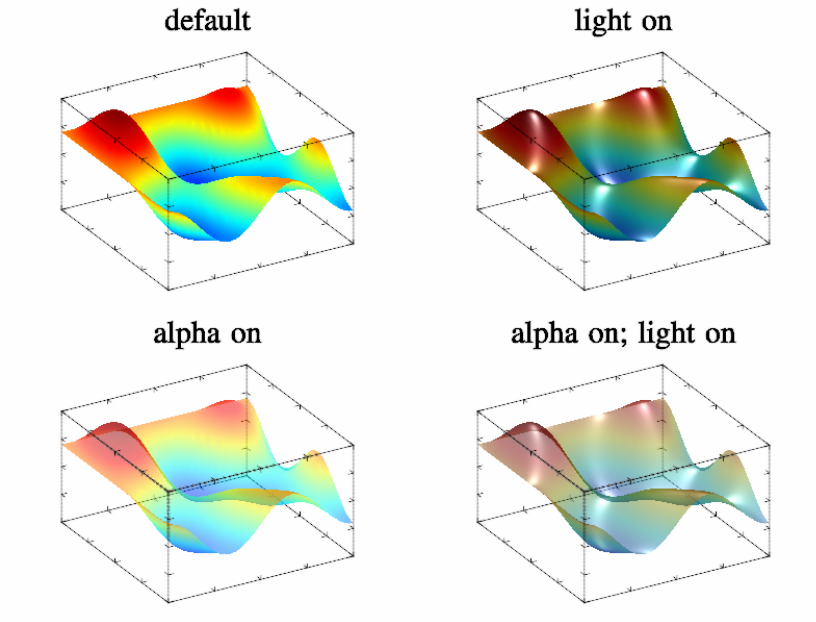


5.9.2 Transparency and lighting

Here I want to show how transparency and lighting both and separately change the look of a surface. So, there is code and picture for that:

```
call 'prepare2d'
```

```
subplot 2 2 0:title 'default':rotate 50 60:box
surf a

subplot 2 2 1:title 'light on':rotate 50 60:box
light on:surf a

subplot 2 2 3:title 'light on; alpha on':rotate 50 60:box
alpha on:surf a

subplot 2 2 2:title 'alpha on':rotate 50 60:box
light off:surf a
```



5.9.3 Types of transparency

MathGL library has advanced features for setting and handling the surface transparency. The simplest way to add transparency is the using of command [alpha], page 17. As a result, all further surfaces (and isosurfaces, density plots and so on) become transparent. However, their look can be additionally improved.

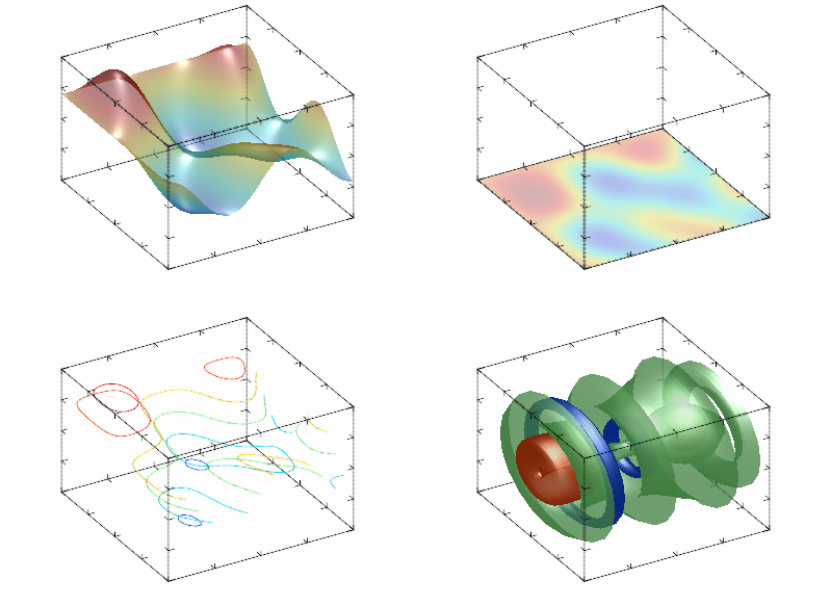
The value of transparency can be different from surface to surface. To do it just use `SetAlphaDef` before the drawing of the surface, or use option `alpha` (see Section 2.7 [Command options], page 15). If its value is close to 0 then the surface becomes more and more transparent. Contrary, if its value is close to 1 then the surface becomes practically non-transparent.

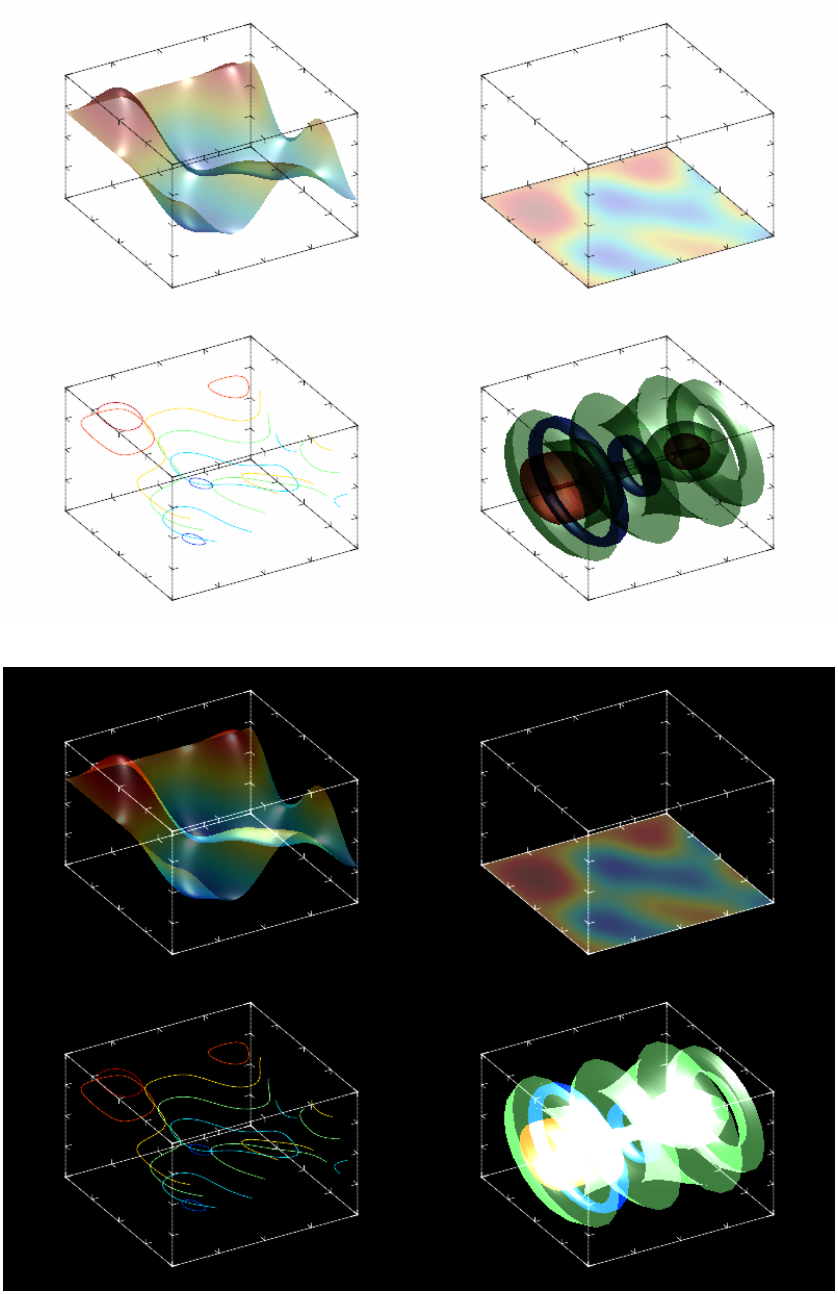
Also you can change the way how the light goes through overlapped surfaces. The function `SetTranspType` defines it. By default the usual transparency is used ('`0`') – surfaces below is less visible than the upper ones. A "glass-like" transparency ('`1`') has a different look – each surface just decreases the background light (the surfaces are commutable in this case).

A “neon-like” transparency (‘2’) has more interesting look. In this case a surface is the light source (like a lamp on the dark background) and just adds some intensity to the color. At this, the library sets automatically the black color for the background and changes the default line color to white.

As example I shall show several plots for different types of transparency. The code is the same except the values of `SetTranspType` function:

```
call 'prepare2d'
alpha on:light on
transptype 0:clf
subplot 2 2 0:rotate 50 60:surf a:box
subplot 2 2 1:rotate 50 60:dens a:box
subplot 2 2 2:rotate 50 60:cont a:box
subplot 2 2 3:rotate 50 60:axial a:box
```



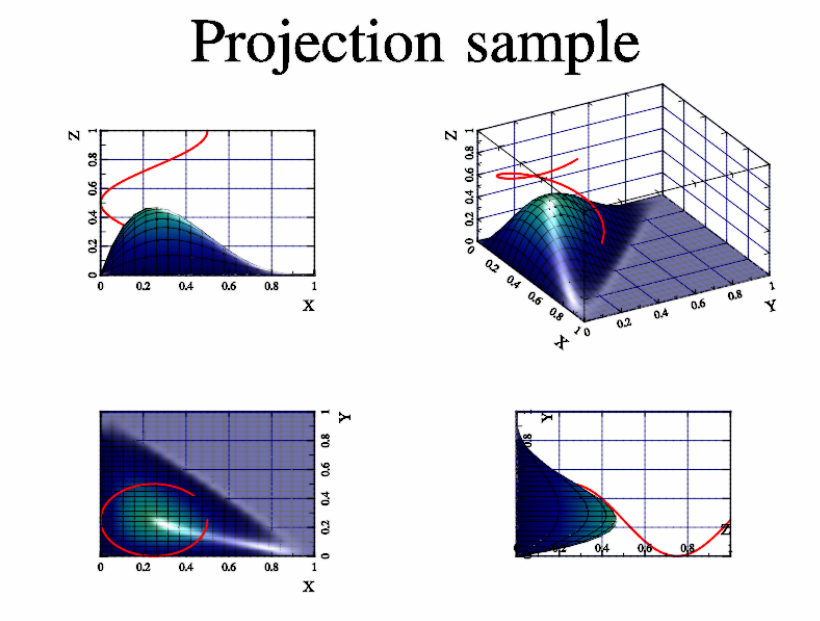


5.9.4 Axis projection

You can easily make 3D plot and draw its x-,y-,z-projections (like in CAD) by using [ternary], page 22, function with arguments: 4 for Cartesian, 5 for Ternary and 6 for Quaternary coordinates. The sample code is:

```
ranges 0 1 0 1 0 1
new x 50 '0.25*(1+cos(2*pi*x))'
new y 50 '0.25*(1+sin(2*pi*x))'
new z 50 'x'
new a 20 30 '30*x*y*(1-x-y)^2*(x+y<1)'
```

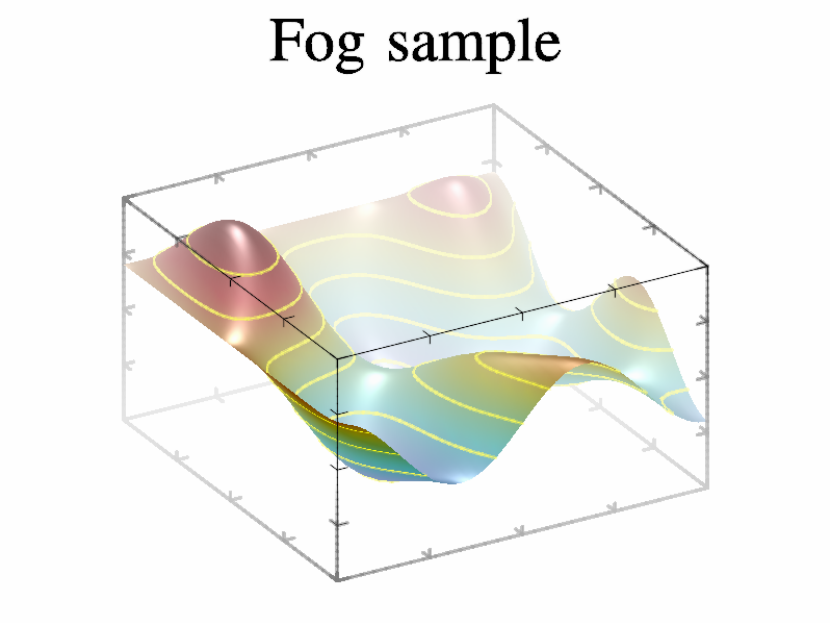
```
new rx 10 'rnd':new ry 10:fill ry '(1-v)*rnd' rx
light on

title 'Projection sample':ternary 4:rotate 50 60
box:axis:grid
plot x y z 'r2':surf a '#'
xlabel 'X':ylabel 'Y':zlabel 'Z'
```



