

# MathGL

---

for version 2.3.4

A.A. Balakin (<http://mathgl.sourceforge.net/>)

---

This manual is for MathGL (version 2.3.4), a collection of classes and routines for scientific plotting. Please report any errors in this manual to [mathgl.abalakin@gmail.org](mailto:mathgl.abalakin@gmail.org).

Copyright © 2008-2012 Alexey A. Balakin.

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.”

# Table of Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	What is MathGL?	1
1.2	MathGL features	1
1.3	Installation	2
1.4	Quick guide	3
1.5	Changes from v.1.*	4
1.6	Utilities for parsing MGL	4
1.7	Thanks	5
<b>2</b>	<b>MathGL examples</b>	<b>6</b>
2.1	Basic usage	7
2.1.1	Using MathGL window	7
2.1.2	Drawing to file	9
2.1.3	Animation	10
2.1.4	Drawing in memory	12
2.1.5	Draw and calculate	13
2.1.6	Using QMathGL	15
2.1.7	MathGL and PyQt	16
2.1.8	MathGL and MPI	17
2.2	Advanced usage	19
2.2.1	Subplots	19
2.2.2	Axis and ticks	21
2.2.3	Curvilinear coordinates	25
2.2.4	Colorbars	27
2.2.5	Bounding box	28
2.2.6	Ternary axis	29
2.2.7	Text features	30
2.2.8	Legend sample	33
2.2.9	Cutting sample	34
2.3	Data handling	35
2.3.1	Array creation	35
2.3.2	Linking array	37
2.3.3	Change data	37
2.4	Data plotting	41
2.5	1D samples	44
2.5.1	Plot sample	45
2.5.2	Radar sample	46
2.5.3	Step sample	47
2.5.4	Tens sample	48
2.5.5	Area sample	49
2.5.6	Region sample	50
2.5.7	Stem sample	51
2.5.8	Bars sample	52

2.5.9	Barh sample	53
2.5.10	Cones sample	54
2.5.11	Chart sample	55
2.5.12	BoxPlot sample	56
2.5.13	Candle sample	57
2.5.14	OHLC sample	57
2.5.15	Error sample	58
2.5.16	Mark sample	60
2.5.17	TextMark sample	61
2.5.18	Label sample	62
2.5.19	Table sample	63
2.5.20	Tube sample	64
2.5.21	Tape sample	65
2.5.22	Torus sample	66
2.5.23	Lamerey sample	67
2.5.24	Bifurcation sample	68
2.5.25	Pmap sample	69
2.6	2D samples	70
2.6.1	Surf sample	71
2.6.2	SurfC sample	72
2.6.3	SurfA sample	73
2.6.4	SurfCA sample	74
2.6.5	Mesh sample	75
2.6.6	Fall sample	76
2.6.7	Belt sample	77
2.6.8	Boxs sample	78
2.6.9	Tile sample	79
2.6.10	TileS sample	80
2.6.11	Dens sample	80
2.6.12	Cont sample	81
2.6.13	ContF sample	82
2.6.14	ContD sample	83
2.6.15	ContV sample	84
2.6.16	Axial sample	85
2.6.17	Grad sample	86
2.7	3D samples	87
2.7.1	Surf3 sample	87
2.7.2	Surf3C sample	88
2.7.3	Surf3A sample	89
2.7.4	Surf3CA sample	90
2.7.5	Cloud sample	91
2.7.6	Dens3 sample	92
2.7.7	Cont3 sample	93
2.7.8	ContF3 sample	94
2.7.9	Dens projection sample	95
2.7.10	Cont projection sample	96
2.7.11	ContF projection sample	97
2.7.12	TriPlot and QuadPlot	98



2.7.13	Dots sample .....	99
2.7.14	IFS sample .....	100
2.8	Vector field samples .....	102
2.8.1	Vect sample .....	104
2.8.2	Vect3 sample .....	105
2.8.3	Traj sample .....	106
2.8.4	Flow sample .....	107
2.8.5	Pipe sample .....	108
2.8.6	Dew sample .....	109
2.9	Hints .....	110
2.9.1	“Compound” graphics .....	110
2.9.2	Transparency and lighting .....	112
2.9.3	Types of transparency .....	113
2.9.4	Axis projection .....	115
2.9.5	Adding fog .....	116
2.9.6	Lighting sample .....	117
2.9.7	Using primitives .....	119
2.9.8	STFA sample .....	122
2.9.9	Mapping visualization .....	123
2.9.10	Data interpolation .....	124
2.9.11	Making regular data .....	127
2.9.12	Making histogram .....	128
2.9.13	Nonlinear fitting hints .....	128
2.9.14	PDE solving hints .....	130
2.9.15	Drawing phase plain .....	133
2.9.16	Pulse properties .....	134
2.9.17	Using MGL parser .....	135
2.9.18	Using options .....	137
2.9.19	“Templates” .....	138
2.9.20	Stereo image .....	139
2.9.21	Reduce memory usage .....	140
2.9.22	Scanning file .....	140
2.10	FAQ .....	141
<b>3</b>	<b>General concepts .....</b>	<b>146</b>
3.1	Coordinate axes .....	146
3.2	Color styles .....	147
3.3	Line styles .....	147
3.4	Color scheme .....	149
3.5	Font styles .....	151
3.6	Textual formulas .....	152
3.7	Command options .....	153
3.8	Interfaces .....	154
3.8.1	C/Fortran interface .....	154
3.8.2	C++/Python interface .....	155

<b>4</b>	<b>MathGL core</b>	<b>157</b>
4.1	Create and delete objects	157
4.2	Graphics setup	157
4.2.1	Transparency	158
4.2.2	Lighting	158
4.2.3	Fog	159
4.2.4	Default sizes	160
4.2.5	Cutting	161
4.2.6	Font settings	161
4.2.7	Palette and colors	162
4.2.8	Masks	163
4.2.9	Error handling	163
4.2.10	Stop drawing	165
4.3	Axis settings	166
4.3.1	Ranges (bounding box)	166
4.3.2	Curved coordinates	167
4.3.3	Ticks	168
4.4	Subplots and rotation	172
4.5	Export picture	175
4.5.1	Export to file	176
4.5.2	Frames/Animation	180
4.5.3	Bitmap in memory	181
4.5.4	Parallelization	183
4.6	Background	183
4.7	Primitives	184
4.8	Text printing	188
4.9	Axis and Colorbar	190
4.10	Legend	193
4.11	1D plotting	194
4.12	2D plotting	207
4.13	3D plotting	213
4.14	Dual plotting	217
4.15	Vector fields	222
4.16	Other plotting	227
4.17	Nonlinear fitting	232
4.18	Data manipulation	235
<b>5</b>	<b>Widget classes</b>	<b>237</b>
5.1	mglWnd class	238
5.2	mglDraw class	240
5.3	Fl_MathGL class	241
5.4	QMathGL class	243
5.5	wxMathGL class	248

<b>6</b>	<b>Data processing</b>	<b>251</b>
6.1	Public variables	251
6.2	Data constructor	252
6.3	Data resizing	253
6.4	Data filling	255
6.5	File I/O	261
6.6	Make another data	264
6.7	Data changing	268
6.8	Interpolation	271
6.9	Data information	273
6.10	Operators	275
6.11	Global functions	277
6.12	Evaluate expression	282
6.13	Special data classes	283
<b>7</b>	<b>MGL scripts</b>	<b>285</b>
7.1	MGL definition	285
7.2	Program flow commands	286
7.3	LaTeX package	288
7.4	mglParse class	291
<b>8</b>	<b>UDAV</b>	<b>295</b>
8.1	UDAV overview	295
8.2	UDAV dialogs	297
8.3	UDAV hints	301
<b>9</b>	<b>Other classes</b>	<b>303</b>
9.1	Define new kind of plot (mglBase class)	303
9.2	User defined types (mglDataA class)	309
9.3	mglColor class	311
9.4	mglPoint class	313
	<b>Appendix A Symbols and hot-keys</b>	<b>315</b>
A.1	Symbols for styles	315
A.2	Hot-keys for mglview	321
A.3	Hot-keys for UDAV	322
	<b>Appendix B File formats</b>	<b>326</b>
B.1	Font files	326
B.2	MGLD format	326
B.3	JSON format	327
	<b>Appendix C Plotting time</b>	<b>329</b>

**Appendix D GNU Free Documentation License . . 335**

**Index ..... 342**

# 1 Overview

MathGL is ...

- a library for making high-quality scientific graphics under Linux and Windows;
- a library for the fast data plotting and handling of large data arrays;
- a library for working in window and console modes and for easy embedding into other programs;
- a library with large and growing set of graphics.

## 1.1 What is MathGL?

A code for making high-quality scientific graphics under Linux and Windows. A code for the fast handling and plotting of large data arrays. A code for working in window and console regimes and for easy including into another program. A code with large and renewal set of graphics. Exactly such a code I tried to put in MathGL library.

At this version (2.3.4) MathGL has more than 50 general types of graphics for 1d, 2d and 3d data arrays. It can export graphics to bitmap and vector (EPS or SVG) files. It has OpenGL interface and can be used from console programs. It has functions for data handling and script MGL language for simplification of data plotting. It also has several types of transparency and smoothed lighting, vector fonts and TeX-like symbol parsing, arbitrary curvilinear coordinate system and many other useful things (see pictures section at homepage (<http://mathgl.sf.net/>)). Finally it is platform-independent and free (under GPL v.2.0 or later license).

## 1.2 MathGL features

MathGL can plot a wide range of graphics. It includes:

- one-dimensional (Plot, Area, Bars, Step, Stem, Torus, Chart, Error, Tube, Mark, see Section 4.11 [1D plotting], page 194);
- two-dimensional plots (Mesh, Surf, Dens, Cont, ContF, Boxs, Axial, Fall, Belt, Tile, see Section 4.12 [2D plotting], page 207);
- three-dimensional plots (Surf3, Dens3, Cont3, ContF3, Cloud-like, see Section 4.13 [3D plotting], page 213);
- dual data plots: vector fields Vect, flow threads Flow, mapping chart Map, surfaces and isosurfaces, transparent or colored (i.e. with transparency or color varied) by other data SurfA, SurfC, Surf3A, Surf3C (see Section 4.14 [Dual plotting], page 217);
- and so on. For details see see Chapter 4 [MathGL core], page 157.

In fact, I created the functions for drawing of all the types of scientific plots that I know. The list of plots is growing; if you need some special type of a plot then please email me e-mail and it will appear in the new version.

I tried to make plots as nice looking as possible: e.g., a surface can be transparent and highlighted by several (up to 10) light sources. Most of the drawing functions have 2 variants: simple one for the fast plotting of data, complex one for specifying of the exact position of the plot (including parametric representation). Resulting image can be saved in

bitmap PNG, JPEG, GIF, TGA, BMP format, or in vector EPS, SVG or TeX format, or in 3D formats OBJ, OFF, STL, or in PRC format which can be converted into U3D.

All texts are drawn by vector fonts, which allows for high scalability and portability. Texts may contain commands for: some of the TeX-like symbols, changing index (upper or lower indexes) and the style of font inside the text string (see Section 3.5 [Font styles], page 151). Texts of ticks are rotated with axis rotation. It is possible to create a legend of plot and put text in an arbitrary position on the plot. Arbitrary text encoding (by the help of function `setlocale()`) and UTF-16 encoding are supported.

Special class `mgldata` is used for data encapsulation (see Chapter 6 [Data processing], page 251). In addition to a safe creation and deletion of data arrays it includes functions for data processing (smoothing, differentiating, integrating, interpolating and so on) and reading of data files with automatic size determination. Class `mgldata` can handle arrays with up to three dimensions (arrays which depend on up to 3 independent indexes  $a_{ijk}$ ). Using an array with higher number of dimensions is not meaningful, because I do not know how it can be plotted. Data filling and modification may be done manually or by textual formulas.

There is fast evaluation of a textual mathematical expression (see Section 3.6 [Textual formulas], page 152). It is based on string precompilation to tree-like code at the creation of class instance. At evaluation stage code performs only fast tree-walk and returns the value of the expression. In addition to changing data values, textual formulas are also used for drawing in *arbitrary* curvilinear coordinates. A set of such curvilinear coordinates is limited only by user's imagination rather than a fixed list like: polar, parabolic, spherical, and so on.

## 1.3 Installation

MathGL can be installed in 4 different ways.

1. Compile from sources. The `cmake` build system is used in the library. To run it, one should execute commands: `cmake .` twice, after it `make` and `make install` with root/sudo rights. Sometimes after installation you may need to update the library list – just execute `ldconfig` with root/sudo rights.

There are several additional options which are switched off by default. They are: `enable-fltk`, `enable-glut`, `enable-qt4`, `enable-qt5` for enabling FLTK, GLUT and/or Qt windows; `enable-jpeg`, `enable-gif`, `enable-hdf5` and so on for enabling corresponding file formats; `enable-all` for enabling all additional features. For using `double` as base internal data type use option `enable-double`. For enabling language interfaces use `enable-python`, `enable-octave` or `enable-all-swig` for all languages. You can use WYSIWYG tool (`cmake-gui`) to view all of them, or type `cmake -D enable-all=on -D enable-all-widgets=on -D enable-all-swig=on .` in command line for enabling all features.

There is known bug for building in MinGW – you need to manually add linker option `-fopenmp` (i.e. `CMAKE_EXE_LINKER_FLAGS:STRING='-fopenmp'` and `CMAKE_SHARED_LINKER_FLAGS:STRING='-fopenmp'`) if you enable OpenMP support (i.e. if `enable-openmp=ON`).

2. Use a precompiled binary. There are binaries for MinGW (platform Win32). For a precompiled variant one needs only to unpack the archive to the location of the

compiler (i.e. mathgl/lib in mingw/lib, mathgl/include in mingw/include and so on) or in arbitrary other folder and setup paths in compiler. By default, precompiled versions include the support of GSL ([www.gsl.org](http://www.gsl.org)) and PNG. So, one needs to have these libraries installed on system (it can be found, for example, at <http://gnuwin32.sourceforge.net/packages.html>).

3. Install precompiled versions from standard packages (RPM, deb, DevPak and so on).

Note, you can download the latest sources (which can be not stable) from sourceforge.net SVN by command

```
svn checkout http://svn.code.sf.net/p/mathgl/code/mathgl-2x mathgl-code
```

**IMPORTANT!** MathGL use a set of defines, which were determined at configure stage and may differ if used with non-default compiler (like using MathGL binaries compiled by MinGW in VisualStudio). There are `MGL_SYS_NAN`, `MGL_HAVE_TYPEOF`, `MGL_HAVE_PTHREAD`, `MGL_HAVE_ATTRIBUTE`, `MGL_HAVE_C99_COMPLEX`, `MGL_HAVE_RVAL`.

I specially set them to 0 for Borland and Microsoft compilers due to compatibility reasons. Also default setting are good for GNU (gcc, mingw) and clang compilers. However, for another compiler you may need to manually set this defines to 0 in file `include/mgl2/config.h` if you are using precompiled binaries.