5.9.5 Adding fog

MathGL can add a fog to the image. Its switching on is rather simple – just use [fog], page 18, function. There is the only feature – fog is applied for whole image. Not to particular subplot. The sample code is:

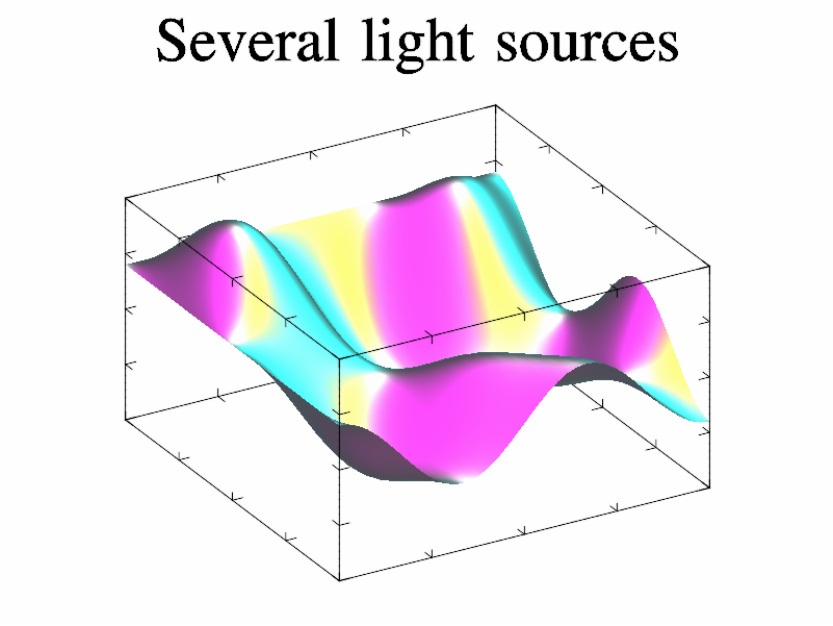
```
call 'prepare2d'
title 'Fog sample':rotate 50 60:light on
fog 1
box:surf a:cont a 'y'
```



5.9.6 Lighting sample

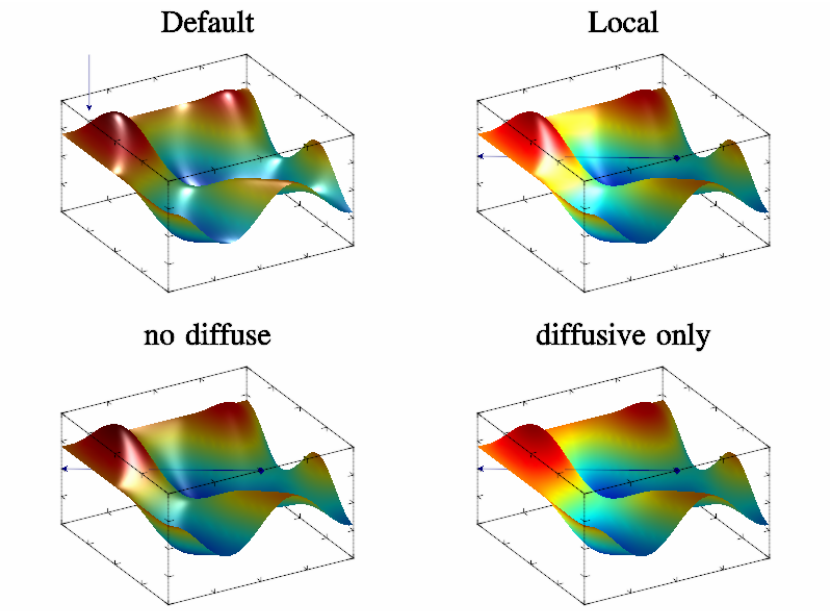
In contrast to the most of other programs, MathGL supports several (up to 10) light sources. Moreover, the color each of them can be different: white (this is usual), yellow, red, cyan, green and so on. The use of several light sources may be interesting for the highlighting of some peculiarities of the plot or just to make an amusing picture. Note, each light source can be switched on/off individually. The sample code is:

```
call 'prepare2d'
title 'Several light sources':rotate 50 60:light on
light 1 0 1 0 'c':light 2 1 0 0 'y':light 3 0 -1 0 'm'
box:surf a 'h'
```



Additionally, you can use local light sources and set to use [diffuse], page 18, reflection instead of specular one (by default) or both kinds. Note, I use [attachlight], page 18, command to keep light settings relative to subplot.

```
light on: attachlight on
call 'prepare2d'
subplot 2 2 0:title 'Default':rotate 50 60:box:surf a
line -1 -0.7 1.7 -1 -0.7 0.7 'BA'

subplot 2 2 1:title 'Local':rotate 50 60
light 0 1 0 1 -2 -1 -1
line 1 0 1 -1 -1 0 'BAO':box:surf a

subplot 2 2 2:title 'no diffuse':rotate 50 60
diffuse 0
line 1 0 1 -1 -1 0 'BAO':box:surf a

subplot 2 2 3:title 'diffusive only':rotate 50 60
diffuse 0.5:light 0 1 0 1 -2 -1 -1 'w' 0
line 1 0 1 -1 -1 0 'BAO':box:surf a
```

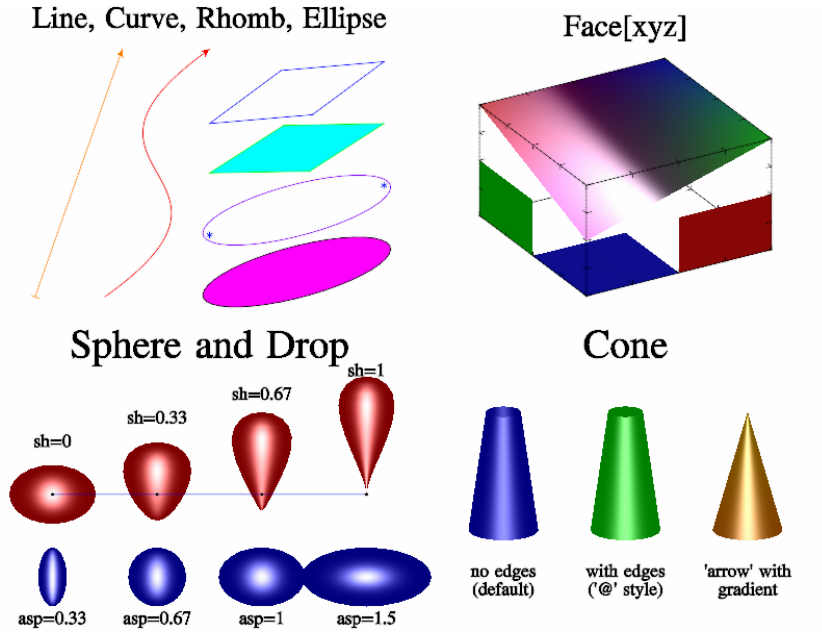


5.9.7 Using primitives

MathGL provide a set of functions for drawing primitives (see Section 3.7 [Primitives], page 27). Primitives are low level object, which used by most of plotting functions. Picture below demonstrate some of commonly used primitives.

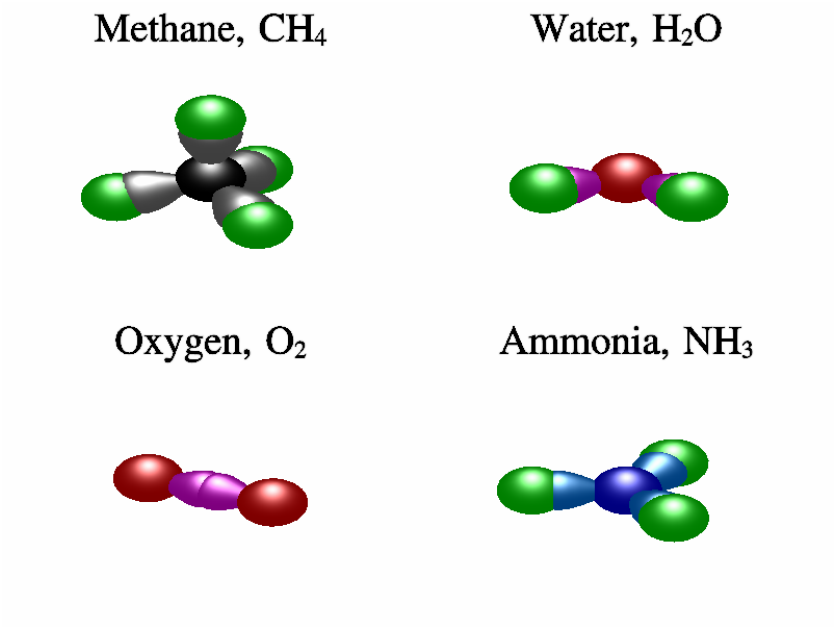
```
subplot 2 2 0 '':title 'Line, Curve, Rhomb, Ellipse' '' -1.5
line -1 -1 -0.5 1 'qAI'
curve -0.6 -1 1 1 0 1 1 1 'rA'
ball 0 -0.5 '*':ball 1 -0.1 '*'
rhomb 0 0.4 1 0.9 0.2 'b#'
rhomb 0 0 1 0.4 0.2 'cg@'
ellipse 0 -0.5 1 -0.1 0.2 'u#'
ellipse 0 -1 1 -0.6 0.2 'm@'

light on
subplot 2 2 1:title 'Face[xyz]':rotate 50 60:box
facex 1 0 -1 1 1 'r':facey -1 -1 -1 1 1 'g':facez 1 -1 -1 -1 1 'b'
face -1 -1 1 -1 1 1 1 -1 0 1 1 1 'bmgr'

subplot 2 2 3 '':title 'Cone'
cone -0.7 -0.3 0 -0.7 0.7 0.5 0.2 0.1 'b':text -0.7 -0.7 'no edges\n(default)'
cone 0 -0.3 0 0 0.7 0.5 0.2 0.1 'g@':text 0 -0.7 'with edges\n('\@' style)'
cone 0.7 -0.3 0 0.7 0.7 0.5 0.2 0.1 'ry':text 0.7 -0.7 '"arrow" with\n{}gradient'

subplot 2 2 2 '':title 'Sphere and Drop'
line -0.9 0 1 0.9 0 1
text -0.9 -0.7 'sh=0':drop -0.9 0 0 1 0.5 'r' 0:ball -0.9 0 1 'k'
text -0.3 -0.7 'sh=0.33':drop -0.3 0 0 1 0.5 'r' 0.33:ball -0.3 0 1 'k'
text 0.3 -0.7 'sh=0.67':drop 0.3 0 0 1 0.5 'r' 0.67:ball 0.3 0 1 'k'
```

```
text 0.9 -0.7 'sh=1':drop 0.9 0 0 1 0.5 'r' 1:ball 0.9 0 1 'k'
```



Generally, you can create arbitrary new kind of plot using primitives. For example, MathGL don't provide any special functions for drawing molecules. However, you can do it using only one type of primitives [drop], page 29. The sample code is:

```
alpha on:light on
subplot 2 2 0 '':title 'Methane, CH_4':rotate 60 120
sphere 0 0 0 0.25 'k':drop 0 0 0 0 0 1 0.35 'h' 1 2:sphere 0 0 0.7 0.25 'g'
drop 0 0 0 -0.94 0 -0.33 0.35 'h' 1 2:sphere -0.66 0 -0.23 0.25 'g'
drop 0 0 0 0.47 0.82 -0.33 0.35 'h' 1 2:sphere 0.33 0.57 -0.23 0.25 'g'
drop 0 0 0 0.47 -0.82 -0.33 0.35 'h' 1 2:sphere 0.33 -0.57 -0.23 0.25 'g'

subplot 2 2 1 '':title 'Water, H{_2}O':rotate 60 100
sphere 0 0 0 0.25 'r':drop 0 0 0 0.3 0.5 0 0.3 'm' 1 2:sphere 0.3 0.5 0 0.25 'g'
drop 0 0 0 0.3 -0.5 0 0.3 'm' 1 2:sphere 0.3 -0.5 0 0.25 'g'

subplot 2 2 2 '':title 'Oxygen, O_2':rotate 60 120
drop 0 0.5 0 0 -0.3 0 0.3 'm' 1 2:sphere 0 0.5 0 0.25 'r'
drop 0 -0.5 0 0 0.3 0 0.3 'm' 1 2:sphere 0 -0.5 0 0.25 'r'

subplot 2 2 3 '':title 'Ammonia, NH_3':rotate 60 120
sphere 0 0 0 0.25 'b':drop 0 0 0 0.33 0.57 0 0.32 'n' 1 2
sphere 0.33 0.57 0 0.25 'g':drop 0 0 0 0.33 -0.57 0 0.32 'n' 1 2
sphere 0.33 -0.57 0 0.25 'g':drop 0 0 0 -0.65 0 0 0.32 'n' 1 2
sphere -0.65 0 0 0.25 'g'
```