## 1.4 Quick guide

There are 3 steps to prepare the plot in MathGL: (1) prepare data to be plotted, (2) setup plot, (3) plot data. Let me show this on the example of surface plotting.

First we need the data. MathGL use its own class `mglData` to handle data arrays (see Chapter 6 [Data processing], page 251). This class give ability to handle data arrays by more or less format independent way. So, create it

```
int main()
{
    mglData dat(30,40);    // data to for plotting
    for(long i=0;i<30;i++) for(long j=0;j<40;j++)
        dat.a[i+30*j] = 1/(1+(i-15)*(i-15)/225.+(j-20)*(j-20)/400.);
```

Here I create matrix 30\*40 and initialize it by formula. Note, that I use `long` type for indexes *i*, *j* because data arrays can be really large and `long` type will automatically provide proper indexing.

Next step is setup of the plot. The only setup I need is axis rotation and lighting.

```
mglGraph gr;           // class for plot drawing
gr.Rotate(50,60);       // rotate axis
gr.Light(true);         // enable lighting
```

Everything is ready. And surface can be plotted.

```
gr.Surf(dat);           // plot surface
```

Basically plot is done. But I decide to add yellow ('y' color, see Section 3.2 [Color styles], page 147) contour lines on the surface. To do it I can just add:

```
gr.Cont(dat,"y");       // plot yellow contour lines
```

This demonstrate one of base MathGL concept (see, Chapter 3 [General concepts], page 146) – “new drawing never clears things drawn already”. So, you can just consequently call different plotting functions to obtain “combined” plot. For example, if one need to draw axis then he can just call one more plotting function

```
gr.Axis(); // draw axis
```

Now picture is ready and we can save it in a file.

```
gr.WriteFrame("sample.png"); // save it
}
```

To compile your program, you need to specify the linker option `-lmg1`.

This is enough for a compilation of console program or with external (non-MathGL) window library. If you want to use FLTK or Qt windows provided by MathGL then you need to add the option `-lmg1-wnd`.

Fortran users also should add C++ library by the option `-lstdc++`. If library was built with `enable-double=ON` (this default for v.2.1 and later) then all real numbers must be `real*8`. You can make it automatic if use option `-fdefault-real-8`.

## 1.5 Changes from v.1.\*

There are a lot of changes for v.2. Here I denote only main of them.

- `mglGraph` class is single plotter class instead of `mglGraphZB`, `mglGraphPS` and so on.
- Text style and text color positions are swapped. I.e. text style ‘`r:C`’ give red centered text, but not roman dark cyan text as for v.1.\*.
- `ColumnPlot()` indexing is reverted.
- Move most of arguments of plotting functions into the string parameter and/or options.
- “Bright” colors (like {b8}) can be used in color schemes and line styles.
- Intensively use `pthread` internally for parallelization of drawing and data processing.
- Add tick labels rotation and skipping. Add ticks in time/date format.
- New kinds of plots (`Tape()`, `Label()`, `Cones()`, `ContV()`). Extend existing plots. New primitives (`Circle()`, `Ellipse()`, `Rhomb()`, ...). New plot positioning (`MultiPlot()`, `GridPlot()`)
- Improve MGL scripts. Add ‘ask’ command and allow string concatenation from different lines.
- Export to LaTeX and to 3D formats (OBJ, OFF, STL).
- Add pipes support in utilities (`mglconv`, `mglview`).

## 1.6 Utilities for parsing MGL

MathGL library provides several tools for parsing MGL scripts. There is tools saving it to bitmap or vectorial images (`mglconv`). Tool `mglview` show MGL script and allow to rotate and setup the image. Another feature of `mglview` is loading \*.mgld files (see `ExportMGLD()`) for quick viewing 3d pictures.

Both tools have similar set of arguments. They can be name of script file or options. You can use ‘-’ as script name for using standard input (i.e. pipes). Options are:

- `-1 str` set `str` as argument \$1 for script;



- ... ..
- **-9** *str* set *str* as argument \$9 for script;
- **-L** *loc* set locale to *loc*;
- **-s** *fname* set MGL script for setting up the plot;
- **-h** print help message.

Additionally **mg1conv** have following options:

- **-A** *val* add *val* into the list of animation parameters;
- **-C** *v1:v2[:dv]* add values from *v1* ot *v2* with step *dv* (default is 1) into the list of animation parameters;
- **-o** *name* set output file name;
- **-n** disable default output (script should save results by itself);
- **-S** *val* set set scaling factor for [setsize], page 175;
- **-q** *val* set [quality], page 175, for output (*val*=0...9).

Also you can create animated GIF file or a set of JPEG files with names ‘**frameNNNN.jpg**’ (here ‘NNNN’ is frame index). Values of the parameter \$0 for making animation can be specified inside the script by comment **##a val** for each value *val* (one comment for one value) or by option(s) ‘**-A val**’. Also you can specify a cycle for animation by comment **##c v1 v2 dv** or by option **-C v1:v2:dv**. In the case of found/specified animation parameters, tool will execute script several times – once for each value of \$0.

MathGL also provide another simple tool **mg1.cgi** which parse MGL script from CGI request and send back produced PNG file. Usually this program should be placed in **/usr/lib/cgi-bin/**. But you need to put this program by yourself due to possible security issues and difference of Apache server settings.

## 1.7 Thanks

- My special thanks to my wife for the patience during the writing of this library and for the help in documentation writing and spelling.
- I’m thankful to my coauthors D. Kulagin and M. Vidasov for help in developing MathGL.
- I’m thankful to Diego Sejas Viscarra for developing mgltex, contribution to fractal generation and fruitful suggestions.
- I’m thankful to D. Eftaxiopoulos, D. Haley, V. Lipatov and S.M. Plis for making binary packages for Linux.
- I’m thankful to S. Skobeev, C. Mikhailenko, M. Veysman, A. Prokhorov, A. Korotkevich, V. Onuchin, S.M. Plis, R. Kiselev, A. Ivanov, N. Troickiy and V. Lipatov for fruitful comments.
- I’m thankful to sponsors M. Veysman (IHED RAS ([http://jiht.ru/en/about/structure.php?set\\_filter\\_structure=Y & structure\\_UF\\_DEPARTMENT=241 & filter=Y&set\\_filter=Y](http://jiht.ru/en/about/structure.php?set_filter_structure=Y&structure_UF_DEPARTMENT=241&filter=Y&set_filter=Y))) and A. Prokhorov (DATADVANCE ([www.datadvance.net](http://www.datadvance.net))).

Javascript interface was developed with support of DATADVANCE ([www.datadvance.net](http://www.datadvance.net)) company.

## 2 MathGL examples

This chapter contains 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 2.9 [Hints], page 110, section. Also I recommend you to look at Chapter 3 [General concepts], page 146, and Section 2.10 [FAQ], page 141.

Note, that MathGL v.2.\* have only 2 end-user interfaces: one for C/Fortran and similar languages which don't support classes, another one for C++/Python/Octave and similar languages which support classes. So, most of samples placed in this chapter can be run as is (after minor changes due to different syntaxes for different languages). For example, the C++ code

```
#include <mgl2/mgl.h>
int main()
{
    mglGraph gr;
    gr.FPlot("sin(pi*x)");
    gr.WriteFrame("test.png");
}
```

in Python will be as

```
from mathgl import *
gr = mglGraph();
gr.FPlot("sin(pi*x)");
gr.WriteFrame("test.png");
```

in Octave will be as (you need first execute `mathgl`; in newer Octave versions)

```
gr = mglGraph();
gr.FPlot("sin(pi*x)");
gr.WriteFrame("test.png");
```

in C will be as

```
#include <mgl2/mgl_cf.h>
int main()
{
    HMGL gr = mgl_create_graph(600,400);
    mgl_fplot(gr,"sin(pi*x)","","");
    mgl_write_frame(gr,"test.png","");
    mgl_delete_graph(gr);
}
```

in Fortran will be as

```
integer gr, mgl_create_graph
gr = mgl_create_graph(600,400);
call mgl_fplot(gr,'sin(pi*x)', '', '');
call mgl_write_frame(gr,'test.png', '');
call mgl_delete_graph(gr);
```

and so on.

## 2.1 Basic usage

MathGL library can be used by several manners. Each has positive and negative sides:

- *Using of MathGL library features for creating graphical window (requires FLTK, Qt or GLUT libraries).*

Positive side is the possibility to view the plot at once and to modify it (rotate, zoom or switch on transparency or lighting) by hand or by mouse. Negative sides are: the need of X-terminal and limitation consisting in working with the only one set of data at a time.

- *Direct writing to file in bitmap or vector format without creation of graphical window.*

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.

- *Drawing in memory with the following displaying by other graphical program.*

In this case the programmer has more freedom in selecting the window libraries (not only FLTK, Qt or GLUT), in positioning and surroundings control and so on. I recommend to use such way for “stand alone” programs.

- *Using FLTK or Qt widgets provided by MathGL*

Here one can use a set of standard widgets which support export to many file formats, copying to clipboard, handle mouse and so on.

MathGL drawing can be created not only by object oriented languages (like, C++ or Python), but also by pure C or Fortran-like languages. The usage of last one is mostly identical to usage of classes (except the different function names). But there are some differences. C functions must have argument HMGL (for graphics) and/or HMDT (for data arrays) which specifies the object for drawing or manipulating (changing). Fortran users may regard these variables as integer. So, firstly the user has to create this object by function `mgl_create_*`() and has to delete it after the using by function `mgl_delete_*`()

Let me consider the aforesaid in more detail.

### 2.1.1 Using MathGL window

The “interactive” way of drawing in MathGL consists in window creation with help of class `mglQT`, `mglFLTK` or `mglGLUT` (see Chapter 5 [Widget classes], page 237) and the following drawing in this window. There is a corresponding code:

```
#include <mgl2/qt.h>
int sample(mglGraph *gr)
{
    gr->Rotate(60,40);
    gr->Box();
    return 0;
}
```

```

}
//-----
int main(int argc,char **argv)
{
    mglQT gr(sample,"MathGL examples");
    return gr.Run();
}

```

Here callback function `sample` is defined. This function does all drawing. Other function `main` is entry point function for console program. For compilation, just execute the command

```
gcc test.cpp -lmgl-qt -lmgl
```

Alternatively you can create yours own class inherited from Section 5.2 [`mglDraw` class], page 240, and re-implement the function `Draw()` in it:

```

#include <mgl2/qt.h>
class Foo : public mglDraw
{
public:
    int Draw(mglGraph *gr);
};
//-----
int Foo::Draw(mglGraph *gr)
{
    gr->Rotate(60,40);
    gr->Box();
    return 0;
}
//-----
int main(int argc,char **argv)
{
    Foo foo;
    mglQT gr(&foo,"MathGL examples");
    return gr.Run();
}

```

Or use pure C-functions:

```

#include <mgl2/mgl_cf.h>
int sample(HMGL gr, void *)
{
    mgl_rotate(gr,60,40,0);
    mgl_box(gr);
}
int main(int argc,char **argv)
{
    HMGL gr;
    gr = mgl_create_graph_qt(sample,"MathGL examples",0,0);
    return mgl_qt_run();
}

```

```
/* generally I should call mgl_delete_graph() here,
 * but I omit it in main() function. */
}
```

The similar code can be written for mglGLUT window (function `sample()` is the same):

```
#include <mgl2/glut.h>
int main(int argc, char **argv)
{
    mglGLUT gr(sample, "MathGL examples");
    return 0;
}
```

The rotation, shift, zooming, switching on/off transparency and lighting can be done with help of tool-buttons (for `mglQT`, `mglFLTK`) or by hot-keys: ‘a’, ‘d’, ‘w’, ‘s’ for plot rotation, ‘r’ and ‘f’ switching on/off transparency and lighting. Press ‘x’ for exit (or closing the window).

In this example function `sample` rotates axes (`Rotate()`, see Section 4.4 [Subplots and rotation], page 172) and draws the bounding box (`Box()`). Drawing is placed in separate function since it will be used on demand when window canvas needs to be redrawn.

### 2.1.2 Drawing to file

Another way of using MathGL library is the direct writing of the picture to the file. It is most usable for plot creation during long calculation or for using of small programs (like Matlab or Scilab scripts) for visualizing repetitive sets of data. But the speed of drawing is much higher in comparison with a script language.

The following code produces a bitmap PNG picture:

```
#include <mgl2/mgl.h>
int main(int , char **)
{
    mglGraph gr;
    gr.Alpha(true);    gr.Light(true);
    sample(&gr);        // The same drawing function.
    gr.WritePNG("test.png"); // Don't forget to save the result!
    return 0;
}
```

For compilation, you need only `libmgl` library not the one with widgets

```
gcc test.cpp -lmgl
```

This can be important if you create a console program in computer/cluster where X-server (and widgets) is inaccessible.

The only difference from the previous variant (using windows) is manual switching on the transparency `Alpha` and lightning `Light`, if you need it. The usage of frames (see Section 2.1.3 [Animation], page 10) is not advisable since the whole image is prepared each time. If function `sample` contains frames then only last one will be saved to the file. In principle, one does not need to separate drawing functions in case of direct file writing in consequence of the single calling of this function for each picture. However, one may use the same drawing procedure to create a plot with changeable parameters, to export in different

file types, to emphasize the drawing code and so on. So, in future I will put the drawing in the separate function.

The code for export into other formats (for example, into vector EPS file) looks the same:

```
#include <mgl2/mgl.h>
int main(int ,char **)
{
    mglGraph gr;
    gr.Light(true);
    sample(&gr);           // The same drawing function.
    gr.WriteEPS("test.eps"); // Don't forget to save the result!
    return 0;
}
```

The difference from the previous one is using other function `WriteEPS()` for EPS format instead of function `WritePNG()`. Also, there is no switching on of the plot transparency `Alpha` since EPS format does not support it.

### 2.1.3 Animation

Widget classes (`mglWindow`, `mglGLUT`) support a delayed drawing, when all plotting functions are called once at the beginning of writing to memory lists. Further program displays the saved lists faster. Resulting redrawing will be faster but it requires sufficient memory. Several lists (frames) can be displayed one after another (by pressing ‘,’ ‘.’) or run as cinema. To switch these feature on one needs to modify function `sample`:

```
int sample(mglGraph *gr)
{
    gr->NewFrame();           // the first frame
    gr->Rotate(60,40);
    gr->Box();
    gr->EndFrame();           // end of the first frame
    gr->NewFrame();           // the second frame
    gr->Box();
    gr->Axis("xy");
    gr->EndFrame();           // end of the second frame
    return gr->GetNumFrame(); // returns the frame number
}
```

First, the function creates a frame by calling `NewFrame()` for rotated axes and draws the bounding box. The function `EndFrame()` **must be** called after the frame drawing! The second frame contains the bounding box and axes `Axis("xy")` in the initial (unrotated) coordinates. Function `sample` returns the number of created frames `GetNumFrame()`.

Note, that animation can be also done as visualization of running calculations (see Section 2.1.5 [Draw and calculate], page 13).

Pictures with **animation can be saved in file(s)** as well. You can: export in animated GIF, or save each frame in separate file (usually JPEG) and convert these files into the movie (for example, by help of ImageMagic). Let me show both methods.

The simplest method is making animated GIF. There are 3 steps: (1) open GIF file by `StartGIF()` function; (2) create the frames by calling `NewFrame()` before and `EndFrame()` after plotting; (3) close GIF by `CloseGIF()` function. So the simplest code for “running” sinusoid will look like this:

```
#include <mgl2/mgl.h>
int main(int ,char **)
{
    mglGraph gr;
    mglData dat(100);
    char str[32];
    gr.StartGIF("sample.gif");
    for(int i=0;i<40;i++)
    {
        gr.NewFrame();    // start frame
        gr.Box();         // some plotting
        for(int j=0;j<dat.nx;j++)
            dat.a[j]=sin(M_PI*j/dat.nx+M_PI*0.05*i);
        gr.Plot(dat,"b");
        gr.EndFrame();    // end frame
    }
    gr.CloseGIF();
    return 0;
}
```

The second way is saving each frame in separate file (usually JPEG) and later make the movie from them. MathGL have special function for saving frames – it is `WriteFrame()`. This function save each frame with automatic name ‘frame0001.jpg, frame0002.jpg’ and so on. Here prefix ‘frame’ is defined by *PlotId* variable of `mglGraph` class. So the similar code will look like this:

```
#include <mgl2/mgl.h>
int main(int ,char **)
{
    mglGraph gr;
    mglData dat(100);
    char str[32];
    for(int i=0;i<40;i++)
    {
        gr.NewFrame();    // start frame
        gr.Box();         // some plotting
        for(int j=0;j<dat.nx;j++)
            dat.a[j]=sin(M_PI*j/dat.nx+M_PI*0.05*i);
        gr.Plot(dat,"b");
        gr.EndFrame();    // end frame
        gr.WriteFrame();  // save frame
    }
    return 0;
}
```

Created files can be converted to movie by help of a lot of programs. For example, you can use ImageMagic (command ‘`convert frame*.jpg movie.mpg`’), MPEG library, GIMP and so on.

Finally, you can use `mglconv` tool for doing the same with MGL scripts (see Section 1.6 [Utilities], page 4).

### 2.1.4 Drawing in memory

The last way of MathGL using is the drawing in memory. Class `mglGraph` allows one to create a bitmap picture in memory. Further this picture can be displayed in window by some window libraries (like `wxWidgets`, `FLTK`, `Windows GDI` and so on). For example, the code for drawing in `wxWidget` library looks like:

```
void MyForm::OnPaint(wxPaintEvent& event)
{
    int w,h,x,y;
    GetClientSize(&w,&h);    // size of the picture
    mglGraph gr(w,h);

    gr.Alpha(true);          // draws something using MathGL
    gr.Light(true);
    sample(&gr,NULL);

    wxImage img(w,h,gr.GetRGB(),true);
    ToolBar->GetSize(&x,&y);    // gets a height of the toolbar if any
    wxPaintDC dc(this);        // and draws it
    dc.DrawBitmap(wxBitmap(img),0,y);
}
```

The drawing in other libraries is most the same.

For example, `FLTK` code will look like

```
void Fl_MyWidget::draw()
{
    mglGraph gr(w(),h());
    gr.Alpha(true);          // draws something using MathGL
    gr.Light(true);
    sample(&gr,NULL);
    fl_draw_image(gr.GetRGB(), x(), y(), gr.GetWidth(), gr.GetHeight(), 3);
}
```

Qt code will look like

```
void MyWidget::paintEvent(QPaintEvent *)
{
    mglGraph gr(w(),h());

    gr.Alpha(true);          // draws something using MathGL
    gr.Light(true);          gr.Light(0,mglPoint(1,0,-1));
    sample(&gr,NULL);
}
```



```

// Qt don't support RGB format as is. So, let convert it to BGRN.
long w=gr.GetWidth(), h=gr.GetHeight();
unsigned char *buf = new uchar[4*w*h];
gr.GetBGRN(buf, 4*w*h)
QPixmap pic = QPixmap::fromImage(QImage(*buf, w, h, QImage::Format_RGB32));

QPainter paint;
paint.begin(this);  paint.drawPixmap(0,0,pic);  paint.end();
delete []buf;
}

```

### 2.1.5 Draw and calculate

MathGL can be used to draw plots in parallel with some external calculations. The simplest way for this is the usage of Section 5.2 [mglDraw class], page 240. At this you should enable pthread for widgets by setting `enable-pthr-widget=ON` at configure stage (it is set by default). First, you need to inherit your class from `mglDraw` class, define virtual members `Draw()` and `Calc()` which will draw the plot and proceed calculations. You may want to add the pointer `mglWnd *wnd`; to window with plot for interacting with them. Finally, you may add any other data or member functions. The sample class is shown below

```

class myDraw : public mglDraw
{
    mglPoint pnt;    // some variable for changeable data
    long i;          // another variable to be shown
    mglWnd *wnd;     // external window for plotting
public:
    myDraw(mglWnd *w=0) : mglDraw() {      wnd=w;  }
    void SetWnd(mglWnd *w) {      wnd=w;  }
    int Draw(mglGraph *gr)
    {
        gr->Line(mglPoint(),pnt,"Ar2");
        char str[16];  snprintf(str,15,"i=%ld",i);
        gr->Puts(mglPoint(),str);
        return 0;
    }
    void Calc()
    {
        for(i=0;;i++)    // do calculation
        {
            long_calculations();// which can be very long
            Check();       // check if need pause
            pnt.Set(2*mgl_rnd()-1,2*mgl_rnd()-1);
            if(wnd) wnd->Update();
        }
    }
} dr;

```

There is only one issue here. Sometimes you may want to pause calculations to view result carefully, or save state, or change something. So, you need to provide a mechanism for pausing. Class `mglDraw` provide function `Check()`; which check if toolbar button with pause is pressed and wait until it will be released. This function should be called in a "safety" places, where you can pause the calculation (for example, at the end of time step). Also you may add call `exit(0)`; at the end of `Calc()`; function for closing window and exit after finishing calculations. Finally, you need to create a window itself and run calculations.

```
int main(int argc, char **argv)
{
    mglFLTK gr(&dr, "Multi-threading test"); // create window
    dr.SetWnd(&gr); // pass window pointer to yours class
    dr.Run();       // run calculations
    gr.Run();       // run event loop for window
    return 0;
}
```

Note, that you can reach the similar functionality without using `mglDraw` class (i.e. even for pure C code).

```
mglFLTK *gr=NULL; // pointer to window
void *calc(void *) // function with calculations
{
    mglPoint pnt; // some data for plot
    for(long i=0;;i++) // do calculation
    {
        long_calculations(); // which can be very long
        pnt.Set(2*mgl_rnd()-1, 2*mgl_rnd()-1);
        if(gr)
        {
            gr->Clf(); // make new drawing
            // draw something
            gr->Line(mglPoint(), pnt, "Ar2");
            char str[16]; snprintf(str, 15, "i=%ld", i);
            gr->Puts(mglPoint(), str);
            // don't forgot to update window
            gr->Update();
        }
    }
}

int main(int argc, char **argv)
{
    static pthread_t thr;
    pthread_create(&thr, 0, calc, 0); // create separate thread for calculations
    pthread_detach(thr); // and detach it
    gr = new mglFLTK; // now create window
    gr->Run(); // and run event loop
    return 0;
}
```

This sample is exactly the same as one with `mglDraw` class, but it don't have functionality for pausing calculations. If you need it then you have to create global mutex (like `pthread_mutex_t *mutex = pthread_mutex_init(&mutex, NULL);`), set it to window (like `gr->SetMutex(mutex);`) and periodically check it at calculations (like `pthread_mutex_lock(&mutex); pthread_mutex_unlock(&mutex);`).

Finally, you can put the event-handling loop in separate instead of yours code by using `RunThr()` function instead of `Run()` one. Unfortunately, such method work well only for FLTK windows and only if pthread support was enabled. Such limitation come from the Qt requirement to be run in the primary thread only. The sample code will be:

```
int main(int argc, char **argv)
{
    mglFLTK gr("test");
    gr.RunThr();    // <-- need MathGL version which use pthread for widgets
    mglPoint pnt;   // some data
    for(int i=0; i<10; i++)    // do calculation
    {
        long_calculations(); // which can be very long
        pnt.Set(2*mgl_rnd()-1, 2*mgl_rnd()-1);
        gr.Clf();           // make new drawing
        gr.Line(mglPoint(), pnt, "Ar2");
        char str[10] = "i=0";    str[3] = '0'+i;
        gr->Puts(mglPoint(), str);
        gr.Update();           // update window
    }
    return 0;    // finish calculations and close the window
}
```

### 2.1.6 Using QMathGL

MathGL have several interface widgets for different widget libraries. There are `QMathGL` for Qt, `Fl_MathGL` for FLTK. These classes provide control which display MathGL graphics. Unfortunately there is no uniform interface for widget classes because all libraries have slightly different set of functions, features and so on. However the usage of MathGL widgets is rather simple. Let me show it on the example of `QMathGL`.

First of all you have to define the drawing function or inherit a class from `mglDraw` class. After it just create a window and setup `QMathGL` instance as any other Qt widget:

```
#include <QApplication>
#include <QMainWindow>
#include <QScrollArea>
#include <mgl2/qmathgl.h>
int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    QMainWindow *Wnd = new QMainWindow;
    Wnd->resize(810, 610); // for fill up the QMGL, menu and toolbars
    Wnd->setWindowTitle("QMathGL sample");
    // here I allow to scroll QMathGL -- the case
```

```

// then user want to prepare huge picture
QScrollArea *scroll = new QScrollArea(Wnd);

// Create and setup QMathGL
QMathGL *QMGL = new QMathGL(Wnd);
//QMGL->setPopup(popup); // if you want to setup popup menu for QMGL
QMGL->setDraw(sample);
// or use QMGL->setDraw(foo); for instance of class Foo:public mglDraw
QMGL->update();

// continue other setup (menu, toolbar and so on)
scroll->setWidget(QMGL);
Wnd->setCentralWidget(scroll);
Wnd->show();
return a.exec();
}

```

### 2.1.7 MathGL and PyQt

Generally SWIG based classes (including the Python one) are the same as C++ classes. However, there are few tips for using MathGL with PyQt. Below I place a very simple python code which demonstrate how MathGL can be used with PyQt. This code is mostly written by Prof. Dr. Heino Falcke. You can just copy it to a file `mgl-pyqt-test.py` and execute it from python shell by command `execfile("mgl-pyqt-test.py")`

```

from PyQt4 import QtGui,QtCore
from mathgl import *
import sys
app = QtGui.QApplication(sys.argv)
qpointf=QtCore.QPointF()

class hfQtPlot(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.img=(QtGui.QImage())
    def setgraph(self,gr):
        self.buffer='\t'
        self.buffer=self.buffer.expandtabs(4*gr.GetWidth()*gr.GetHeight())
        gr.GetBGRN(self.buffer,len(self.buffer))
        self.img=QtGui.QImage(self.buffer, gr.GetWidth(),gr.GetHeight(),QtGui.QImage.Format_
        self.update()
    def paintEvent(self, event):
        paint = QtGui.QPainter()
        paint.begin(self)
        paint.drawImage(qpointf,self.img)
        paint.end()

BackgroundColor=[1.0,1.0,1.0]

```

```

size=100
gr=mglGraph()
y=mglData(size)
#y.Modify("((0.7*cos(2*pi*(x+.2)*500)+0.3)*(rnd*0.5+0.5)+362.135+10000.)")
y.Modify("(cos(2*pi*x*10)+1.1)*1000.*rnd-501")
x=mglData(size)
x.Modify("x^2");

def plotpanel(gr,x,y,n):
    gr.SubPlot(2,2,n)
    gr.SetXRange(x)
    gr.SetYRange(y)
    gr.AdjustTicks()
    gr.Axis()
    gr.Box()
    gr.Label("x","x-Axis",1)
    gr.Label("y","y-Axis",1)
    gr.ClearLegend()
    gr.AddLegend("Legend: "+str(n),"k")
    gr.Legend()
    gr.Plot(x,y)

gr.Clf(BackgroundColor[0],BackgroundColor[1],BackgroundColor[2])
gr.SetPlotFactor(1.5)
plotpanel(gr,x,y,0)
y.Modify("(cos(2*pi*x*10)+1.1)*1000.*rnd-501")
plotpanel(gr,x,y,1)
y.Modify("(cos(2*pi*x*10)+1.1)*1000.*rnd-501")
plotpanel(gr,x,y,2)
y.Modify("(cos(2*pi*x*10)+1.1)*1000.*rnd-501")
plotpanel(gr,x,y,3)

gr.WritePNG("test.png","Test Plot")

qw = hfQtPlot()
qw.show()
qw.setgraph(gr)
qw.raise_()

```

### 2.1.8 MathGL and MPI

For using MathGL in MPI program you just need to: (1) plot its own part of data for each running node; (2) collect resulting graphical information in a single program (for example, at node with rank=0); (3) save it. The sample code below demonstrate this for very simple sample of surface drawing.

First you need to initialize MPI

```

#include <stdio.h>
#include <mgl2/mpi.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    // initialize MPI
    int rank=0, numproc=1;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==0) printf("Use %d processes.\n", numproc);

    Next step is data creation. For simplicity, I create data arrays with the same sizes for
    all nodes. At this, you have to create mglGraph object too.

    // initialize data similarly for all nodes
    mglData a(128,256);
    mglGraphMPI gr;

    Now, data should be filled by numbers. In real case, it should be some kind of calcula-
    tions. But I just fill it by formula.

    // do the same plot for its own range
    char buf[64];
    sprintf(buf,"xrange %g %g",2.*rank/numproc-1,2.*(rank+1)/numproc-1);
    gr.Fill(a,"sin(2*pi*x)",buf);

    It is time to plot the data. Don't forget to set proper axis range(s) by using parametric
    form or by using options (as in the sample).

    // plot data in each node
    gr.Clf(); // clear image before making the image
    gr.Rotate(40,60);
    gr.Surf(a,"",buf);

    Finally, let send graphical information to node with rank=0.

    // collect information
    if(rank!=0) gr.MPI_Send(0);
    else for(int i=1;i<numproc;i++) gr.MPI_Recv(i);

    Now, node with rank=0 have whole image. It is time to save the image to a file. Also,
    you can add a kind of annotations here – I draw axis and bounding box in the sample.

    if(rank==0)
    {
        gr.Box(); gr.Axis(); // some post processing
        gr.WritePNG("test.png"); // save result
    }

    In my case the program is done, and I finalize MPI. In real program, you can repeat the
    loop of data calculation and data plotting as many times as you need.

    MPI_Finalize();
    return 0;

```

```
}
```

You can type `'mpic++ test.cpp -lmgl-mpi -lmgl && mpirun -np 8 ./a.out'` for compilation and running the sample program on 8 nodes. Note, that you have to set `enable-mpi=ON` at MathGL configure to use this feature.