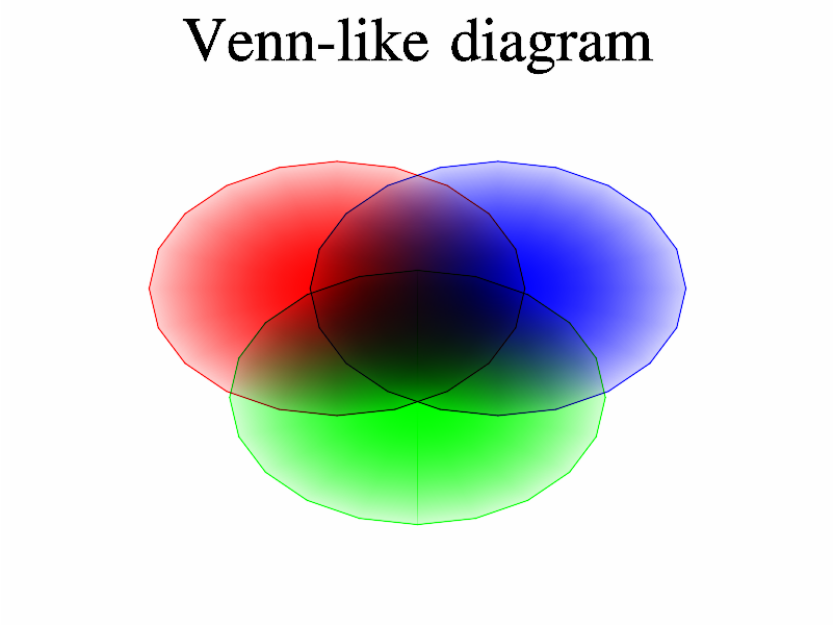
Moreover, some of special plots can be more easily produced by primitives rather than by specialized function. For example, Venn diagram can be produced by `Error` plot:

```
list x -0.3 0 0.3:list y 0.3 -0.3 0.3:list e 0.7 0.7 0.7
title 'Venn-like diagram':alpha on
error x y e e '!rgb@#o'
```

You see that you have to specify and fill 3 data arrays. The same picture can be produced by just 3 calls of [circle], page 29, function:

```
title 'Venn-like diagram':alpha on
circle -0.3 0.3 0.7 'rr@'
circle 0 -0.3 0.7 'gg@'
circle 0.3 0.3 0.7 'bb@'
```

Of course, the first variant is more suitable if you need to plot a lot of circles. But for few ones the usage of primitives looks easy.

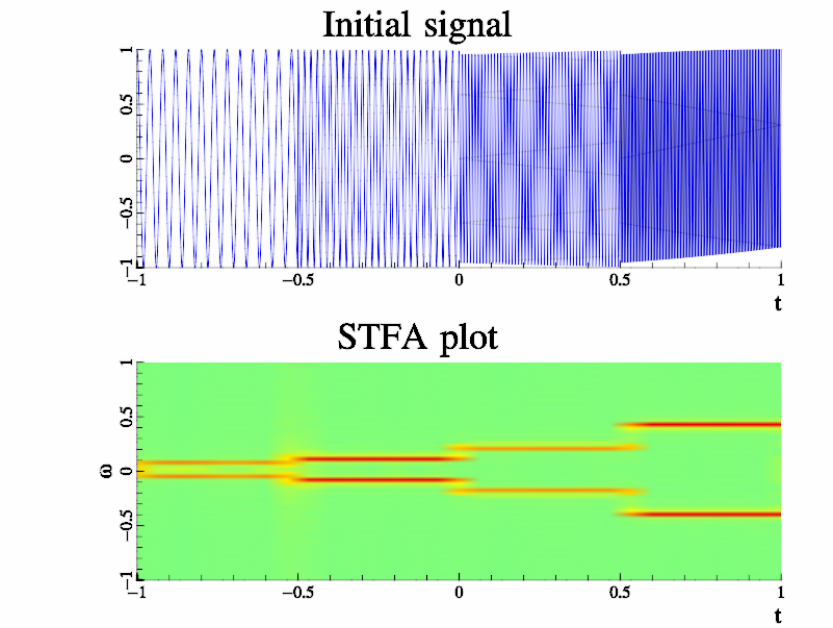


5.9.8 STFA sample

Short-time Fourier Analysis ([stfa], page 46) is one of informative method for analyzing long rapidly oscillating 1D data arrays. It is used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time.

MathGL can find and draw STFA result. Just to show this feature I give following sample. Initial data arrays is 1D arrays with step-like frequency. Exactly this you can see at bottom on the STFA plot. The sample code is:

```
new a 2000:new b 2000
fill a 'cos(50*pi*x)*(x<-.5)+cos(100*pi*x)*(x<0)*(x>-.5)+\
cos(200*pi*x)*(x<.5)*(x>0)+cos(400*pi*x)*(x>.5)'

subplot 1 2 0 '<_':title 'Initial signal'
plot a:axis:xlabel '\i t'

subplot 1 2 1 '<_':title 'STFA plot'
stfa a b 64:axis:ylabel '\omega' 0:xlabel '\i t'
```



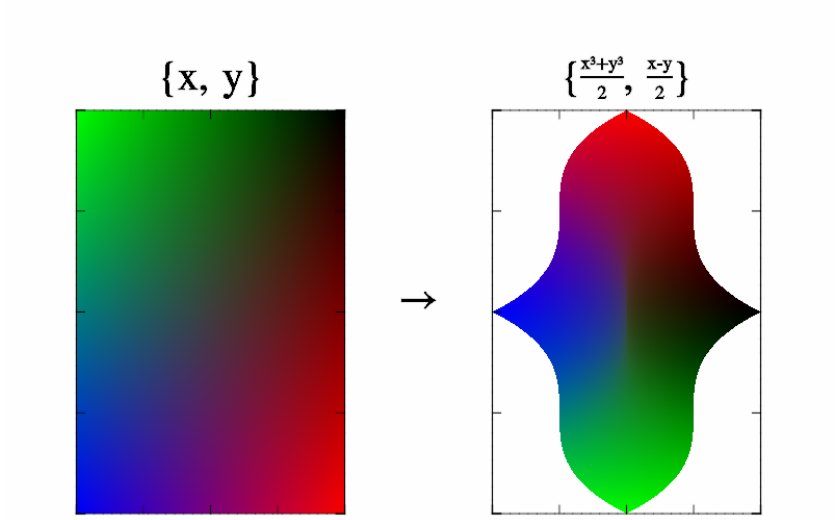
5.9.9 Mapping visualization

Sometime ago I worked with mapping and have a question about its visualization. Let me remember you that mapping is some transformation rule for one set of number to another one. The 1d mapping is just an ordinary function – it takes a number and transforms it to another one. The 2d mapping (which I used) is a pair of functions which take 2 numbers and transform them to another 2 ones. Except general plots (like [surfc], page 44, [surfa], page 45) there is a special plot – Arnold diagram. It shows the area which is the result of mapping of some initial area (usually square).

I tried to make such plot in [map], page 46. It shows the set of points or set of faces, which final position is the result of mapping. At this, the color gives information about their initial position and the height describes Jacobian value of the transformation. Unfortunately, it looks good only for the simplest mapping but for the real multivalent quasi-chaotic mapping it produces a confusion. So, use it if you like :).

The sample code for mapping visualization is:

```
new a 50 40 'x':new b 50 40 'y':zrange -2 2:text 0 0 '\to'
subplot 2 1 0:text 0 1.1 '\{x, y\}' '' -2:box
map a b 'brgk'

subplot 2 1 1:box
text 0 1.1 '\{\frac{x^3+y^3}{2}, \frac{x-y}{2}\}' '' -2
fill a '(x^3+y^3)/2':fill b '(x-y)/2':map a b 'brgk'
```



5.9.10 Data interpolation

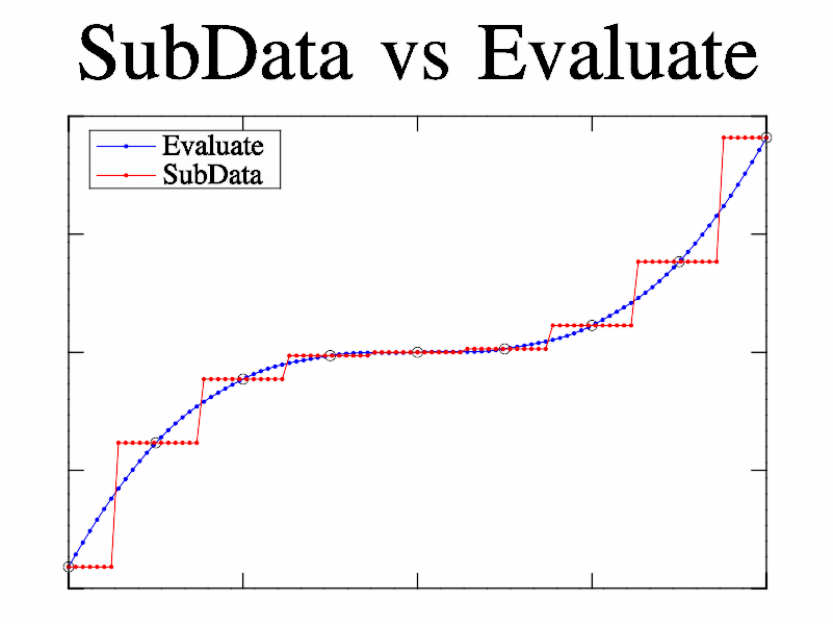
There are many functions to get interpolated values of a data array. Basically all of them can be divided by 3 categories:

1. functions which return single value at given point (see Section 4.8 [Interpolation], page 63, and `mglGSpline()` in Section 4.11 [Global functions], page 65);
2. functions [subdata], page 58, and [evaluate], page 59, for indirect access to data elements;
3. functions [refill], page 57, [gspline], page 57, and [datagrid], page 56, which fill regular (rectangular) data array by interpolated values.

The usage of first category is rather straightforward and don't need any special comments.