## 2.2 Advanced usage

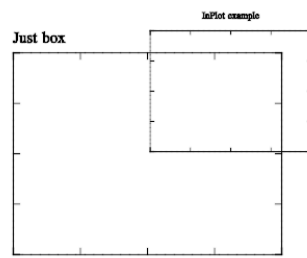
Now I show several non-obvious features of MathGL: several subplots in a single picture, curvilinear coordinates, text printing and so on. Generally you may miss this section at first reading.

### 2.2.1 Subplots

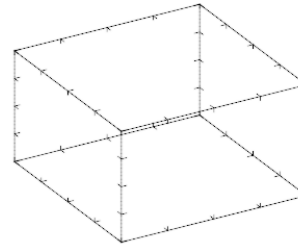
Let me demonstrate possibilities of plot positioning and rotation. MathGL has a set of functions: `[subplot]`, page 172, `[inplot]`, page 173, `[title]`, page 173, `[aspect]`, page 174, and `[rotate]`, page 174, and so on (see Section 4.4 [Subplots and rotation], page 172). The order of their calling is strictly determined. First, one changes the position of plot in image area (functions `[subplot]`, page 172, `[inplot]`, page 173, and `[multiplot]`, page 172). Secondly, you can add the title of plot by `[title]`, page 173, function. After that one may rotate the plot (function `[rotate]`, page 174). Finally, one may change aspects of axes (function `[aspect]`, page 174). The following code illustrates the aforesaid it:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0); gr->Box();
    gr->Puts(mglPoint(-1,1.1),"Just box",":L");
    gr->InPlot(0.2,0.5,0.7,1,false); gr->Box();
    gr->Puts(mglPoint(0,1.2),"InPlot example");
    gr->SubPlot(2,2,1); gr->Title("Rotate only");
    gr->Rotate(50,60); gr->Box();
    gr->SubPlot(2,2,2); gr->Title("Rotate and Aspect");
    gr->Rotate(50,60); gr->Aspect(1,1,2); gr->Box();
    gr->SubPlot(2,2,3); gr->Title("Aspect in other direction");
    gr->Rotate(50,60); gr->Aspect(1,2,2); gr->Box();
    return 0;
}
```

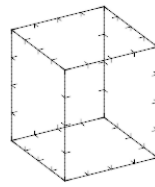
Here I used function `Puts` for printing the text in arbitrary position of picture (see Section 4.8 [Text printing], page 188). Text coordinates and size are connected with axes. However, text coordinates may be everywhere, including the outside the bounding box. I'll show its features later in Section 2.2.7 [Text features], page 30.



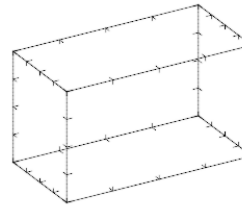
Rotate only



Rotate and Aspect



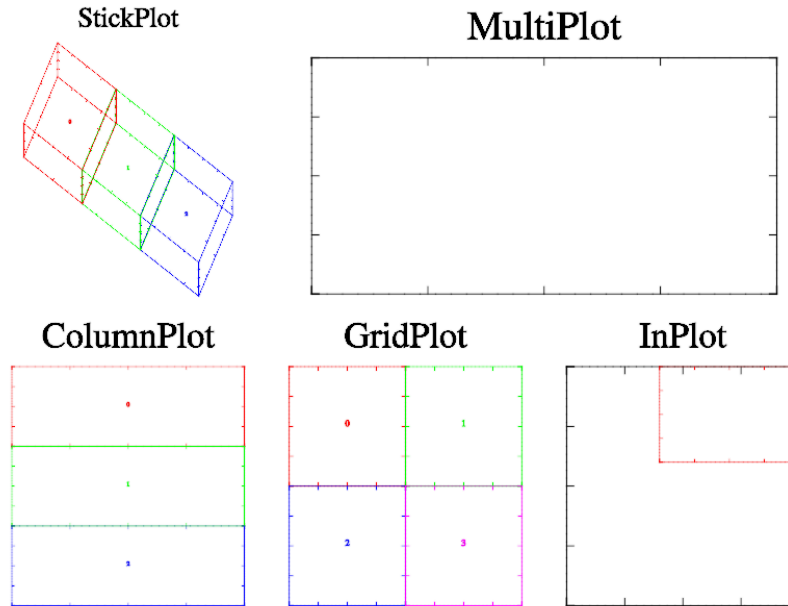
Aspect in other direction



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

```
int sample(mglGraph *gr)
{
    gr->SubPlot(3,2,0); gr->Title("StickPlot");
    gr->StickPlot(3, 0, 20, 30); gr->Box("r"); gr->Puts(mglPoint(0),"0","r");
    gr->StickPlot(3, 1, 20, 30); gr->Box("g"); gr->Puts(mglPoint(0),"1","g");
    gr->StickPlot(3, 2, 20, 30); gr->Box("b"); gr->Puts(mglPoint(0),"2","b");
    gr->SubPlot(3,2,3,""); gr->Title("ColumnPlot");
    gr->ColumnPlot(3, 0); gr->Box("r"); gr->Puts(mglPoint(0),"0","r");
    gr->ColumnPlot(3, 1); gr->Box("g"); gr->Puts(mglPoint(0),"1","g");
    gr->ColumnPlot(3, 2); gr->Box("b"); gr->Puts(mglPoint(0),"2","b");
    gr->SubPlot(3,2,4,""); gr->Title("GridPlot");
    gr->GridPlot(2, 2, 0); gr->Box("r"); gr->Puts(mglPoint(0),"0","r");
    gr->GridPlot(2, 2, 1); gr->Box("g"); gr->Puts(mglPoint(0),"1","g");
    gr->GridPlot(2, 2, 2); gr->Box("b"); gr->Puts(mglPoint(0),"2","b");
    gr->GridPlot(2, 2, 3); gr->Box("m"); gr->Puts(mglPoint(0),"3","m");
    gr->SubPlot(3,2,5,""); gr->Title("InPlot"); gr->Box();
    gr->InPlot(0.4, 1, 0.6, 1, true); gr->Box("r");
    gr->MultiPlot(3,2,1, 2, 1,""); gr->Title("MultiPlot"); gr->Box();
    return 0;
}
```





### 2.2.2 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 4.3.1 [Ranges (bounding box)], page 166). Ticks on axis are specified by function `SetTicks`, `SetTicksVal`, `SetTicksTime` (see Section 4.3.3 [Ticks], page 168). But usually

Function `[axis]`, page 190, 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). Function `[grid]`, page 192, draws grid perpendicularly to specified directions. Example of axes and grid drawing is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0); gr->Title("Axis origin, Grid"); gr->SetOrigin(0,0);
    gr->Axis(); gr->Grid(); gr->FPlot("x^3");

    gr->SubPlot(2,2,1); gr->Title("2 axis");
    gr->SetRanges(-1,1,-1,1); gr->SetOrigin(-1,-1,-1); // first axis
    gr->Axis(); gr->Label('y',"axis 1",0); gr->FPlot("sin(pi*x)");
    gr->SetRanges(0,1,0,1); gr->SetOrigin(1,1,1); // second axis
    gr->Axis(); gr->Label('y',"axis 2",0); gr->FPlot("cos(pi*x)");

    gr->SubPlot(2,2,3); gr->Title("More axis");
    gr->SetOrigin(NAN,NAN); gr->SetRange('x',-1,1);
    gr->Axis(); gr->Label('x',"x",0); gr->Label('y',"y_1",0);
    gr->FPlot("x^2","k");
    gr->SetRanges(-1,1,-1,1); gr->SetOrigin(-1.3,-1); // second axis
    gr->Axis("y","r"); gr->Label('y',"#r{y_2}",0.2);
    gr->FPlot("x^3","r");
}
```

```

gr->SubPlot(2,2,2); gr->Title("4 segments, inverted axis");
gr->SetOrigin(0,0);
gr->InPlot(0.5,1,0.5,1); gr->SetRanges(0,10,0,2); gr->Axis();
gr->FPlot("sqrt(x/2)"); gr->Label('x',"W",1); gr->Label('y',"U",1);
gr->InPlot(0,0.5,0.5,1); gr->SetRanges(1,0,0,2); gr->Axis("x");
gr->FPlot("sqrt(x)+x^3"); gr->Label('x',"\\tau",-1);
gr->InPlot(0.5,1,0,0.5); gr->SetRanges(0,10,4,0); gr->Axis("y");
gr->FPlot("x/4"); gr->Label('y',"L",-1);
gr->InPlot(0,0.5,0,0.5); gr->SetRanges(1,0,4,0); gr->FPlot("4*x^2");
return 0;
}

```

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).

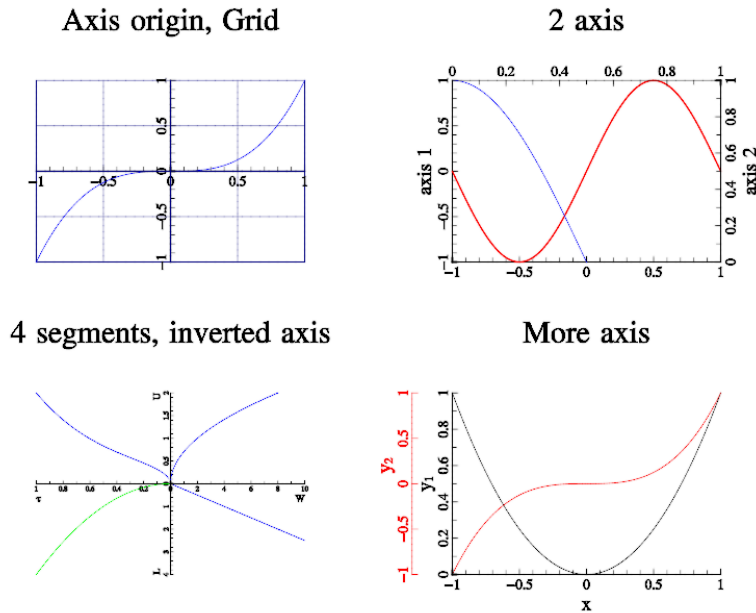
At this inverted axis can be created by 2 methods. First one is used in this sample – just specify minimal axis value to be large than maximal one. This method work well for 2D axis, but can wrongly place labels in 3D case. Second method is more general and work in 3D case too – just use [aspect], page 174, 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
gr->SetRanges(0,10,4,0); gr->Axis();

// variant 2
gr->SetRanges(0,10,0,4); gr->Aspect(1,-1); gr->Axis();

```



Another MathGL feature is fine ticks tuning. 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.

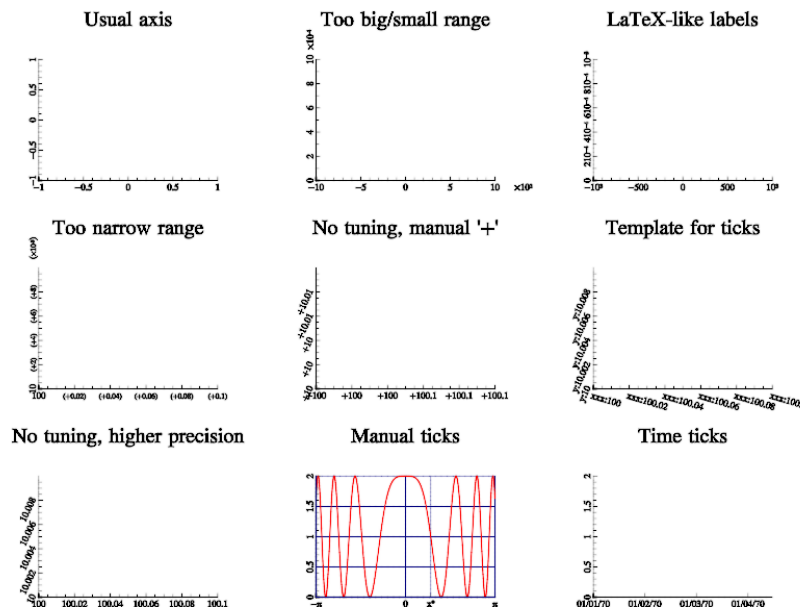
The sample code, demonstrated ticks feature is

```
int sample(mglGraph *gr)
{
    gr->SubPlot(3,3,0); gr->Title("Usual axis"); gr->Axis();
    gr->SubPlot(3,3,1); gr->Title("Too big/small range");
    gr->SetRanges(-1000,1000,0,0.001); gr->Axis();
    gr->SubPlot(3,3,2); gr->Title("LaTeX-like labels");
    gr->Axis("F!");
    gr->SubPlot(3,3,3); gr->Title("Too narrow range");
    gr->SetRanges(100,100.1,10,10.01); gr->Axis();
    gr->SubPlot(3,3,4); gr->Title("No tuning, manual '+'");
    // for version<2.3 you need first call gr->SetTuneTicks(0);
    gr->Axis("+!");
    gr->SubPlot(3,3,5); gr->Title("Template for ticks");
    gr->SetTickTempl('x',"xxx:%g"); gr->SetTickTempl('y',"y:%g");
    gr->Axis();
    // now switch it off for other plots
```

```

gr->SetTickTempl('x',""); gr->SetTickTempl('y',"");
gr->SubPlot(3,3,6); gr->Title("No tuning, higher precision");
gr->Axis("!4");
gr->SubPlot(3,3,7); gr->Title("Manual ticks"); gr->SetRanges(-M_PI,M_PI, 0, 2);
gr->SetTicks('x',M_PI,0,0,"\\pi"); gr->AddTick('x',0.886,"x^*");
// alternatively you can use following lines
//double val[]={-M_PI, -M_PI/2, 0, 0.886, M_PI/2, M_PI};
//gr->SetTicksVal('x', mglData(6,val), "-\\pi\\n-\\pi/2\\n0\\nx^*\\n\\pi/2\\n\\pi");
gr->Axis(); gr->Grid(); gr->FPlot("2*cos(x^2)^2", "r2");
gr->SubPlot(3,3,8); gr->Title("Time ticks"); gr->SetRange('x',0,3e5);
gr->SetTicksTime('x',0); gr->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 2.2.3 [Curvilinear coordinates], page 25) by help of `SetFunc` or `SetCoor` functions. At this one should vary about proper axis range. So the code looks as following:

```

int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0,"<_"); gr->Title("Semi-log axis");
    gr->SetRanges(0.01,100,-1,1); gr->SetFunc("lg(x)","");
    gr->Axis(); gr->Grid("xy","g"); gr->FPlot("sin(1/x)");
    gr->Label('x',"x",0); gr->Label('y',"y = sin 1/x",0);

    gr->SubPlot(2,2,1,"<_"); gr->Title("Log-log axis");
    gr->SetRanges(0.01,100,0.1,100); gr->SetFunc("lg(x)","lg(y)");
    gr->Axis(); gr->Grid("!","h="); gr->Grid();
}

```

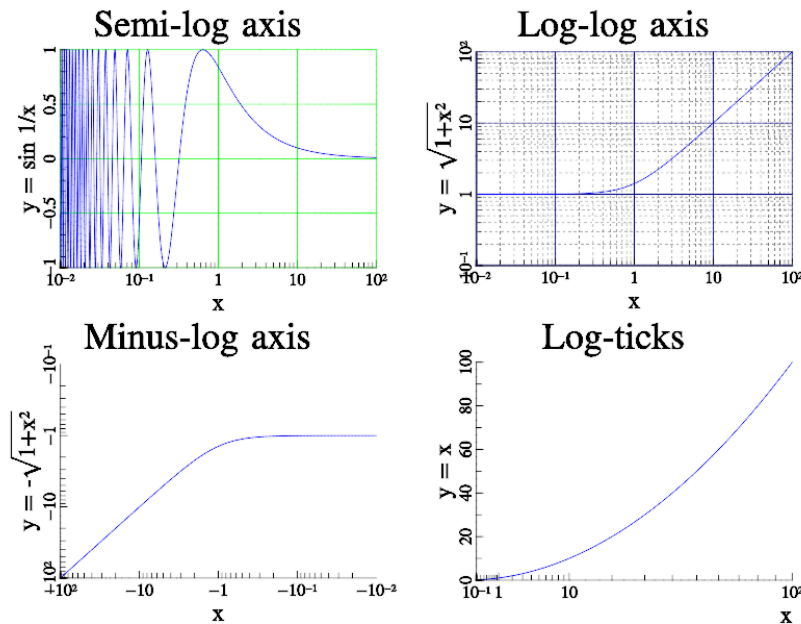
```

gr->FPlot("sqrt(1+x^2)"); gr->Label('x',"x",0);
gr->Label('y', "y = \sqrt{1+x^2}",0);

gr->SubPlot(2,2,2,"<_"); gr->Title("Minus-log axis");
gr->SetRanges(-100,-0.01,-100,-0.1); gr->SetFunc("-lg(-x)","-lg(-y)");
gr->Axis(); gr->FPlot("-sqrt(1+x^2)");
gr->Label('x',"x",0); gr->Label('y', "y = -\sqrt{1+x^2}",0);

gr->SubPlot(2,2,3,"<_"); gr->Title("Log-ticks");
gr->SetRanges(0.1,100,0,100); gr->SetFunc("sqrt(x)","");
gr->Axis(); gr->FPlot("x");
gr->Label('x',"x",1); gr->Label('y', "y = x",0);
return 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).

### 2.2.3 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:

```
int sample(mglGraph *gr)
```

```

{
  gr->SetOrigin(-1,1,-1);

  gr->SubPlot(2,2,0); gr->Title("Cartesian"); gr->Rotate(50,60);
  gr->FPlot("2*t-1","0.5","0","r2");
  gr->Axis(); gr->Grid();

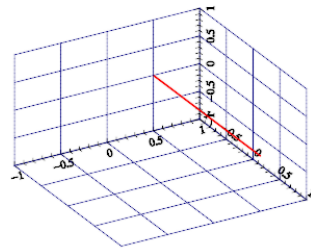
  gr->SetFunc("y*sin(pi*x)","y*cos(pi*x)",0);
  gr->SubPlot(2,2,1); gr->Title("Cylindrical"); gr->Rotate(50,60);
  gr->FPlot("2*t-1","0.5","0","r2");
  gr->Axis(); gr->Grid();

  gr->SetFunc("2*y*x","y*y - x*x",0);
  gr->SubPlot(2,2,2); gr->Title("Parabolic"); gr->Rotate(50,60);
  gr->FPlot("2*t-1","0.5","0","r2");
  gr->Axis(); gr->Grid();

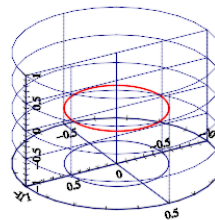
  gr->SetFunc("y*sin(pi*x)","y*cos(pi*x)","x+z");
  gr->SubPlot(2,2,3); gr->Title("Spiral"); gr->Rotate(50,60);
  gr->FPlot("2*t-1","0.5","0","r2");
  gr->Axis(); gr->Grid();
  gr->SetFunc(0,0,0); // set to default Cartesian
  return 0;
}

```

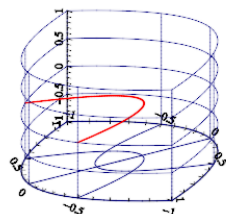
Cartesian



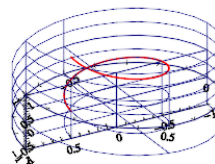
Cylindrical



Parabolic



Spiral



## 2.2.4 Colorbars

MathGL handle [colorbar], page 191, 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:

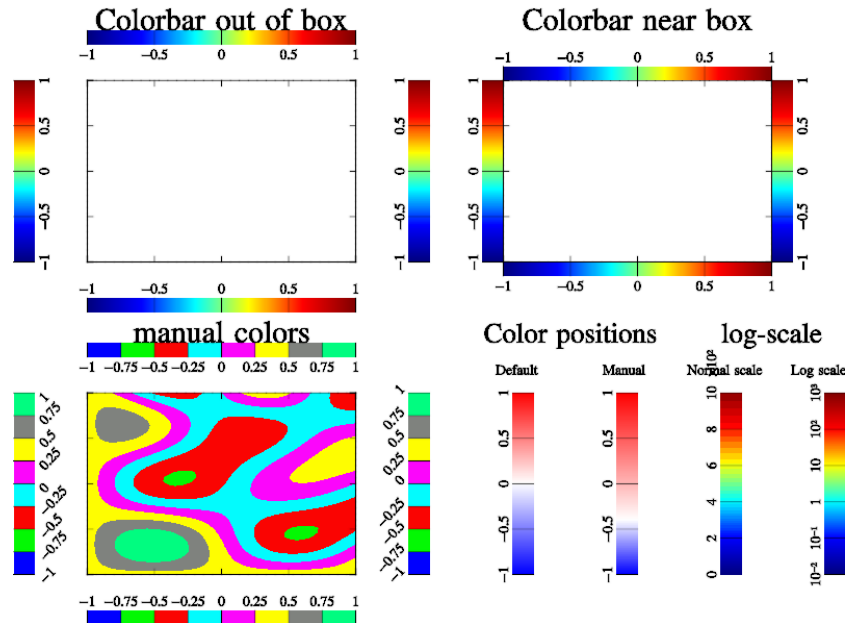
```
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0); gr->Title("Colorbar out of box"); gr->Box();
    gr->Colorbar("<"); gr->Colorbar(">");
    gr->Colorbar("_"); gr->Colorbar("^");

    gr->SubPlot(2,2,1); gr->Title("Colorbar near box"); gr->Box();
    gr->Colorbar("<I"); gr->Colorbar(">I");
    gr->Colorbar("_I"); gr->Colorbar("^I");

    gr->SubPlot(2,2,2); gr->Title("manual colors");
    mglData a,v; mgl_prepare2d(&a,0,&v);
    gr->Box(); gr->ContD(v,a);
    gr->Colorbar(v,"<"); gr->Colorbar(v,">");
    gr->Colorbar(v,"_"); gr->Colorbar(v,"^");

    gr->SubPlot(2,2,3); gr->Title(" ");
    gr->Puts(mglPoint(-0.5,1.55),"Color positions",":C",-2);
    gr->Colorbar("bwr>",0.25,0); gr->Puts(mglPoint(-0.9,1.2),"Default");
    gr->Colorbar("b{w,0.3}r>",0.5,0); gr->Puts(mglPoint(-0.1,1.2),"Manual");

    gr->Puts(mglPoint(1,1.55),"log-scale",":C",-2);
    gr->SetRange('c',0.01,1e3);
    gr->Colorbar(">",0.75,0); gr->Puts(mglPoint(0.65,1.2),"Normal scale");
    gr->SetFunc("", "", "", "lg(c)");
    gr->Colorbar(">"); gr->Puts(mglPoint(1.35,1.2),"Log scale");
    return 0;
}
```

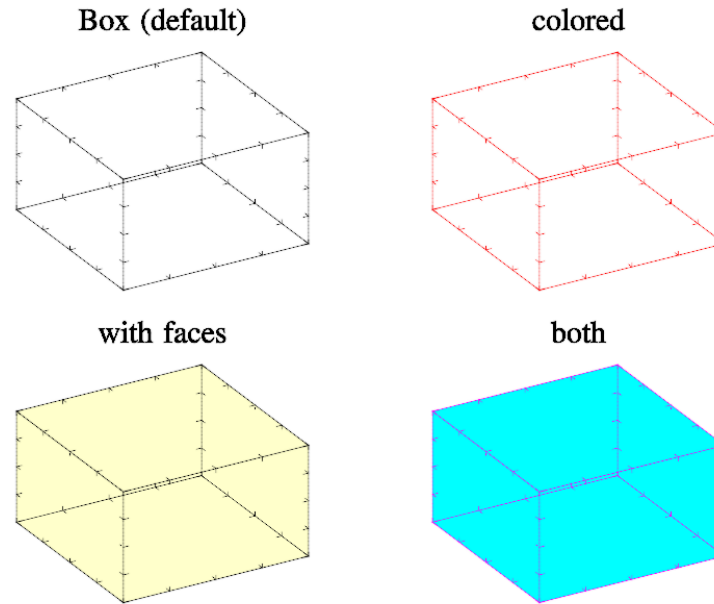


## 2.2.5 Bounding box

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 192, function. By default, it draw black or white box with ticks (color depend on transparency type, see Section 2.9.3 [Types of transparency], page 113). 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 192, features is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0); gr->Title("Box (default)"); gr->Rotate(50,60);
    gr->Box();
    gr->SubPlot(2,2,1); gr->Title("colored"); gr->Rotate(50,60);
    gr->Box("r");
    gr->SubPlot(2,2,2); gr->Title("with faces"); gr->Rotate(50,60);
    gr->Box("@");
    gr->SubPlot(2,2,3); gr->Title("both"); gr->Rotate(50,60);
    gr->Box("@cm");
    return 0;
}
```





## 2.2.6 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 168, function for switching to ternary/quaternary coordinates. The sample code is:

```
int sample(mglGraph *gr)
{
    gr->SetRanges(0,1,0,1,0,1);
    mglData x(50),y(50),z(50),rx(10),ry(10), a(20,30);
    a.Modify("30*x*y*(1-x-y)^2*(x+y<1)");
    x.Modify("0.25*(1+cos(2*pi*x))");
    y.Modify("0.25*(1+sin(2*pi*x))");
    rx.Modify("rnd"); ry.Modify("(1-v)*rnd",rx);
    z.Modify("x");

    gr->SubPlot(2,2,0); gr->Title("Ordinary axis 3D");
    gr->Rotate(50,60); gr->Light(true);
    gr->Plot(x,y,z,"r2"); gr->Surf(a,"BbcyrR#");
    gr->Axis(); gr->Grid(); gr->Box();
    gr->Label('x',"B",1); gr->Label('y',"C",1); gr->Label('z',"Z",1);

    gr->SubPlot(2,2,1); gr->Title("Ternary axis (x+y+t=1)");
    gr->Ternary(1);
```

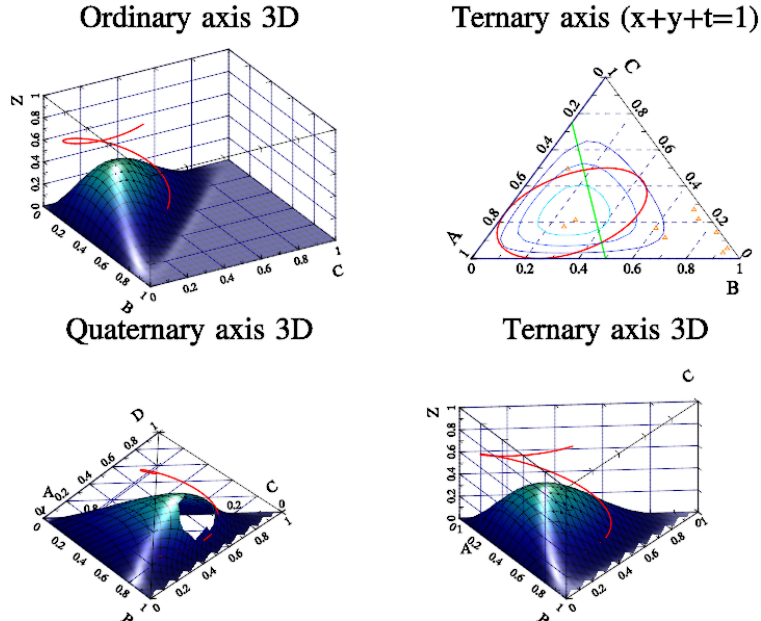
```

gr->Plot(x,y,"r2"); gr->Plot(rx,ry,"q^ "); gr->Cont(a,"BbcyrR");
gr->Line(mglPoint(0.5,0), mglPoint(0,0.75), "g2");
gr->Axis(); gr->Grid("xyz","B;");
gr->Label('x',"B"); gr->Label('y',"C"); gr->Label('t',"A");

gr->SubPlot(2,2,2); gr->Title("Quaternary axis 3D");
gr->Rotate(50,60); gr->Light(true);
gr->Ternary(2);
gr->Plot(x,y,z,"r2"); gr->Surf(a,"BbcyrR#");
gr->Axis(); gr->Grid(); gr->Box();
gr->Label('t',"A",1); gr->Label('x',"B",1);
gr->Label('y',"C",1); gr->Label('z',"D",1);

gr->SubPlot(2,2,3); gr->Title("Ternary axis 3D");
gr->Rotate(50,60); gr->Light(true);
gr->Ternary(1);
gr->Plot(x,y,z,"r2"); gr->Surf(a,"BbcyrR#");
gr->Axis(); gr->Grid(); gr->Box();
gr->Label('t',"A",1); gr->Label('x',"B",1);
gr->Label('y',"C",1); gr->Label('z',"Z",1);
return 0;
}

```



### 2.2.7 Text features

MathGL prints text by vector font. There are functions for manual specifying of text position (like `Put`s) 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 4.2.6 [Font settings], page 161). 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 3.5 [Font styles], page 151).

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’.