There is difference in indirect access functions. Function [subdata], page 58, use use step-like interpolation to handle correctly single `nan` values in the data array. Contrary, function [evaluate], page 59, use local spline interpolation, which give smoother output but spread `nan` values. So, [subdata], page 58, should be used for specific data elements (for example, for given column), and [evaluate], page 59, should be used for distributed elements (i.e. consider data array as some field). Following sample illustrates this difference:

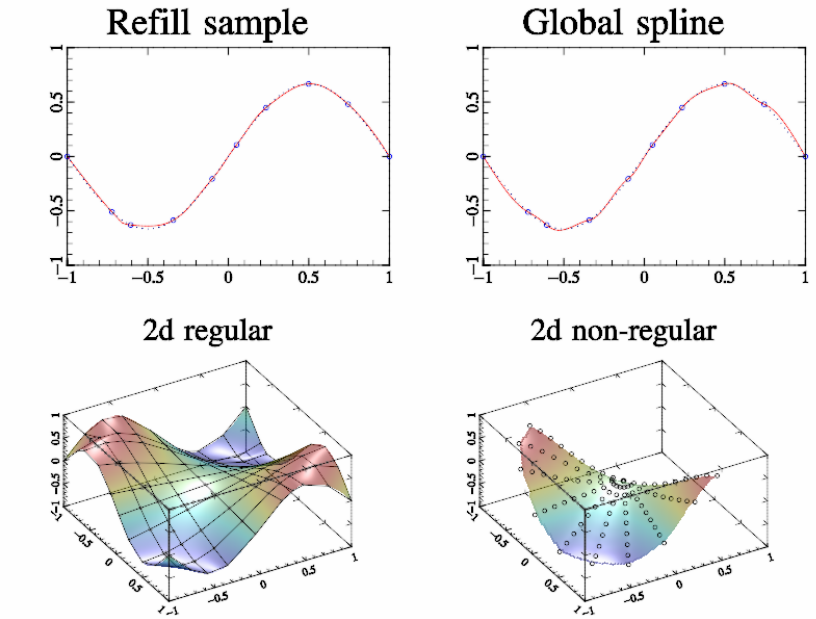
```
subplot 1 1 0 '':title 'SubData vs Evaluate'
new in 9 'x^3/1.1':plot in 'ko ':box
new arg 99 '4*x+4'
evaluate e in arg off:plot e 'b.'; legend 'Evaluate'
subdata s in arg:plot s 'r.';legend 'SubData'
legend 2
```



Example of [datagrid], page 56, usage is done in Section 5.9.11 [Making regular data], page 160. Here I want to show the peculiarities of [refill], page 57, and [gspline], page 57, functions. Both functions require argument(s) which provide coordinates of the data values, and return rectangular data array which equidistantly distributed in axis range. So, in opposite to [evaluate], page 59, function, [refill], page 57, and [gspline], page 57, can interpolate non-equidistantly distributed data. At this both functions [refill], page 57, and [gspline], page 57, provide continuity of 2nd derivatives along coordinate(s). However, [refill], page 57, is slower but give better (from human point of view) result than global spline [gspline], page 57, due to more advanced algorithm. Following sample illustrates this difference:

```
new x 10 '0.5+rnd':cumsum x 'x':norm x -1 1
copy y sin(pi*x)/1.5
subplot 2 2 0 '<_':title 'Refill sample'
box:axis:plot x y 'o ':fplot 'sin(pi*x)/1.5' 'B:'
new r 100:refill r x y:plot r 'r'

subplot 2 2 1 '<_':title 'Global spline'
box:axis:plot x y 'o ':fplot 'sin(pi*x)/1.5' 'B:'
new r 100:gspline r x y:plot r 'r'

new y 10 '0.5+rnd':cumsum y 'x':norm y -1 1
copy xx x:extend xx 10
copy yy y:extend yy 10:transpose yy
copy z sin(pi*xx*yy)/1.5
alpha on:light on
subplot 2 2 2:title '2d regular':rotate 40 60
box:axis:mesh xx yy z 'k'
new rr 100 100:refill rr x y z:surf rr
```

```
new xx 10 10 '(x+1)/2*cos(y*pi/2-1)'
new yy 10 10 '(x+1)/2*sin(y*pi/2-1)'
copy z sin(pi*xx*yy)/1.5
subplot 2 2 3:title '2d non-regular':rotate 40 60
box:axis:plot xx yy z 'ko '
new rr 100 100:refill rr xx yy z:surf rr
```

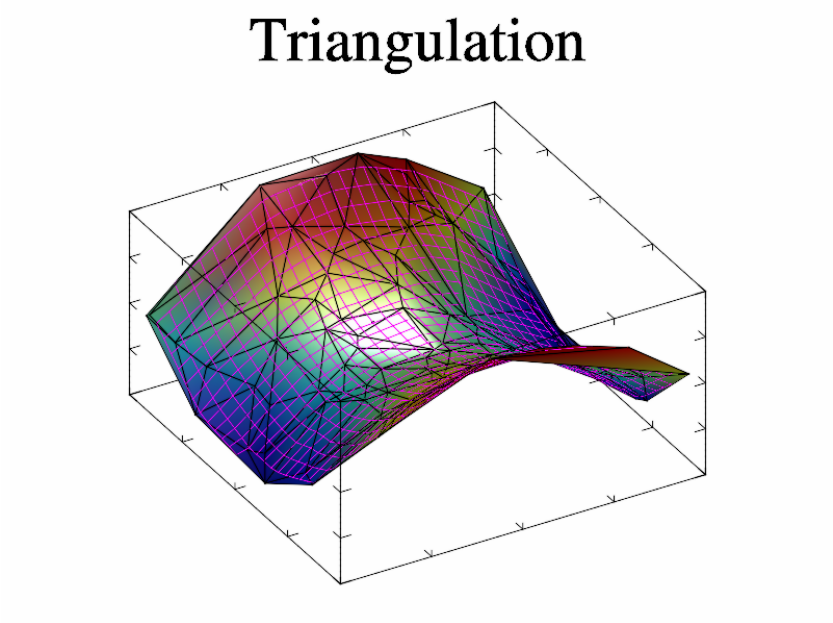


5.9.11 Making regular data

Sometimes, one have only unregular data, like as data on triangular grids, or experimental results and so on. Such kind of data cannot be used as simple as regular data (like matrices). Only few functions, like [dots], page 51, can handle unregular data as is.

However, one can use built in triangulation functions for interpolating unregular data points to a regular data grids. There are 2 ways. First way, one can use [triangulation], page 67, function to obtain list of vertexes for triangles. Later this list can be used in functions like [triplot], page 50, or [tricont], page 50. Second way consist in usage of [datagrid], page 56, function, which fill regular data grid by interpolated values, assuming that coordinates of the data grid is equidistantly distributed in axis range. Note, you can use options (see Section 2.7 [Command options], page 15) to change default axis range as well as in other plotting functions.

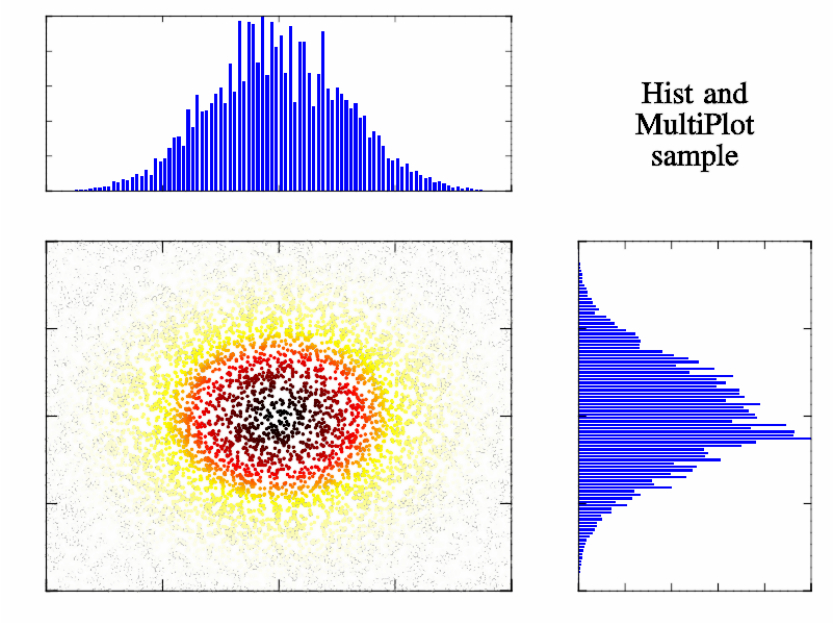
```
new x 100 '2*rnd-1':new y 100 '2*rnd-1':copy z x^2-y^2
# first way - plot triangular surface for points
triangulate d x y
title 'Triangulation'
rotate 50 60:box:light on
triplot d x y z:triplot d x y z '#k'
# second way - make regular data and plot it
new g 30 30:datagrid g x y z:mesh g 'm'
```



5.9.12 Making histogram

Using the [hist], page 59, function(s) for making regular distributions is one of useful fast methods to process and plot irregular data. `Hist` can be used to find some momentum of set of points by specifying weight function. It is possible to create not only 1D distributions but also 2D and 3D ones. Below I place the simplest sample code which demonstrate [hist], page 59, usage:

```
new x 10000 '2*rnd-1':new y 10000 '2*rnd-1':copy z exp(-6*(x^2+y^2))
hist xx x z:norm xx 0 1:hist yy y z:norm yy 0 1
multiplot 3 3 3 2 2 '':ranges -1 1 -1 1 0 1:box:dots x y z 'wyrRk'
multiplot 3 3 0 2 1 '':ranges -1 1 0 1:box:bars xx
multiplot 3 3 5 1 2 '':ranges 0 1 -1 1:box:barh yy
subplot 3 3 2:text 0.5 0.5 'Hist and\n{}MultiPlot\n{}sample' 'a' -3
```



5.9.13 Nonlinear fitting hints

Nonlinear fitting is rather simple. All that you need is the data to fit, the approximation formula and the list of coefficients to fit (better with its initial guess values). Let me demonstrate it on the following simple example. First, let us use sin function with some random noise:

```
new dat 100 '0.4*rnd+0.1+sin(2*pi*x)'
new in 100 '0.3+sin(2*pi*x)'
```

and plot it to see that data we will fit

```
title 'Fitting sample':yrange -2 2:box:axis:plot dat 'k. '
```

The next step is the fitting itself. For that let me specify an initial values *ini* for coefficients '`abc`' and do the fitting for approximation formula '`a+b*sin(c*x)`'

```
list ini 1 1 3:fit res dat 'a+b*sin(c*x)' 'abc' ini
```

Now display it

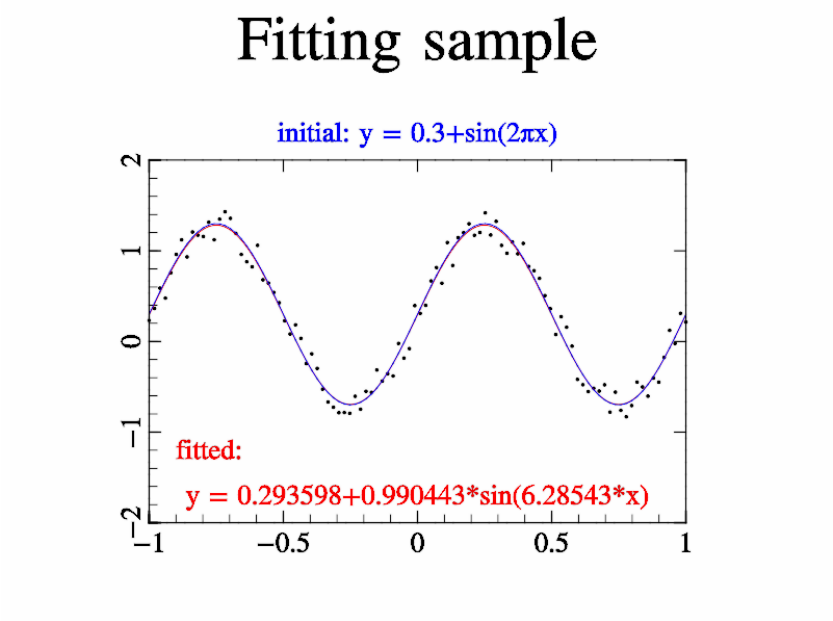
```
plot res 'r':plot in 'b'
text -0.9 -1.3 'fitted:' 'r:L'
putsfit 0 -1.8 'y = ' 'r'
text 0 2.2 'initial: y = 0.3+sin(2\pi x)' 'b'
```

NOTE! the fitting results may have strong dependence on initial values for coefficients due to algorithm features. The problem is that in general case there are several local "optimums" for coefficients and the program returns only first found one! There are no guaranties that it will be the best. Try for example to set `ini[3] = {0, 0, 0}` in the code above.

The full sample code for nonlinear fitting is:

```
new dat 100 '0.4*rnd+0.1+sin(2*pi*x)'
new in 100 '0.3+sin(2*pi*x)'
list ini 1 1 3:fit res dat 'a+b*sin(c*x)' 'abc' ini
```

```
title 'Fitting sample':yrange -2 2:box:axis:plot dat 'k. '
plot res 'r':plot in 'b'
text -0.9 -1.3 'fitted:' 'r:L'
putsfit 0 -1.8 'y = ' 'r'
text 0 2.2 'initial: y = 0.3+sin(2\pi x)' 'b'
```



5.9.14 PDE solving hints

Solving of Partial Differential Equations (PDE, including beam tracing) and ray tracing (or finding particle trajectory) are more or less common task. So, MathGL have several functions for that. There are [ray], page 66, for ray tracing, [pde], page 66, for PDE solving, [qo2d], page 66, for beam tracing in 2D case (see Section 4.11 [Global functions], page 65). Note, that these functions take "Hamiltonian" or equations as string values. And I don't plan now to allow one to use user-defined functions. There are 2 reasons: the complexity of corresponding interface; and the basic nature of used methods which are good for samples but may not good for serious scientific calculations.