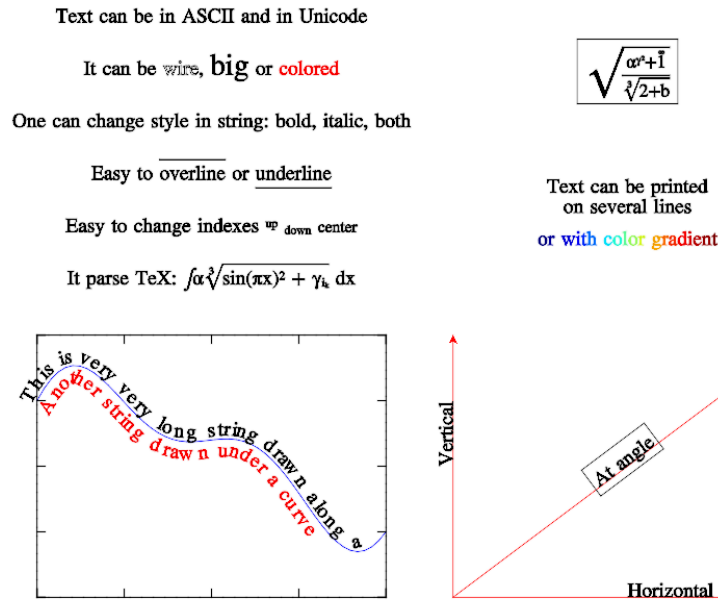
Example of MathGL font drawing is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0,"");
    gr->Putsw(mglPoint(0,1),L"Text can be in ASCII and in Unicode");
    gr->Puts(mglPoint(0,0.6),"It can be \\wire{wire}, \\big{big} or #r{colored}");
    gr->Puts(mglPoint(0,0.2),"One can change style in string: "
        "\\b{bold}, \\i{italic}, \\b{both}");
    gr->Puts(mglPoint(0,-0.2),"Easy to \\a{overline} or "
        "\\u{underline}");
    gr->Puts(mglPoint(0,-0.6),"Easy to change indexes ^{up} _{down} @{center}");
    gr->Puts(mglPoint(0,-1),"It parse TeX: \\int \\alpha \\cdot "
        "\\sqrt{3\\sin(\\pi x)^2 + \\gamma_{i_k}} dx");

    gr->SubPlot(2,2,1,"");
    gr->Puts(mglPoint(0,0.5), "\\sqrt{\\frac{\\alpha^{\\gamma^2}+\\overset{1}{\\big\\infty}}{\\big\\infty}}");
    gr->Puts(mglPoint(0,-0.5),"Text can be printed\non several lines");

    gr->SubPlot(2,2,2,"");
    mglData y; mgl_prepare1d(&y);
    gr->Box(); gr->Plot(y.SubData(-1,0));
    gr->Text(y,"This is very very long string drawn along a curve",":k");
    gr->Text(y,"Another string drawn under a curve","T:r");

    gr->SubPlot(2,2,3,"");
    gr->Line(mglPoint(-1,-1),mglPoint(1,-1),"rA");
    gr->Puts(mglPoint(0,-1),mglPoint(1,-1),"Horizontal");
    gr->Line(mglPoint(-1,-1),mglPoint(1,1),"rA");
    gr->Puts(mglPoint(0,0),mglPoint(1,1),"At angle","@");
    gr->Line(mglPoint(-1,-1),mglPoint(-1,1),"rA");
    gr->Puts(mglPoint(-1,0),mglPoint(-1,1),"Vertical");
    return 0;
}
```



You can change font faces by loading font files by function [loadfont], page 161. 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](http://sourceforge.net/project/showfiles.php?group_id=152187&package_id=267177)). The sample code is:

```
int sample(mglGraph *gr)
{
    double h=1.1, d=0.25;
    gr->LoadFont("STIX");      gr->Puts(mglPoint(0,h), "default font (STIX)");
    gr->LoadFont("adventor");  gr->Puts(mglPoint(0,h-d), "adventor font");
    gr->LoadFont("bonum");     gr->Puts(mglPoint(0,h-2*d), "bonum font");
    gr->LoadFont("chorus");    gr->Puts(mglPoint(0,h-3*d), "chorus font");
    gr->LoadFont("cursor");    gr->Puts(mglPoint(0,h-4*d), "cursor font");
    gr->LoadFont("heros");     gr->Puts(mglPoint(0,h-5*d), "heros font");
    gr->LoadFont("heroscn");   gr->Puts(mglPoint(0,h-6*d), "heroscn font");
    gr->LoadFont("pagella");   gr->Puts(mglPoint(0,h-7*d), "pagella font");
    gr->LoadFont("schola");    gr->Puts(mglPoint(0,h-8*d), "schola font");
    gr->LoadFont("termes");    gr->Puts(mglPoint(0,h-9*d), "termes font");
    return 0;
}
```

default font (STIX)

adventor font

bonum font

chorus font

cursor font

heros font

heroscn font

pagella font

schola font

termes font

### 2.2.8 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 193, function; or use ‘legend’ option (see Section 3.7 [Command options], page 153), 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 194, 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.

Function [legend], page 193, 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:

```
int sample(mglGraph *gr)
{
    gr->AddLegend("sin(\\pi {x^2})", "b");
    gr->AddLegend("sin(\\pi x)", "g*");
    gr->AddLegend("sin(\\pi \\sqrt{x})", "rd");
    gr->AddLegend("just text", " ");
    gr->AddLegend("no indent for this", "");

    gr->SubPlot(2,2,0,""); gr->Title("Legend (default)");
    gr->Box(); gr->Legend();

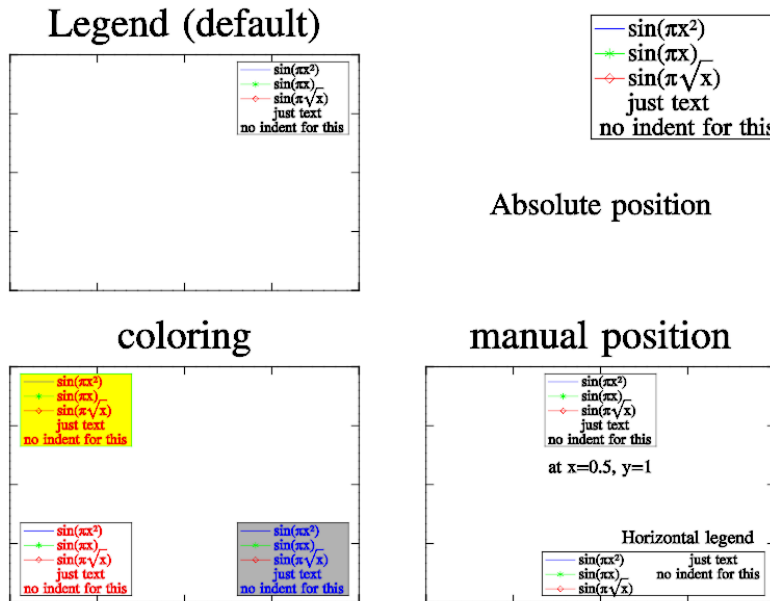
    gr->Legend(3, "A#");
    gr->Puts(mglPoint(0.75,0.65), "Absolute position", "A");
}
```

```

gr->SubPlot(2,2,2,""); gr->Title("coloring"); gr->Box();
gr->Legend(0,"r#"); gr->Legend(1,"Wb#"); gr->Legend(2,"ygr#");

gr->SubPlot(2,2,3,""); gr->Title("manual position"); gr->Box();
gr->Legend(0.5,1); gr->Puts(mglPoint(0.5,0.55),"at x=0.5, y=1","a");
gr->Legend(1,"#-"); gr->Puts(mglPoint(0.75,0.25),"Horizontal legend","a");
return 0;
}

```



### 2.2.9 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:

```

int sample(mglGraph *gr)
{
    mglData a,c,v(1); mglS_prepare2d(&a); mglS_prepare3d(&c); v.a[0]=0.5;
    gr->SubPlot(2,2,0); gr->Title("Cut on (default)");
    gr->Rotate(50,60); gr->Light(true);
}

```

```

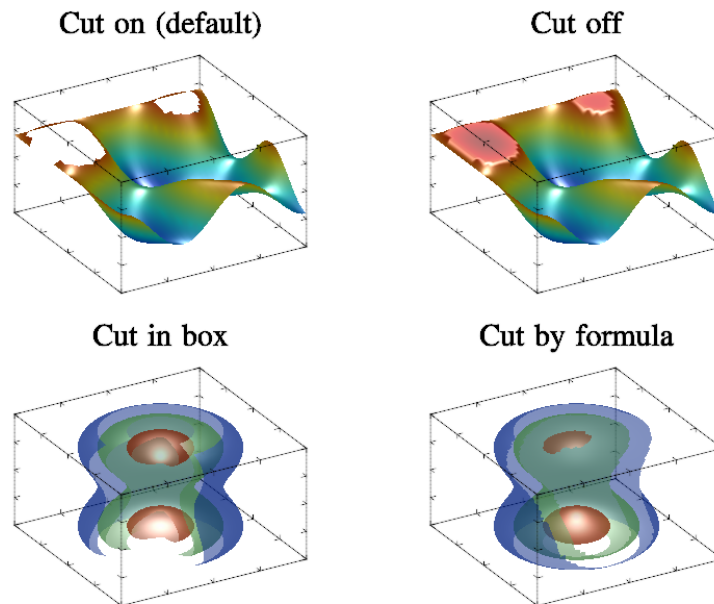
gr->Box(); gr->Surf(a,"","zrange -1 0.5");

gr->SubPlot(2,2,1); gr->Title("Cut off"); gr->Rotate(50,60);
gr->Box(); gr->Surf(a,"","zrange -1 0.5; cut off");

gr->SubPlot(2,2,2); gr->Title("Cut in box"); gr->Rotate(50,60);
gr->SetCutBox(mglPoint(0,-1,-1), mglPoint(1,0,1.1));
gr->Alpha(true); gr->Box(); gr->Surf3(c);
gr->SetCutBox(mglPoint(0), mglPoint(0)); // switch it off

gr->SubPlot(2,2,3); gr->Title("Cut by formula"); gr->Rotate(50,60);
gr->CutOff("(z>(x+0.5*y-1)^2-1) & (z>(x-0.5*y-1)^2-1)");
gr->Box(); gr->Surf3(c); gr->CutOff(""); // switch it off
return 0;
}

```



## 2.3 Data handling

Class `mglData` contains all functions for the data handling in MathGL (see Chapter 6 [Data processing], page 251). 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.

### 2.3.1 Array creation

There are many ways in MathGL how data arrays can be created and filled.

One can put the data in `mglData` instance by several ways. Let us do it for sinus function:

- one can create external array, fill it and put to `mglData` variable

```
double *a = new double[50];
for(int i=0;i<50;i++)    a[i] = sin(M_PI*i/49.);
```

```
mglData y;
y.Set(a,50);
```

- another way is to create `mglData` instance of the desired size and then to work directly with data in this variable

```
mglData y(50);
for(int i=0;i<50;i++)    y.a[i] = sin(M_PI*i/49.);
```

- next way is to fill the data in `mglData` instance by textual formula with the help of `Modify()` function

```
mglData y(50);
y.Modify("sin(pi*x)");
```

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

```
mglData y(50);
y.Fill(0,M_PI);
y.Modify("sin(u)");
```

- finally it can be loaded from file

```
FILE *fp=fopen("sin.dat","wt");    // create file first
for(int i=0;i<50;i++)    fprintf(fp,"%g\n",sin(M_PI*i/49.));
fclose(fp);
```

```
mglData y("sin.dat");              // load it
```

At this you can use textual or HDF files, as well as import values from bitmap image (PNG is supported right now).

- at this one can read only part of data

```
FILE *fp=fopen("sin.dat","wt");    // create large file first
for(int i=0;i<70;i++)    fprintf(fp,"%g\n",sin(M_PI*i/49.));
fclose(fp);
```

```
mglData y;
y.Read("sin.dat",50);              // load it
```

Creation of 2d- and 3d-arrays is mostly the same. But one should keep in mind that class `mglData` uses flat data representation. For example, matrix  $30 \times 40$  is presented as flat (1d-) array with length  $30 \times 40 = 1200$  ( $n_x=30$ ,  $n_y=40$ ). The element with indexes  $\{i,j\}$  is  $a[i+n_x*j]$ . So for 2d array we have:

```
mglData z(30,40);
for(int i=0;i<30;i++)    for(int j=0;j<40;j++)
    z.a[i+30*j] = sin(M_PI*i/29.)*sin(M_PI*j/39.);
```

or by using `Modify()` function

```
mglData z(30,40);
z.Modify("sin(pi*x)*cos(pi*y)");
```

The only non-obvious thing here is using multidimensional arrays in C/C++, i.e. arrays defined like `mreal dat[40][30];`. Since, formally these elements `dat[i]` can address the



memory in arbitrary place you should use the proper function to convert such arrays to `mglData` object. For C++ this is functions like `mglData::Set(mreal **dat, int N1, int N2);`. For C this is functions like `mgl_data_set_mreal2(HMDT d, const mreal **dat, int N1, int N2);`. At this, you should keep in mind that `nx=N2` and `ny=N1` after conversion.

### 2.3.2 Linking array

Sometimes the data arrays are so large, that one couldn't copy its values to another array (i.e. into `mglData`). In this case, he can define its own class derived from `mglDataA` (see Section 9.2 [`mglDataA` class], page 309) or can use `Link` function.

In last case, MathGL just save the link to an external data array, but not copy it. You should provide the existence of this data array for whole time during which MathGL can use it. Another point is that MathGL will automatically create new array if you'll try to modify data values by any of `mglData` functions. So, you should use only function with `const` modifier if you want still using link to the original data array.

Creating the link is rather simple – just the same as using `Set` function

```
double *a = new double[50];
for(int i=0;i<50;i++)    a[i] = sin(M_PI*i/49.);

mglData y;
y.Link(a,50);
```

### 2.3.3 Change data

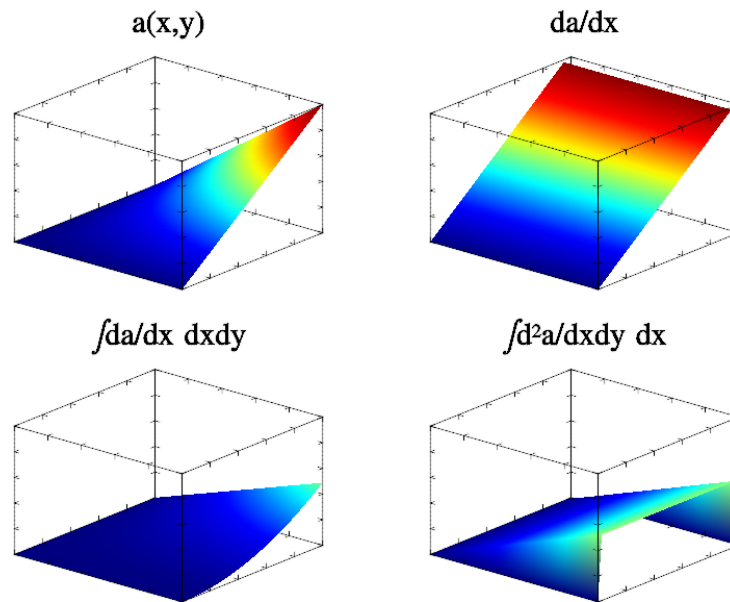
MathGL has functions for data processing: differentiating, integrating, smoothing and so on (for more detail, see Chapter 6 [Data processing], page 251). Let us consider some examples. The simplest ones are integration and differentiation. The direction in which operation will be performed is specified 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 `Integral("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:

```
int sample(mglGraph *gr)
{
    gr->SetRanges(0,1,0,1,0,1);
    mglData a(30,40); a.Modify("x*y");
    gr->SubPlot(2,2,0); gr->Rotate(60,40);
    gr->Surf(a);    gr->Box();
    gr->Puts(mglPoint(0.7,1,1.2),"a(x,y)");
    gr->SubPlot(2,2,1); gr->Rotate(60,40);
    a.Diff("x");    gr->Surf(a); gr->Box();
    gr->Puts(mglPoint(0.7,1,1.2),"da/dx");
    gr->SubPlot(2,2,2); gr->Rotate(60,40);
    a.Integral("xy"); gr->Surf(a); gr->Box();
    gr->Puts(mglPoint(0.7,1,1.2),"\\int da/dx dxdy");
    gr->SubPlot(2,2,3); gr->Rotate(60,40);
    a.Diff2("y"); gr->Surf(a); gr->Box();
    gr->Puts(mglPoint(0.7,1,1.2),"\\int {d^2}a/dxdy dx");
```

```

    return 0;
}

```



Data smoothing (function [smooth], page 271) 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:

```

int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0,""); gr->Title("Envelop sample");
    mglData d1(1000); gr->Fill(d1,"exp(-8*x^2)*sin(10*pi*x)");
    gr->Axis(); gr->Plot(d1, "b");
    d1.Envelop('x'); gr->Plot(d1, "r");

    gr->SubPlot(2,2,1,""); gr->Title("Smooth sample");
    mglData y0(30),y1,y2,y3;
    gr->SetRanges(0,1,0,1);
    gr->Fill(y0, "0.4*sin(pi*x) + 0.3*cos(1.5*pi*x) - 0.4*sin(2*pi*x)+0.5*rand");

    y1=y0; y1.Smooth("x3");
    y2=y0; y2.Smooth("x5");
    y3=y0; y3.Smooth("x");

    gr->Plot(y0,"{m7}:s", "legend 'none'"); //gr->AddLegend("none","k");
}

```

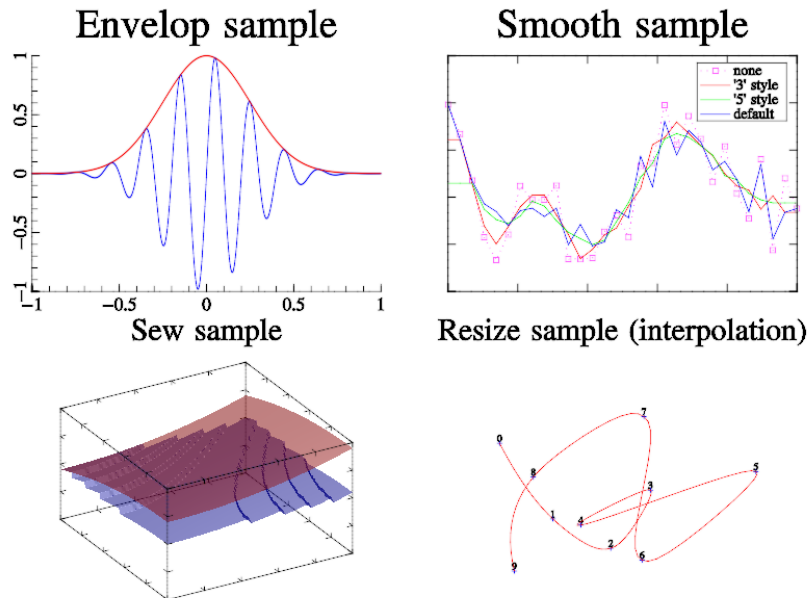
```

gr->Plot(y1,"r", "legend ''3' style'");
gr->Plot(y2,"g", "legend ''5' style'");
gr->Plot(y3,"b", "legend 'default'");
gr->Legend();   gr->Box();

gr->SubPlot(2,2,2);   gr->Title("Sew sample");
mglData d2(100, 100); gr->Fill(d2, "mod((y^2-(1-x)^2)/2,0.1)");
gr->Rotate(50, 60);   gr->Light(true);   gr->Alpha(true);
gr->Box();             gr->Surf(d2, "b");
d2.Sew("xy", 0.1);   gr->Surf(d2, "r");

gr->SubPlot(2,2,3);   gr->Title("Resize sample (interpolation)");
mglData x0(10), v0(10), x1, v1;
gr->Fill(x0,"rnd");   gr->Fill(v0,"rnd");
x1 = x0.Resize(100); v1 = v0.Resize(100);
gr->Plot(x0,v0,"b+ "); gr->Plot(x1,v1,"r-");
gr->Label(x0,v0,"%n");
return 0;
}

```



Also one can create new data arrays on base of the existing one: extract slice, row or column of data ([subdata], page 264), summarize along a direction(s) ([sum], page 267), find distribution of data elements ([hist], page 266) 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 6.8 [Interpolation], page 271). Also there is a function [evaluate], page 265, 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 265, which find array of indexes at which data array is equal to given value (i.e. work as root finding). But [solve], page 265, 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 265, function will return index only in given direction, assuming that other index(es) are the same as equidistant index(es) of original data. If data have multiple roots then second (and later) branches can be found by consecutive call(s) of [solve], page 265, function. Let me demonstrate this on the following sample.

```
int sample(mglGraph *gr)
{
    gr->SetRange('z',0,1);
    mglData x(20,30), y(20,30), z(20,30), xx,yy,zz;
    gr->Fill(x,"(x+2)/3*cos(pi*y)");
    gr->Fill(y,"(x+2)/3*sin(pi*y)");
    gr->Fill(z,"exp(-6*x^2-2*sin(pi*y)^2)");

    gr->SubPlot(2,1,0); gr->Title("Cartesian space");    gr->Rotate(30,-40);
    gr->Axis("xyzU");    gr->Box();    gr->Label('x',"x"); gr->Label('y',"y");
    gr->SetOrigin(1,1); gr->Grid("xy");
    gr->Mesh(x,y,z);

    // section along 'x' direction
    mglData u = x.Solve(0.5,'x');
    mglData v(u.nx);    v.Fill(0,1);
    xx = x.Evaluate(u,v);    yy = y.Evaluate(u,v);    zz = z.Evaluate(u,v);
    gr->Plot(xx,yy,zz,"k2o");

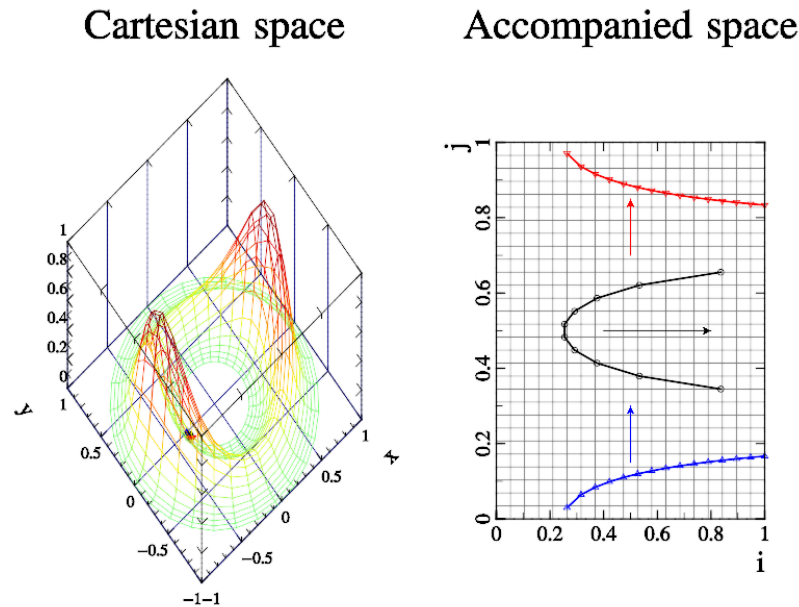
    // 1st section along 'y' direction
    mglData u1 = x.Solve(-0.5,'y');
    mglData v1(u1.nx);    v1.Fill(0,1);
    xx = x.Evaluate(v1,u1); yy = y.Evaluate(v1,u1); zz = z.Evaluate(v1,u1);
    gr->Plot(xx,yy,zz,"b2^");

    // 2nd section along 'y' direction
    mglData u2 = x.Solve(-0.5,'y',u1);
    xx = x.Evaluate(v1,u2); yy = y.Evaluate(v1,u2); zz = z.Evaluate(v1,u2);
    gr->Plot(xx,yy,zz,"r2v");

    gr->SubPlot(2,1,1); gr->Title("Accompanied space");
    gr->SetRanges(0,1,0,1); gr->SetOrigin(0,0);
    gr->Axis(); gr->Box();    gr->Label('x',"i"); gr->Label('y',"j");
    gr->Grid(z,"h");

    gr->Plot(u,v,"k2o");    gr->Line(mglPoint(0.4,0.5),mglPoint(0.8,0.5),"kA");
    gr->Plot(v1,u1,"b2^");    gr->Line(mglPoint(0.5,0.15),mglPoint(0.5,0.3),"bA");
    gr->Plot(v1,u2,"r2v");    gr->Line(mglPoint(0.5,0.7),mglPoint(0.5,0.85),"rA");
```

```
}
```



## 2.4 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 another textual argument for options (see Section 3.7 [Command options], page 153).

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  $\{x(i), y(i), z(i)\}$ ,  $i=1\dots n$ . The textual argument allow you specify styles of line and marks (see Section 3.3 [Line styles], page 147). If this parameter is NULL or empty then solid line with color from palette is used (see Section 4.2.7 [Palette and colors], page 162).

Below I shall show the features of 1D plotting on base of [plot], page 194, function. Let us start from sinus plot:

```
int sample(mglGraph *gr)
{
    mglData y0(50);          y0.Modify("sin(pi*(2*x-1))");
    gr->SubPlot(2,2,0);
    gr->Plot(y0);             gr->Box();
}
```

Style of line is not specified in [plot], page 194, function. So MathGL uses the solid line with first color of palette (this is blue). Next subplot shows array *y1* with 2 rows:

```
gr->SubPlot(2,2,1);
mglData y1(50,2);
y1.Modify("sin(pi*2*x-pi)");
y1.Modify("cos(pi*2*x-pi)/2",1);
gr->Plot(y1);          gr->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:

```
mglData x(50);          x.Modify("cos(pi*2*x-pi)");
gr->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 194, and [subdata], page 264, functions. Let us draw ellipse by orange dash line:

```
gr->Plot(y1.SubData(-1,0),y1.SubData(-1,1),"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):

```
gr->SubPlot(2,2,2);    gr->Rotate(60,40);
mglData z(50);        z.Modify("2*x-1");
gr->Plot(x,y0,z);      gr->Box();
```

Functions [plot], page 194, and [subdata], page 264, make 3D curve plot but for single array. Use it to put circle marks on the previous plot:

```
mglData y2(10,3);     y2.Modify("cos(pi*(2*x-1+y))");
y2.Modify("2*x-1",2);
gr->Plot(y2.SubData(-1,0),y2.SubData(-1,1),y2.SubData(-1,2),"bo ");
```

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

```
gr->SubPlot(2,2,3);    gr->Rotate(60,40);
gr->Bars(x,y0,z,"r");  gr->Box();
return 0;
```

```
}
```

Surfaces [surf], page 207, and other 2D plots (see Section 4.12 [2D plotting], page 207) are drawn 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 3.4 [Color scheme], page 149). 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

```
int sample(mglGraph *gr)
{
    gr->Light(true);      gr->Light(0,mglPoint(0,0,1));
```

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

```
mglData a0(50,40);
a0.Modify("0.6*sin(2*pi*x)*sin(3*pi*y)+0.4*cos(3*pi*(x*y))");
gr->SubPlot(2,2,0);   gr->Rotate(60,40);
gr->Surf(a0);          gr->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:

```
mglData x(50,40),y(50,40),z(50,40);
x.Modify("0.8*sin(2*pi*x)*sin(pi*y)");
y.Modify("0.8*cos(2*pi*x)*sin(pi*y)");
z.Modify("0.8*cos(pi*y)");
gr->SubPlot(2,2,1);   gr->Rotate(60,40);
gr->Surf(x,y,z,"BbwrR");gr->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  $nz=3$ :

```
mglData a1(50,40,3);
a1.Modify("0.6*sin(2*pi*x)*sin(3*pi*y)+0.4*cos(3*pi*(x*y))");
a1.Modify("0.6*cos(2*pi*x)*cos(3*pi*y)+0.4*sin(3*pi*(x*y))",1);
a1.Modify("0.6*cos(2*pi*x)*cos(3*pi*y)+0.4*cos(3*pi*(x*y))",2);
gr->SubPlot(2,2,2);   gr->Rotate(60,40);
gr->Alpha(true);
gr->Surf(a1);          gr->Box();
```

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

```
gr->SubPlot(2,2,3);   gr->Rotate(60,40);
gr->Dens(a1);          gr->Box();
return 0;
}
```

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 192, or [mesh], page 207, for plots in plain or in volume). However, for isosurfaces (including surfaces of rotation [axial], page 212) this flag switches the face drawing off and figure becomes wired. The following code gives example of flag '#' using (compare with normal function drawing as in its description):

```
int sample(mglGraph *gr)
{
    gr->Alpha(true);      gr->Light(true);      gr->Light(0,mglPoint(0,0,1));
    mglData a(30,20);
    a.Modify("0.6*sin(2*pi*x)*sin(3*pi*y) + 0.4*cos(3*pi*(x*y))");

    gr->SubPlot(2,2,0);   gr->Rotate(40,60);
    gr->Surf(a,"BbcyrR#"); gr->Box();
}
```

```

gr->SubPlot(2,2,1);   gr->Rotate(40,60);
gr->Dens(a,"BbcyrR#");           gr->Box();
gr->SubPlot(2,2,2);   gr->Rotate(40,60);
gr->Cont(a,"BbcyrR#");           gr->Box();
gr->SubPlot(2,2,3);   gr->Rotate(40,60);
gr->Axial(a,"BbcyrR#");           gr->Box();
return 0;
}

```

## 2.5 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  $\{x(i),y(i),z(i)\}$ ,  $i=1\dots n$ . Most of samples will use the same data for plotting. So, I put its initialization in separate function

```

void mgls_prepare1d(mglData *y, mglData *y1=0, mglData *y2=0, mglData *x1=0, mglData *x2=0)
{
    register long i,n=50;
    if(y) y->Create(n,3);
    if(x1) x1->Create(n);   if(x2) x2->Create(n);
    if(y1) y1->Create(n);   if(y2) y2->Create(n);
    mreal xx;
    for(i=0;i<n;i++)
    {
        xx = i/(n-1.);
        if(y)
        {
            y->a[i] = 0.7*sin(2*M_PI*xx) + 0.5*cos(3*M_PI*xx) + 0.2*sin(M_PI*xx);
            y->a[i+n] = sin(2*M_PI*xx);
            y->a[i+2*n] = cos(2*M_PI*xx);
        }
        if(y1) y1->a[i] = 0.5+0.3*cos(2*M_PI*xx);
        if(y2) y2->a[i] = 0.3*sin(2*M_PI*xx);
        if(x1) x1->a[i] = xx*2-1;
        if(x2) x2->a[i] = 0.05+0.03*cos(2*M_PI*xx);
    }
}