The ray tracing can be done by [ray], page 66, function. Really ray tracing equation is Hamiltonian equation for 3D space. So, the function can be also used for finding a particle trajectory (i.e. solve Hamiltonian ODE) for 1D, 2D or 3D cases. The function have a set of arguments. First of all, it is Hamiltonian which defined the media (or the equation) you are planning to use. The Hamiltonian is defined by string which may depend on coordinates 'x', 'y', 'z', time 't' (for particle dynamics) and momentums 'p'=p_x, 'q'=p_y, 'v'=p_z. Next, you have to define the initial conditions for coordinates and momentums at 't'=0 and set the integrations step (default is 0.1) and its duration (default is 10). The Runge-Kutta method of 4-th order is used for integration.

```
const char *ham = "p^2+q^2-x-1+i*0.5*(y+x)*(y>-x)";
mglData r = mglRay(ham, mglPoint(-0.7, -1), mglPoint(0, 0.5), 0.02, 2);
```

This example calculate the reflection from linear layer (media with Hamiltonian '`p^2+q^2-x-1`'=$p_x^2 + p_y^2 - x - 1$). This is parabolic curve. The resulting array have 7 columns which contain data for {x,y,z,p,q,v,t}.

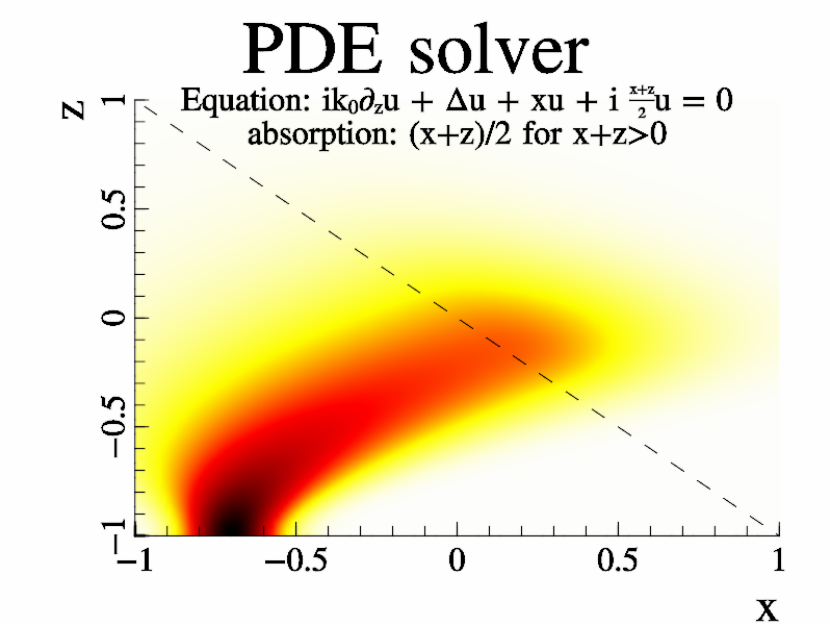
The solution of PDE is a bit more complicated. As previous you have to specify the equation as pseudo-differential operator $\hat{H}(x, \nabla)$ which is called sometime as "Hamiltonian" (for example, in beam tracing). As previously, it is defined by string which may depend on coordinates '`x`', '`y`', '`z`' (but not time!), momentums '`p`'=$(d/dx)/ik_0$, '`q`'=$(d/dy)/ik_0$ and field amplitude '`u`'=$|u|$. The evolutionary coordinate is '`z`' in all cases. So that, the equation look like $du/dz = ik_0 H(x, y, \hat{p}, \hat{q}, |u|)[u]$. Dependence on field amplitude '`u`'=$|u|$ allows one to solve nonlinear problems too. For example, for nonlinear Shrodinger equation you may set `ham="p^2 + q^2 - u^2"`. Also you may specify imaginary part for wave absorption, like `ham = "p^2 + i*x*(x>0)"` or `ham = "p^2 + i1*x*(x>0)"`.

Next step is specifying the initial conditions at '`z`' equal to minimal z-axis value. The function need 2 arrays for real and for imaginary part. Note, that coordinates x,y,z are supposed to be in specified axis range. So, the data arrays should have corresponding scales. Finally, you may set the integration step and parameter k0=k_0. Also keep in mind, that internally the 2 times large box is used (for suppressing numerical reflection from boundaries) and the equation should well defined even in this extended range.

Final comment is concerning the possible form of pseudo-differential operator H. At this moment, simplified form of operator H is supported – all "mixed" terms (like '`x*p`'->x*d/dx) are excluded. For example, in 2D case this operator is effectively $H = f(p, z) + g(x, z, u)$. However commutable combinations (like '`x*q`'->x*d/dy) are allowed for 3D case.

So, for example let solve the equation for beam deflected from linear layer and absorbed later. The operator will have the form '`"p^2+q^2-x-1+i*0.5*(z+x)*(z>-x)"`' that correspond to equation $1/ik_0 * du/dz + d^2u/dx^2 + d^2u/dy^2 + x * u + i(x + z)/2 * u = 0$. This is typical equation for Electron Cyclotron (EC) absorption in magnetized plasmas. For initial conditions let me select the beam with plane phase front $exp(-48 * (x + 0.7)^2)$. The corresponding code looks like this:

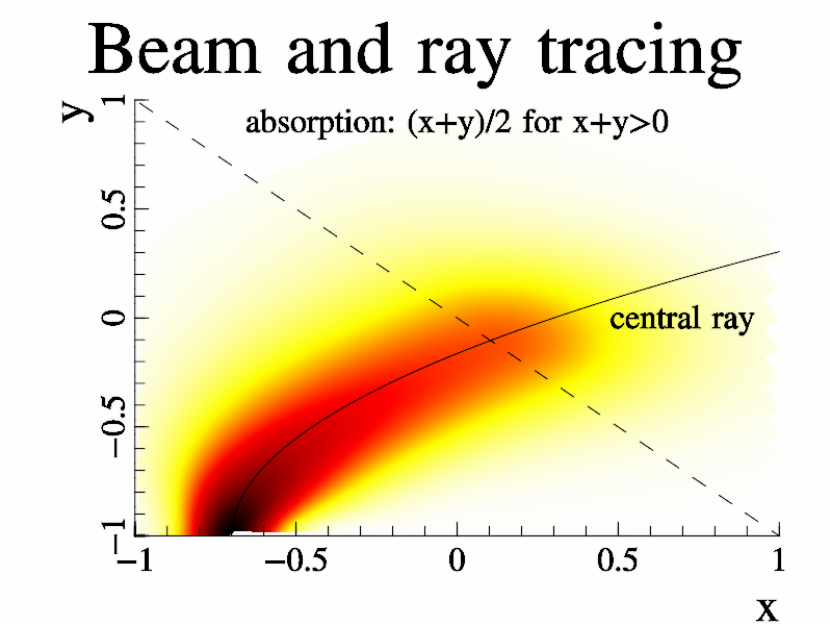
```
new re 128 'exp(-48*(x+0.7)^2)':new im 128
pde a 'p^2+q^2-x-1+i*0.5*(z+x)*(z>-x)' re im 0.01 30
transpose a
subplot 1 1 0 '<_':title 'PDE solver'
axis:xlabel '\i x':ylabel '\i z'
crange 0 1:dens a 'wyrRk'
fplot '-x' 'k|'
text 0 0.95 'Equation: ik_0\partial_zu + \Delta u + x\cdot u +\
 i \frac{x+z}{2}\cdot u = 0\n{}absorption: (x+z)/2 for x+z>0'
```



The last example is example of beam tracing. Beam tracing equation is special kind of PDE equation written in coordinates accompanied to a ray. Generally this is the same parameters and limitation as for PDE solving but the coordinates are defined by the ray and by parameter of grid width w in direction transverse the ray. So, you don't need to specify the range of coordinates. **BUT** there is limitation. The accompanied coordinates are well defined only for smooth enough rays, i.e. then the ray curvature K (which is defined as $1/K^2 = (|r''|^2|r'|^2 - (r'', r'')^2)/|r'|^6$) is much large then the grid width: $K >> w$. So, you may receive incorrect results if this condition will be broken.

You may use following code for obtaining the same solution as in previous example:

```
define $1 'p^2+q^2-x-1+i*0.5*(y+x)*(y>-x)'
subplot 1 1 0 '<_':title 'Beam and ray tracing'
ray r $1 -0.7 -1 0 0 0.5 0 0.02 2:plot r(0) r(1) 'k'
axis:xlabel '\i x':ylabel '\i z'
new re 128 'exp(-48*x^2)':new im 128
new xx 1:new yy 1
qo2d a $1 re im r 1 30 xx yy
crange 0 1:dens xx yy a 'wyrRk':fplot '-x' 'k|'
text 0 0.85 'absorption: (x+y)/2 for x+y>0'
text 0.7 -0.05 'central ray'
```



5.9.15 Drawing phase plain

Here I want say a few words of plotting phase plains. Phase plain is name for system of coordinates x, x', i.e. a variable and its time derivative. Plot in phase plain is very useful for qualitative analysis of an ODE, because such plot is rude (it topologically the same for a range of ODE parameters). Most often the phase plain $\{x, x'\}$ is used (due to its simplicity), that allows to analyze up to the 2nd order ODE (i.e. $x'' + f(x, x') = 0$).

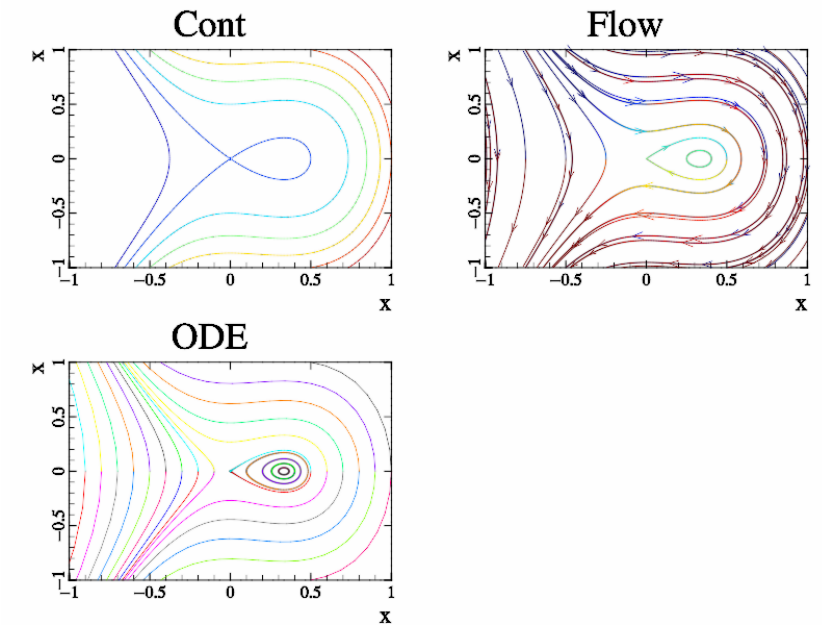
The simplest way to draw phase plain in MathGL is using [flow], page 48, function(s), which automatically select several points and draw flow threads. If the ODE have an integral of motion (like Hamiltonian $H(x, x') = const$ for dissipation-free case) then you can use [cont], page 41, function for plotting isolines (contours). In fact. isolines are the same as flow threads, but without arrows on it. Finally, you can directly solve ODE using [ode], page 66, function and plot its numerical solution.

Let demonstrate this for ODE equation $x'' - x + 3 * x^2 = 0$. This is nonlinear oscillator with square nonlinearity. It has integral $H = y^2 + 2 * x^3 - x^2 = Const$. Also it have 2 typical stationary points: saddle at {x=0, y=0} and center at {x=1/3, y=0}. Motion at vicinity of center is just simple oscillations, and is stable to small variation of parameters. In opposite, motion around saddle point is non-stable to small variation of parameters, and is very slow. So, calculation around saddle points are more difficult, but more important. Saddle points are responsible for solitons, stochasticity and so on.

So, let draw this phase plain by 3 different methods. First, draw isolines for $H = y^2 + 2 * x^3 - x^2 = Const$ – this is simplest for ODE without dissipation. Next, draw flow threads – this is straightforward way, but the automatic choice of starting points is not always optimal. Finally, use [ode], page 66, to check the above plots. At this we need to run [ode], page 66, in both direction of time (in future and in the past) to draw whole plain. Alternatively, one can put starting points far from (or at the bounding box as done in [flow], page 48) the plot, but this is a more complicated. The sample code is:

```
subplot 2 2 0 '<_':title 'Cont':box
axis:xlabel 'x':ylabel '\dot{x}'
```

```
new f 100 100 'y^2+2*x^3-x^2-0.5':cont f

subplot 2 2 1 '<_':title 'Flow':box
axis:xlabel 'x':ylabel '\dot{x}'
new fx 100 100 'x-3*x^2'
new fy 100 100 'y'
flow fy fx 'v';value 7

subplot 2 2 2 '<_':title 'ODE':box
axis:xlabel 'x':ylabel '\dot{x}'
for $x -1 1 0.1
  ode r 'y;x-3*x^2' 'xy' [$x,0]
  plot r(0) r(1)
  ode r '-y;-x+3*x^2' 'xy' [$x,0]
  plot r(0) r(1)
next
```

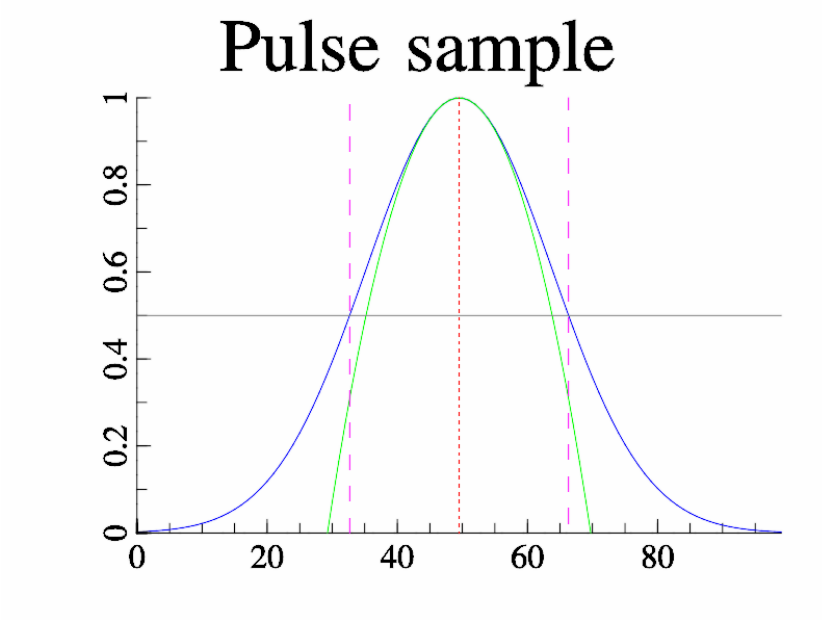


5.9.16 Pulse properties

There is common task in optics to determine properties of wave pulses or wave beams. MathGL provide special function [pulse], page 60, which return the pulse properties (maximal value, center of mass, width and so on). Its usage is rather simple. Here I just illustrate it on the example of Gaussian pulse, where all parameters are obvious.

```
subplot 1 1 0 '<_':title 'Pulse sample'
# first prepare pulse itself
new a 100 'exp(-6*x^2)'

# get pulse parameters
```

```
pulse b a 'x'

# positions and widths are normalized on the number of points. So, set proper axis scale.
ranges 0 a.nx-1 0 1
axis:plot a # draw pulse and axis

# now visualize found pulse properties
define m a.max # maximal amplitude
# approximate position of maximum
line b(1) 0 b(1) m 'r='
# width at half-maximum (so called FWHM)
line b(1)-b(3)/2 0  b(1)-b(3)/2 m 'm|'
line b(1)+b(3)/2 0  b(1)+b(3)/2 m 'm|'
line 0 0.5*m a.nx-1 0.5*m 'h'
# parabolic approximation near maximum
new x 100 'x'
plot b(0)*(1-((x-b(1))/b(2))^2) 'g'
```

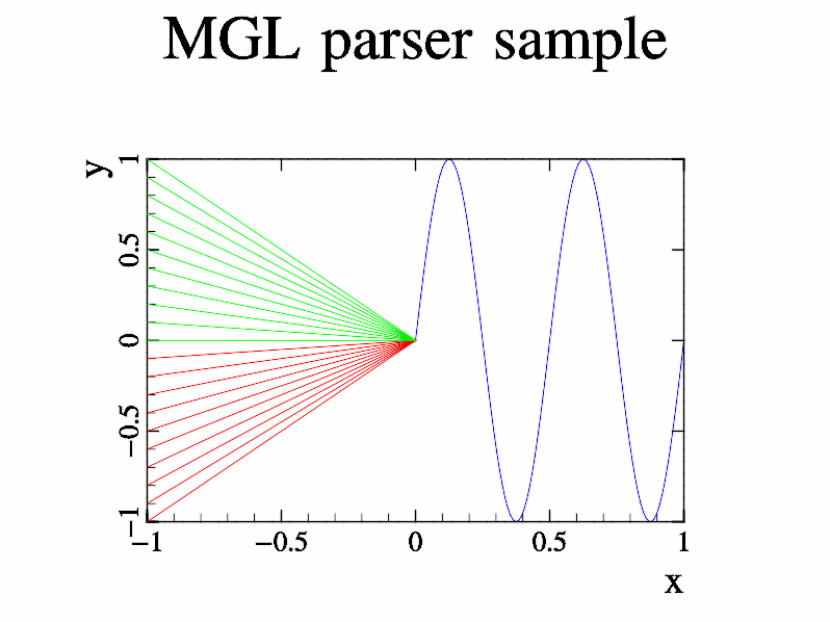


5.9.17 Using MGL parser

MGL scripts can contain loops, conditions and user-defined functions. Below I show very simple example of its usage:

```
title 'MGL parser sample'
call 'sample'
stop

func 'sample'
new dat 100 'sin(2*pi*(x+1))'
```

```
plot dat; xrange 0 1
box:axis:xlabel 'x':ylabel 'y'
for $0 -1 1 0.1
if $0<0
line 0 0 -1 $0 'r'
else
line 0 0 -1 $0 'r'
endif
next
```

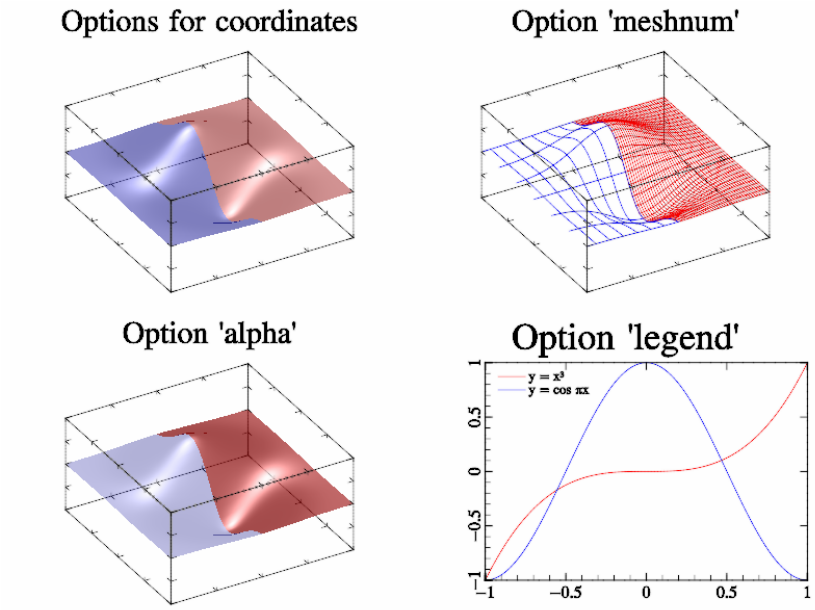


5.9.18 Using options

Section 2.7 [Command options], page 15, allow the easy setup of the selected plot by changing global settings only for this plot. Often, options are used for specifying the range of automatic variables (coordinates). However, options allows easily change plot transparency, numbers of line or faces to be drawn, or add legend entries. The sample function for options usage is:

```
new a 31 41 '-pi*x*exp(-(y+1)^2-4*x^2)'
alpha on:light on
subplot 2 2 0:title 'Options for coordinates':rotate 40 60:box
surf a 'r';yrange 0 1
surf a 'b';yrange 0 -1

subplot 2 2 1:title 'Option "meshnum"':rotate 40 60:box
mesh a 'r'; yrange 0 1
mesh a 'b';yrange 0 -1; meshnum 5

subplot 2 2 2:title 'Option "alpha"':rotate 40 60:box
```

```
surf a 'r';yrange 0 1; alpha 0.7
surf a 'b';yrange 0 -1; alpha 0.3

subplot 2 2 3 '<_':title 'Option "legend"'
fplot 'x^3' 'r'; legend 'y = x^3'
fplot 'cos(pi*x)' 'b'; legend 'y = cos \pi x'
box:axis:legend 2
```



5.9.19 "Templates"

As I have noted before, the change of settings will influence only for the further plotting commands. This allows one to create "template" function which will contain settings and primitive drawing for often used plots. Correspondingly one may call this template-function for drawing simplification.

For example, let one has a set of points (experimental or numerical) and wants to compare it with theoretical law (for example, with exponent law $\exp(-x/2), x \in [0, 20]$). The template-function for this task is:

```
void template(mglGraph *gr)
{
  mglData  law(100);      // create the law
  law.Modify("exp(-10*x)");
  gr->SetRanges(0,20, 0.0001,1);
  gr->SetFunc(0,"lg(y)",0);
  gr->Plot(law,"r2");
  gr->Puts(mglPoint(10,0.2),"Theoretical law: e^x","r:L");
  gr->Label('x',"x val."); gr->Label('y',"y val.");
  gr->Axis(); gr->Grid("xy","g;"); gr->Box();
}
```

At this, one will only write a few lines for data drawing:

```
template(gr);     // apply settings and default drawing from template
mglData dat("fname.dat"); // load the data
// and draw it (suppose that data file have 2 columns)
gr->Plot(dat.SubData(0),dat.SubData(1),"bx ");
```

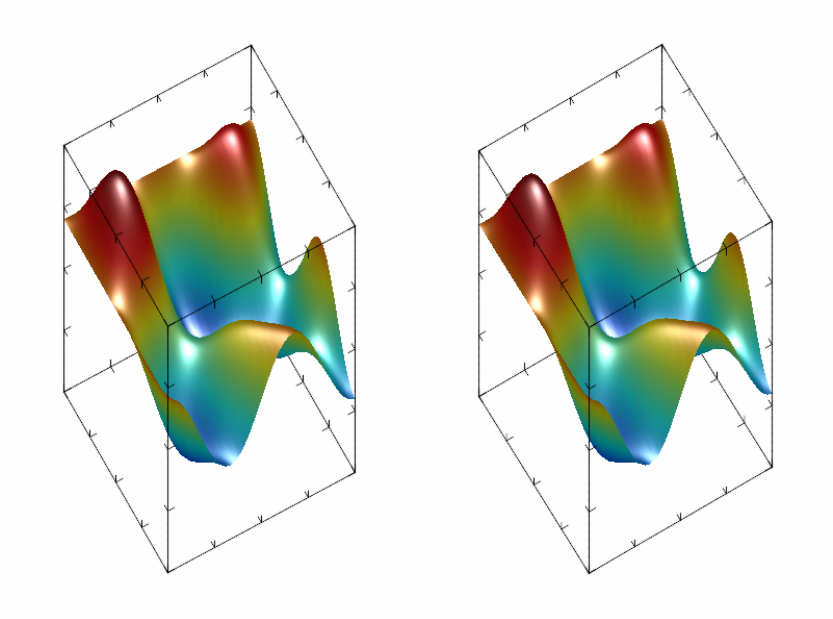
A template-function can also contain settings for font, transparency, lightning, color scheme and so on.

I understand that this is obvious thing for any professional programmer, but I several times receive suggestion about “templates” ... So, I decide to point out it here.

5.9.20 Stereo image

One can easily create stereo image in MathGL. Stereo image can be produced by making two subplots with slightly different rotation angles. The corresponding code looks like this:

```
call 'prepare2d'
light on
subplot 2 1 0:rotate 50 60+1:box:surf a
subplot 2 1 1:rotate 50 60-1:box:surf a
```



5.9.21 Reduce memory usage

By default MathGL save all primitives in memory, rearrange it and only later draw them on bitmaps. Usually, this speed up drawing, but may require a lot of memory for plots which contain a lot of faces (like [cloud], page 43, [dew], page 48). You can use [quality], page 27, function for setting to use direct drawing on bitmap and bypassing keeping any primitives in memory. This function also allow you to decrease the quality of the resulting image but increase the speed of the drawing.

The code for lower memory usage looks like this:

```
quality 6  # firstly, set to draw directly on bitmap
for $1 0 1000
  sphere 2*rnd-1 2*rnd-1 0.05
next
```

5.9.22 Scanning file

MathGL have possibilities to write textual information into file with variable values by help of [save], page 58, command. This is rather useful for generating an ini-files or preparing human-readable textual files. For example, lets create some textual file

```
subplot 1 1 0 '<_':title 'Save and scanfile sample'
list a 1 -1 0
save 'This is test: 0 -> ',a(0),' q' 'test.txt' 'w'
save 'This is test: 1 -> ',a(1),' q' 'test.txt'
save 'This is test: 2 -> ',a(2),' q' 'test.txt'
```

It contents look like

```
This is test: 0 -> 1 q
This is test: 1 -> -1 q
This is test: 2 -> 0 q
```

Note, that I use option '`w`' at first call of `save` to overwrite the contents of the file.

Let assume now that you want to read this values (i.e. [[0,1],[1,-1],[2,0]]) from the file. You can use [scanfile], page 57, for that. The desired values was written using template '`This is test: %g -> %g q`'. So, just use

```
scanfile a 'test.txt' 'This is test: %g -> %g'
```

and plot it to for assurance

```
ranges a(0) a(1):axis:plot a(0) a(1) 'o'
```

Note, I keep only the leading part of template (i.e. '`This is test: %g -> %g`' instead of '`This is test: %g -> %g q`'), because there is no important for us information after the second number in the line.

5.10 FAQ

The plot does not appear

Check that points of the plot are located inside the bounding box and resize the bounding box using [ranges], page 21, function. Check that the data have correct dimensions for selected type of plot. Sometimes the light reflection from flat surfaces (like, [dens], page 41) can look as if the plot were absent.

I can not find some special kind of plot.

Most "new" types of plots can be created by using the existing drawing functions. For example, the surface of curve rotation can be created by a special function [torus], page 39, or as a parametrically specified surface by [surf], page 40. See also, Section 5.9 [Hints], page 145. If you can not find a specific type of plot, please e-mail me and this plot will appear in the next version of MathGL library.

How can I print in Russian/Spanish/Arabic/Japanese, and so on?

The standard way is to use Unicode encoding for the text output. But the MathGL library also has interface for 8-bit (char *) strings with internal conversion to Unicode. This conversion depends on the current locale OS.

How can I exclude a point or a region of plot from the drawing?

There are 3 general ways. First, the point with `nan` value as one of the coordinates (including color/alpha range) will never be plotted. Second, special functions define the condition when the points should be omitted (see Section 3.2.5 [Cutting], page 19). Last, you may change the transparency of a part of the plot by the help of functions [surfa], page 45, [surf3a], page 45, (see Section 3.14 [Dual plotting], page 44). In last case the transparency is switched on smoothly.

How many people write this library?

Most of the library was written by one person. This is a result of nearly a year of work (mostly in the evening and on holidays): I spent half a year to write the kernel and half a year to a year on extending, improving the library and writing documentation. This process continues now :). The build system (cmake files) was written mostly by D.Kulagin, and the export to PRC/PDF was written mostly by M.Vidassov.

How can I display a bitmap on the figure?

You can import data by command [import], page 58, and display it by [dens], page 41, function. For example, for black-and-white bitmap you can use the code: `import bmp 'fname.png' 'wk':dens bmp 'wk'`.

How can I create 3D in PDF?

Just use command `write fname.pdf`, which create PDF file if enable-pdf=ON at MathGL configure.

How can I create TeX figure?

Just use command `write fname.tex`, which create LaTeX files with figure itself '*fname*`.tex`', with MathGL colors '`mglcolors.tex`' and main file '`mglmain.tex`'. Last one can be used for viewing image by command like `pdflatex mglmain.tex`.

How I can change the font family?

First, you should download new font files from here (`http://mathgl.sourceforge.net/download.html`) or from here (`http://sourceforge.net/project/showfiles.php?group_id=152187&package_id=267177`). Next, you should load the font files into by the following command: `loadfont 'fontname'`. Here *fontname* is the base font name like '`STIX`'. Use `loadfont ''` to start using the default font.

How can I draw tick out of a bounding box?

Just set a negative value in [ticklen], page 24. For example, use `ticklen -0.1`.

How can I prevent text rotation?

Just use `rotatetext off`. Also you can use axis style '`U`' for disable only tick labels rotation.

How can I draw equal axis range even for rectangular image?

Just use `aspect nan nan` for each subplot, or at the beginning of the drawing.

How I can set transparent background?

Just use code like `clf 'r{A5}'` or prepare PNG file and set it as background image by call `background 'fname.png'`.

How I can reduce "white" edges around bounding box?

The simplest way is to use [subplot], page 24, style. However, you should be careful if you plan to add [colorbar], page 32, or rotate plot – part of plot can be invisible if you will use non-default [subplot], page 24, style.

Can I combine bitmap and vector output in EPS?

Yes. Sometimes you may have huge surface and a small set of curves and/or text on the plot. You can use function [rasterize], page 27, just after making surface plot. This will put all plot to bitmap background. At this later plotting will be in vector format. For example, you can do something like following:

```
surf x y z
rasterize # make surface as bitmap
axis
write 'fname.eps'
```

Appendix A Symbols and hot-keys

This appendix contain the full list of symbols (characters) used by MathGL for setting up plot. Also it contain sections for full list of hot-keys supported by mglview tool and by UDAV program.

A.1 Symbols for styles

Below is full list of all characters (symbols) which MathGL use for setting up the plot.

'`space ' ''
: empty line style (see Section 2.3 [Line styles], page 9);

  empty color in [chart], page 36.

'`!`'
: set to use new color from palette for each point (not for each curve, as default) in Section 3.11 [1D plotting], page 34;

  set to disable ticks tuning in [axis], page 31, and [colorbar], page 32;

  set to draw [grid], page 32, lines at subticks coordinates too;

  define complex variable/expression in MGL script if placed at beginning.

'`#`'
: set to use solid marks (see Section 2.3 [Line styles], page 9) or solid [error], page 37, boxes;

  set to draw wired plot for [axial], page 42, [surf3], page 42, [surf3a], page 45, [surf3c], page 45, [triplot], page 50, [quadplot], page 51, [area], page 35, [bars], page 35, [barh], page 36, [tube], page 39, [tape], page 35, [cone], page 29, [boxs], page 40, and draw boundary only for [circle], page 29, [ellipse], page 29, [rhomb], page 29;

  set to draw also mesh lines for [surf], page 40, [surfc], page 44, [surfa], page 45, [dens], page 41, [densx], page 49, [densy], page 49, [densz], page 49, [dens3], page 43, or boundary for [chart], page 36, [facex], page 28, [facey], page 28, [facez], page 28, [rect], page 28;

  set to draw boundary and box for [legend], page 33, [title], page 25, or grid lines for [table], page 38;

  set to draw grid for [radar], page 34;

  set to start flow threads and pipes from edges only for [flow], page 48, [pipe], page 49;

  set to use whole are for axis range in [subplot], page 24, [inplot], page 25;

  change text color inside a string (see Section 2.5 [Font styles], page 13);

  start comment in Chapter 1 [MGL scripts], page 1, or in Section 2.7 [Command options], page 15.

'`$`'
: denote parameter of Chapter 1 [MGL scripts], page 1.

'`%`'
: set color scheme along 2 coordinates Section 2.4 [Color scheme], page 11.

'`&`'

  set to pass long integer number in tick template [xtick], page 23, [ytick], page 23, [ztick], page 23, [ctick], page 23.

  operation in Section 2.6 [Textual formulas], page 14.

‘''’ denote string in Chapter 1 [MGL scripts], page 1, or in Section 2.7 [Command options], page 15.

‘*’ one of marks (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
set to start flow threads from 2d array inside data (see [flow], page 48);
operation in Section 2.6 [Textual formulas], page 14.

‘+’ one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
set to print ‘+’ for positive numbers in [axis], page 31, [label], page 38, [table], page 38;
operation in Section 2.6 [Textual formulas], page 14.

‘,’ separator for color positions (see Section 2.2 [Color styles], page 9) or items in a list.

‘-’ solid line style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
place entries horizontally in [legend], page 33;
set to use usual ‘-’ for negative numbers in [axis], page 31, [label], page 38, [table], page 38;
operation in Section 2.6 [Textual formulas], page 14.

‘.’ one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;
set to draw hachures instead of arrows for [vect], page 47, [vect3], page 47;
set to use dots instead of faces for [cloud], page 43, [torus], page 39, [axial], page 42, [surf3], page 42, [surf3a], page 45, [surf3c], page 45, [surf], page 40, [surfa], page 45, [surfc], page 44, [dens], page 41, [map], page 46;
delimiter of fractional parts for numbers.

‘/’ operation in Section 2.6 [Textual formulas], page 14.

‘:’ line dashing style (see Section 2.3 [Line styles], page 9);
stop color scheme parsing (see Section 2.4 [Color scheme], page 11);
range operation in Chapter 1 [MGL scripts], page 1;
separator of commands in Chapter 1 [MGL scripts], page 1.

‘;’ line dashing style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
start of an option in Chapter 1 [MGL scripts], page 1, or in Section 2.7 [Command options], page 15.

‘<’ one of marks (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
style of [subplot], page 24, and [inplot], page 25;

set position of [colorbar], page 32;

style of [vect], page 47, [vect3], page 47;

align left in [bars], page 35, [barh], page 36, [boxplot], page 36, [cones], page 36, [candle], page 37, [ohlc], page 37;

operation in Section 2.6 [Textual formulas], page 14.

'>' one of marks (see Section 2.3 [Line styles], page 9);

one of mask for face filling (see Section 2.4 [Color scheme], page 11);

style of [subplot], page 24, and [inplot], page 25;

set position of [colorbar], page 32;

style of [vect], page 47, [vect3], page 47;

align right in [bars], page 35, [barh], page 36, [boxplot], page 36, [cones], page 36, [candle], page 37, [ohlc], page 37;

operation in Section 2.6 [Textual formulas], page 14.

'=' line dashing style (see Section 2.3 [Line styles], page 9);

one of mask for face filling (see Section 2.4 [Color scheme], page 11);

set to use equidistant columns for [table], page 38;

set to use color gradient for [vect], page 47, [vect3], page 47;

operation in Section 2.6 [Textual formulas], page 14.

'@' set to draw box around text for [text], page 30, and similar functions;

set to draw boundary and fill it for [circle], page 29, [ellipse], page 29, [rhomb], page 29;

set to fill faces for [box], page 33;

set to draw large semitransparent mark instead of error box for [error], page 37;

set to draw edges for [cone], page 29;

set to draw filled boxes for [boxs], page 40;

reduce text size inside a string (see Section 2.5 [Font styles], page 13);

operation in Section 2.6 [Textual formulas], page 14.

'^' one of marks (see Section 2.3 [Line styles], page 9);

one of mask for face filling (see Section 2.4 [Color scheme], page 11);

style of [subplot], page 24, and [inplot], page 25;

set position of [colorbar], page 32;

set outer position for [legend], page 33;

inverse default position for [axis], page 31;

switch to upper index inside a string (see Section 2.5 [Font styles], page 13);

align center in [bars], page 35, [barh], page 36, [boxplot], page 36, [cones], page 36, [candle], page 37, [ohlc], page 37;

operation in Section 2.6 [Textual formulas], page 14.

'_' empty arrow style (see Section 2.3 [Line styles], page 9);
disable drawing of tick labels for [axis], page 31;
style of [subplot], page 24, and [inplot], page 25;
set position of [colorbar], page 32;
set to draw contours at bottom for [cont], page 41, [contf], page 41, [contd], page 41, [contv], page 42, [tricont], page 50;
switch to lower index inside a string (see Section 2.5 [Font styles], page 13).

'[]' contain symbols excluded from color scheme parsing (see Section 2.4 [Color scheme], page 11).

'{}' contain extended specification of color (see Section 2.2 [Color styles], page 9), dashing (see Section 2.3 [Line styles], page 9) or mask (see Section 2.4 [Color scheme], page 11);
denote special operation in Chapter 1 [MGL scripts], page 1;
denote 'meta-symbol' for LaTeX like string parsing (see Section 2.5 [Font styles], page 13).

'|' line dashing style (see Section 2.3 [Line styles], page 9);
set to use sharp color scheme (see Section 2.4 [Color scheme], page 11);
set to limit width by subplot width for [table], page 38;
delimiter in [list], page 55, command;
operation in Section 2.6 [Textual formulas], page 14.

'\' string continuation symbol on next line for Chapter 1 [MGL scripts], page 1.

'~' disable drawing of tick labels for [axis], page 31, and [colorbar], page 32;
disable first segment in [lamerey], page 39;
one of mask for face filling (see Section 2.4 [Color scheme], page 11).

'0,1,2,3,4,5,6,7,8,9'
line width (see Section 2.3 [Line styles], page 9);
brightness of a color (see Section 2.2 [Color styles], page 9);
precision of numbers in [axis], page 31, [label], page 38, [table], page 38;
kind of smoothing (for digits 1,3,5) in [smooth], page 62;
digits for a value.

'4,6,8' set to draw square, hex- or octo-pyramids instead of cones in [cone], page 29, [cones], page 36.

'A,B,C,D,E,F,a,b,c,d,e,f'
can be hex-digit for color specification if placed inside {} (see Section 2.2 [Color styles], page 9).

'A' arrow style (see Section 2.3 [Line styles], page 9);
set to use absolute position in whole picture for [text], page 30, [colorbar], page 32, [legend], page 33.

'a' set to use absolute position in subplot for [text], page 30;
style of [bars], page 35, [barh], page 36, [cones], page 36.

'B' dark blue color (see Section 2.2 [Color styles], page 9).

'b' blue color (see Section 2.2 [Color styles], page 9);
bold font face if placed after ':' (see Section 2.5 [Font styles], page 13).

'C' dark cyan color (see Section 2.2 [Color styles], page 9);
align text to center if placed after ':' (see Section 2.5 [Font styles], page 13).

'c' cyan color (see Section 2.2 [Color styles], page 9);
name of color axis;
cosine transform for [transform], page 65.

'D' arrow style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11).

'd' one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
start hex-dash description if placed inside {} (see Section 2.3 [Line styles], page 9).

'E' dark green-yellow color (see Section 2.2 [Color styles], page 9).

'e' green-yellow color (see Section 2.2 [Color styles], page 9).

'F'
set LaTeX-like format for numbers in [axis], page 31, [label], page 38, [table], page 38.

'f' style of [bars], page 35, [barh], page 36;
style of [vect], page 47, [vect3], page 47;
set fixed format for numbers in [axis], page 31, [label], page 38, [table], page 38;
Fourier transform for [transform], page 65.

'G' dark green color (see Section 2.2 [Color styles], page 9).

'g' green color (see Section 2.2 [Color styles], page 9).

'H' dark gray color (see Section 2.2 [Color styles], page 9).

'h' gray color (see Section 2.2 [Color styles], page 9);
Hankel transform for [transform], page 65.

'I' arrow style (see Section 2.3 [Line styles], page 9);
set [colorbar], page 32, position near boundary.

'i' line dashing style (see Section 2.3 [Line styles], page 9);
italic font face if placed after ':' (see Section 2.5 [Font styles], page 13).
set to use inverse values for [cloud], page 43, [pipe], page 49, [dew], page 48;
set to fill only area with y1<y<y2 for [region], page 35;
inverse Fourier transform for [transform], page 65.

'j' line dashing style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11).

'K' arrow style (see Section 2.3 [Line styles], page 9).

'k' black color (see Section 2.2 [Color styles], page 9).

'L' dark green-blue color (see Section 2.2 [Color styles], page 9);
align text to left if placed after ':' (see Section 2.5 [Font styles], page 13).

'l' green-blue color (see Section 2.2 [Color styles], page 9).

'M' dark magenta color (see Section 2.2 [Color styles], page 9).

'm' magenta color (see Section 2.2 [Color styles], page 9).

'N' dark sky-blue color (see Section 2.2 [Color styles], page 9).

'n' sky-blue color (see Section 2.2 [Color styles], page 9).

'O' arrow style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11).

'o' one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
over-line text if placed after ':' (see Section 2.5 [Font styles], page 13).

'P' dark purple color (see Section 2.2 [Color styles], page 9).

'p' purple color (see Section 2.2 [Color styles], page 9).

'Q' dark orange or brown color (see Section 2.2 [Color styles], page 9).

'q' orange color (see Section 2.2 [Color styles], page 9).

'R' dark red color (see Section 2.2 [Color styles], page 9);
align text to right if placed after ':' (see Section 2.5 [Font styles], page 13).

'r' red color (see Section 2.2 [Color styles], page 9).

'S' arrow style (see Section 2.3 [Line styles], page 9);
one of mask for face filling (see Section 2.4 [Color scheme], page 11).

's' one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;
one of mask for face filling (see Section 2.4 [Color scheme], page 11);
start hex-mask description if placed inside {} (see Section 2.4 [Color scheme], page 11);
sine transform for [transform], page 65.

't' draw tubes instead of cones in [cone], page 29, [cones], page 36;

'T' arrow style (see Section 2.3 [Line styles], page 9);
place text under the curve for [text], page 30, [cont], page 41, [cont3], page 43.

'`t`' set to draw text labels for [cont], page 41, [cont3], page 43;

name of t-axis (one of ternary axis);

variable in Section 2.6 [Textual formulas], page 14, which usually is varied in range [0,1].

'`U`' dark blue-violet color (see Section 2.2 [Color styles], page 9);

disable rotation of tick labels for [axis], page 31.

'`u`' blue-violet color (see Section 2.2 [Color styles], page 9);

under-line text if placed after '`:`' (see Section 2.5 [Font styles], page 13);

name of u-axis (one of ternary axis);

variable in Section 2.6 [Textual formulas], page 14, which usually denote array itself.

'`V`' arrow style (see Section 2.3 [Line styles], page 9);

place text centering on vertical direction for [text], page 30.

'`v`' one of marks (see Section 2.3 [Line styles], page 9);

set to draw vectors on flow threads for [flow], page 48, and on segments for [lamerey], page 39.

'`W`' bright gray color (see Section 2.2 [Color styles], page 9).

'`w`' white color (see Section 2.2 [Color styles], page 9);

wired text if placed after '`:`' (see Section 2.5 [Font styles], page 13);

name of w-axis (one of ternary axis);

'`X`' arrow style (see Section 2.3 [Line styles], page 9).

'`x`' one of marks (see Section 2.3 [Line styles], page 9) or kind of [error], page 37, boxes;

name of x-axis or x-direction or 1st dimension of a data array;

start hex-color description if placed inside {} (see Section 2.2 [Color styles], page 9);

style of [tape], page 35.

'`Y`' dark yellow or gold color (see Section 2.2 [Color styles], page 9).

'`y`' yellow color (see Section 2.2 [Color styles], page 9);

name of y-axis or y-direction or 2nd dimension of a data array.

'`z`'

name of z-axis or z-direction or 3d dimension of a data array;

style of [tape], page 35.

A.2 Hot-keys for mglview

Key	Description
`Ctrl-P`	Open printer dialog and print graphics.
`Ctrl-W`	Close window.
`Ctrl-T`	Switch on/off transparency for the graphics.
`Ctrl-L`	Switch on/off additional lightning for the graphics.
`Ctrl-Space`	Restore default graphics rotation, zoom and perspective.
`F5`	Execute script and redraw graphics.
`F6`	Change canvas size to fill whole region.
`F7`	Stop drawing and script execution.
`Ctrl-F5`	Run slideshow. If no parameter specified then the dialog with slideshow options will appear.
`Ctrl-Comma`, `Ctrl-Period`	Show next/previous slide. If no parameter specified then the dialog with slideshow options will appear.
`Ctrl-Shift-G`	Copy graphics to clipboard.
`Alt-P`	Export as semitransparent PNG.
`Alt-F`	Export as solid PNG.
`Alt-J`	Export as JPEG.
`Alt-E`	Export as vector EPS.
`Alt-S`	Export as vector SVG.
`Alt-L`	Export as LaTeX/Tikz image.
`Alt-M`	Export as MGLD.
`Alt-D`	Export as PRC/PDF.
`Alt-O`	Export as OBJ.

A.3 Hot-keys for UDAV

Key	Description
`Ctrl-N`	Create new window with empty script. Note, all scripts share variables. So, second window can be used to see some additional information of existed variables.
`Ctrl-O`	Open and execute/show script or data from file. You may switch off automatic exection in UDAV properties
`Ctrl-S`	Save script to a file.
`Ctrl-P`	Open printer dialog and print graphics.
`Ctrl-Z`	Undo changes in script editor.
`Ctrl-Shift-Z`	Redo changes in script editor.
`Ctrl-X`	Cut selected text into clipboard.
`Ctrl-C`	Copy selected text into clipboard.
`Ctrl-V`	Paste selected text from clipboard.
`Ctrl-A`	Select all text in editor.
`Ctrl-F`	Show dialog for text finding.
`F3`	Find next occurrence of the text.
`Win-C` or `Meta-C`	Show dialog for new command and put it into the script.
`Win-F` or `Meta-F`	Insert last fitted formula with found coefficients.
`Win-S` or `Meta-S`	Show dialog for styles and put it into the script. Styles define the plot view (color scheme, marks, dashing and so on).
`Win-O` or `Meta-O`	Show dialog for options and put it into the script. Options are used for additional setup the plot.
`Win-N` or `Meta-N`	Replace selected expression by its numerical value.
`Win-P` or `Meta-P`	Select file and insert its file name into the script.

`Win-G` or `Meta-G`	Show dialog for plot setup and put resulting code into the script. This dialog setup axis, labels, lighting and other general things.
`Ctrl-Shift-O`	Load data from file. Data will be deleted only at exit but UDAV will not ask to save it.
`Ctrl-Shift-S`	Save data to a file.
`Ctrl-Shift-C`	Copy range of numbers to clipboard.
`Ctrl-Shift-V`	Paste range of numbers from clipboard.
`Ctrl-Shift-N`	Recreate the data with new sizes and fill it by zeros.
`Ctrl-Shift-R`	Resize (interpolate) the data to specified sizes.
`Ctrl-Shift-T`	Transform data along dimension(s).
`Ctrl-Shift-M`	Make another data.
`Ctrl-Shift-H`	Find histogram of data.
`Ctrl-T`	Switch on/off transparency for the graphics.
`Ctrl-L`	Switch on/off additional lightning for the graphics.
`Ctrl-G`	Switch on/off grid of absolute coordinates.
`Ctrl-Space`	Restore default graphics rotation, zoom and perspective.
`F5`	Execute script and redraw graphics.
`F6`	Change canvas size to fill whole region.
`F7`	Stop script execution and drawing.
`F8`	Show/hide tool window with list of hidden plots.
`F9`	Restore status for 'once' command and reload data.
`Ctrl-F5`	Run slideshow. If no parameter specified then the dialog with slideshow options will appear.
`Ctrl-Comma`, `Ctrl-Period`	Show next/previous slide. If no parameter specified then the dialog with slideshow options will appear.

`Ctrl-W`	Open dialog with slideshow options.
`Ctrl-Shift-G`	Copy graphics to clipboard.
`F1`	Show help on MGL commands
`F2`	Show/hide tool window with messages and information.
`F4`	Show/hide calculator which evaluate and help to type textual formulas. Textual formulas may contain data variables too.
`Meta-Shift-Up`, `Meta-Shift-Down`	Change view angle θ.
`Meta-Shift-Left`, `Meta-Shift-Right`	Change view angle ϕ.
`Alt-Minus`, `Alt-Equal`	Zoom in/out whole image.
`Alt-Up`, `Alt-Down`, `Alt-Right`, `Alt-Left`	Shift whole image.
`Alt-P`	Export as semitransparent PNG.
`Alt-F`	Export as solid PNG.
`Alt-J`	Export as JPEG.
`Alt-E`	Export as vector EPS.
`Alt-S`	Export as vector SVG.
`Alt-L`	Export as LaTeX/Tikz image.
`Alt-M`	Export as MGLD.
`Alt-D`	Export as PRC/PDF.
`Alt-O`	Export as OBJ.

Appendix B GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

G

H

I

J

L

M

N

O

P

Q

R

S

T

V

W

X

Y

Z