```

or using C functions

```

void mgls_prepare1d(HMDT y, HMDT y1=0, HMDT y2=0, HMDT x1=0, HMDT x2=0)
{
    register long i,n=50;
    if(y) mgl_data_create(y,n,3,1);
    if(x1) mgl_data_create(x1,n,1,1);
    if(x2) mgl_data_create(x2,n,1,1);
    if(y1) mgl_data_create(y1,n,1,1);
    if(y2) mgl_data_create(y2,n,1,1);
    mreal xx;
    for(i=0;i<n;i++)

```



```

{
    xx = i/(n-1.);
    if(y)
    {
        mgl_data_set_value(y, 0.7*sin(2*M_PI*xx) + 0.5*cos(3*M_PI*xx) + 0.2*sin(M_PI*xx), i,0);
        mgl_data_set_value(y, sin(2*M_PI*xx), i,1,0);
        mgl_data_set_value(y, cos(2*M_PI*xx), i,2,0);
    }
    if(y1) mgl_data_set_value(y1, 0.5+0.3*cos(2*M_PI*xx), i,0,0);
    if(y2) mgl_data_set_value(y2, 0.3*sin(2*M_PI*xx), i,0,0);
    if(x1) mgl_data_set_value(x1, xx*2-1, i,0,0);
    if(x2) mgl_data_set_value(x2, 0.05+0.03*cos(2*M_PI*xx), i,0,0);
}
}

```

### 2.5.1 Plot sample

Function [plot], page 194, 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:

```

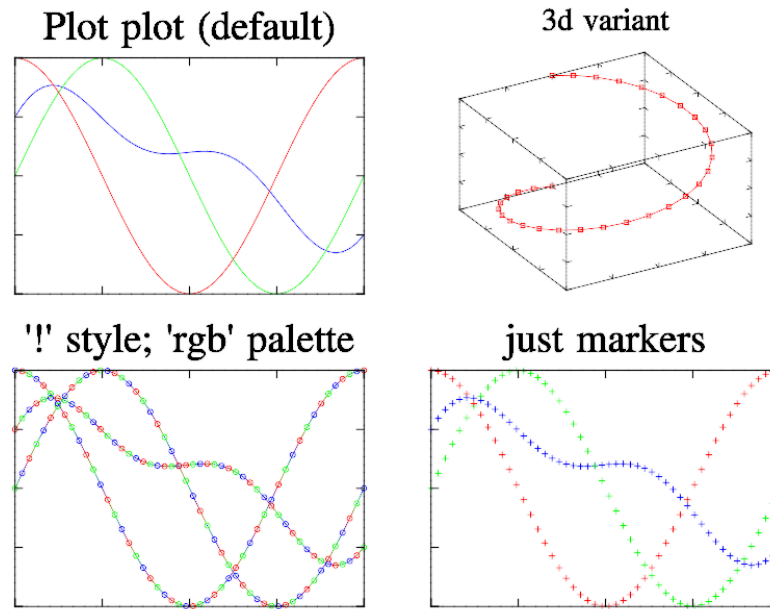
int sample(mglGraph *gr)
{
    mglData y; mgl_prepare1d(&y); gr->SetOrigin(0,0,0);
    gr->SubPlot(2,2,0,""); gr->Title("Plot plot (default)");
    gr->Box(); gr->Plot(y);

    gr->SubPlot(2,2,2,""); gr->Title("'!' style; 'rgb' palette");
    gr->Box(); gr->Plot(y,"o!rgb");

    gr->SubPlot(2,2,3,""); gr->Title("just markers");
    gr->Box(); gr->Plot(y," +");

    gr->SubPlot(2,2,1); gr->Title("3d variant");
    gr->Rotate(50,60); gr->Box();
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");
    gr->Plot(xc,yc,z,"rs");
    return 0;
}

```

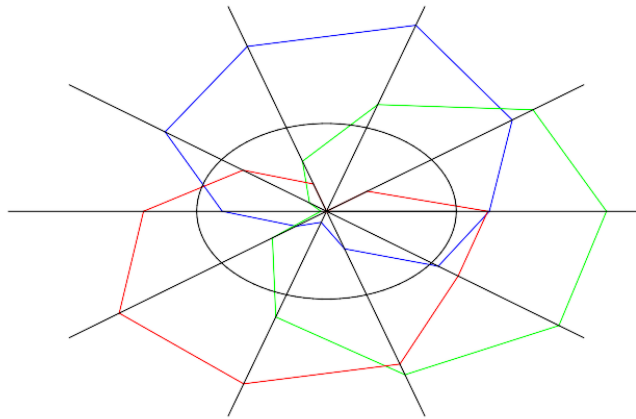


### 2.5.2 Radar sample

Function [radar], page 195, 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 2.2.3 [Curvilinear coordinates], page 25, or `Plot` in parabolic form with  $x=r*\cos(fi)$ ;  $y=r*\sin(fi)$ ; . The sample code is:

```
int sample(mglGraph *gr)
{
    mglData yr(10,3); yr.Modify("0.4*sin(pi*(2*x+y))+0.1*rnd");
    gr->SubPlot(1,1,0,""); gr->Title("Radar plot (with grid, '\n#')");
    gr->Radar(yr,"#");
    return 0;
}
```

# Radar plot (with grid, '#')



## 2.5.3 Step sample

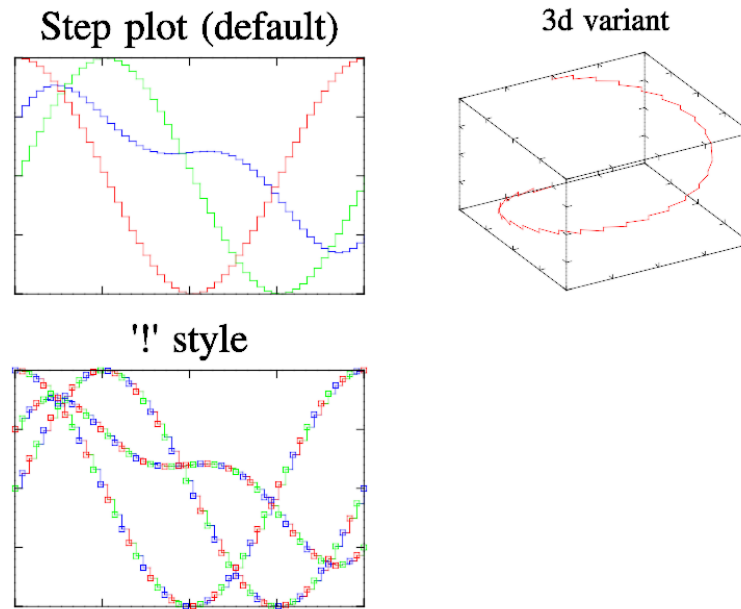
Function [step], page 195, plot data as stairs. It have the same options as Plot. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mglS_prepare1d(&y); gr->SetOrigin(0,0,0);
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");

    gr->SubPlot(2,2,0,""); gr->Title("Step plot (default)");
    gr->Box(); gr->Step(y);

    gr->SubPlot(2,2,1); gr->Title("3d variant"); gr->Rotate(50,60);
    gr->Box(); gr->Step(xc,yc,z,"r");

    gr->SubPlot(2,2,2,""); gr->Title("'!' style");
    gr->Box(); gr->Step(y,"s!rgb");
    return 0;
}
```



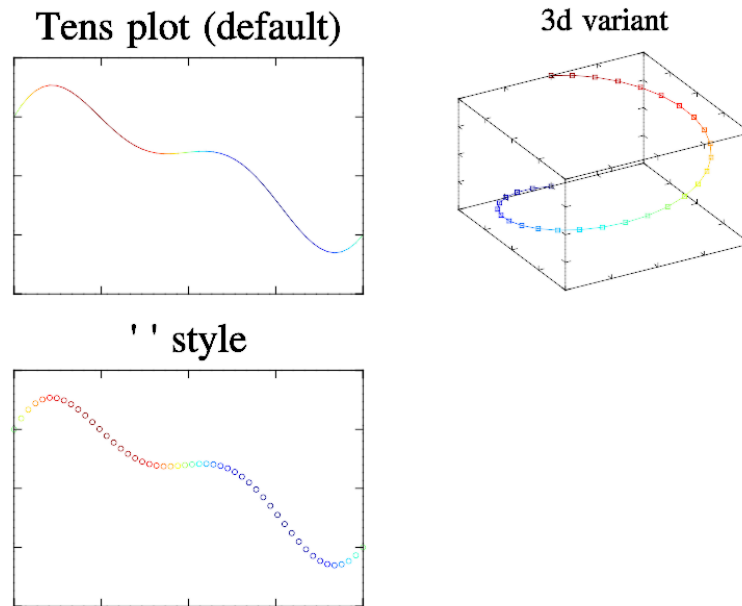
### 2.5.4 Tens sample

Function [tens], page 195, is variant of [plot], page 194, with smooth coloring along the curves. At this, color is determined as for surfaces (see Section 3.4 [Color scheme], page 149). The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mglS_prepare1d(&y); gr->SetOrigin(0,0,0);
    gr->SubPlot(2,2,0,""); gr->Title("Tens plot (default)");
    gr->Box(); gr->Tens(y.SubData(-1,0), y.SubData(-1,1));

    gr->SubPlot(2,2,2,""); gr->Title("' ' style");
    gr->Box(); gr->Tens(y.SubData(-1,0), y.SubData(-1,1),"o ");

    gr->SubPlot(2,2,1); gr->Title("3d variant"); gr->Rotate(50,60); gr->Box();
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");
    gr->Tens(xc,yc,z,z,"s");
    return 0;
}
```



### 2.5.5 Area sample

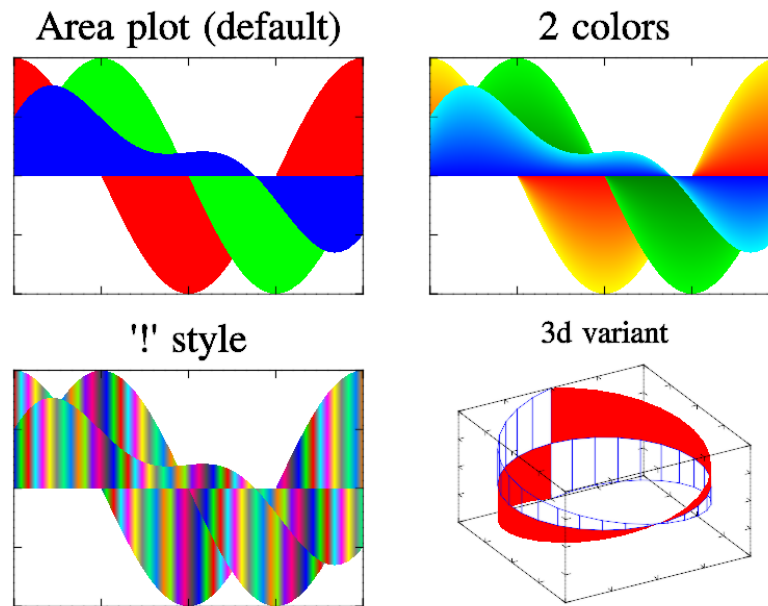
Function [area], page 196, fill the area between curve and axis plane. It support gradient filling if 2 colors per curve is specified. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mgl_prepare1d(&y); gr->SetOrigin(0,0,0);
    gr->SubPlot(2,2,0,""); gr->Title("Area plot (default)");
    gr->Box(); gr->Area(y);

    gr->SubPlot(2,2,1,""); gr->Title("2 colors");
    gr->Box(); gr->Area(y,"cbgGyr");

    gr->SubPlot(2,2,2,""); gr->Title("'!' style");
    gr->Box(); gr->Area(y,"!");

    gr->SubPlot(2,2,3); gr->Title("3d variant");
    gr->Rotate(50,60); gr->Box();
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");
    gr->Area(xc,yc,z,"r");
    yc.Modify("-sin(pi*(2*x-1))"); gr->Area(xc,yc,z,"b#");
    return 0;
}
```



### 2.5.6 Region sample

Function [region], page 197, fill the area between 2 curves. It support gradient filling if 2 colors per curve is specified. Also it can fill only the region  $y_1 < y < y_2$  if style 'i' is used. The sample code is:

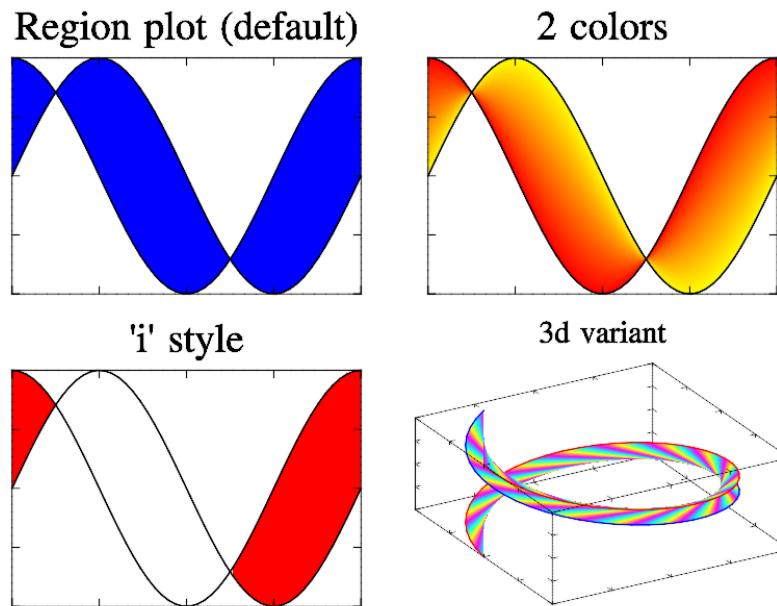
```
int sample(mglGraph *gr)
{
    mglData y;  mglS_prepare1d(&y);
    mglData y1 = y.SubData(-1,1), y2 = y.SubData(-1,2);  gr->SetOrigin(0,0,0);

    gr->SubPlot(2,2,0,"");  gr->Title("Region plot (default)"); gr->Box();
    gr->Region(y1,y2);  gr->Plot(y1,"k2");  gr->Plot(y2,"k2");

    gr->SubPlot(2,2,1,"");  gr->Title("2 colors");  gr->Box();
    gr->Region(y1,y2,"yr");  gr->Plot(y1,"k2");  gr->Plot(y2,"k2");

    gr->SubPlot(2,2,2,"");  gr->Title("'i' style");  gr->Box();
    gr->Region(y1,y2,"ir");  gr->Plot(y1,"k2");  gr->Plot(y2,"k2");

    gr->SubPlot(2,2,3,"^_");  gr->Title("3d variant");  gr->Rotate(40,60);  gr->Box();
    gr->Fill(y1,"cos(pi*x)"); gr->Fill(y2,"cos(pi*x+pi/3)");
    mglData x1(y1.nx), x2(y1.nx), z(y1.nx);
    gr->Fill(x1,"sin(pi*x)"); gr->Fill(x2,"sin(pi*x+pi/3)");  gr->Fill(z,"x");
    gr->Plot(x1,y1,z,"r2");  gr->Plot(x2,y2,z,"b2");
    gr->Region(x1,y1,z,x2,y2,z,"cmy!");
    return 0;
}
```



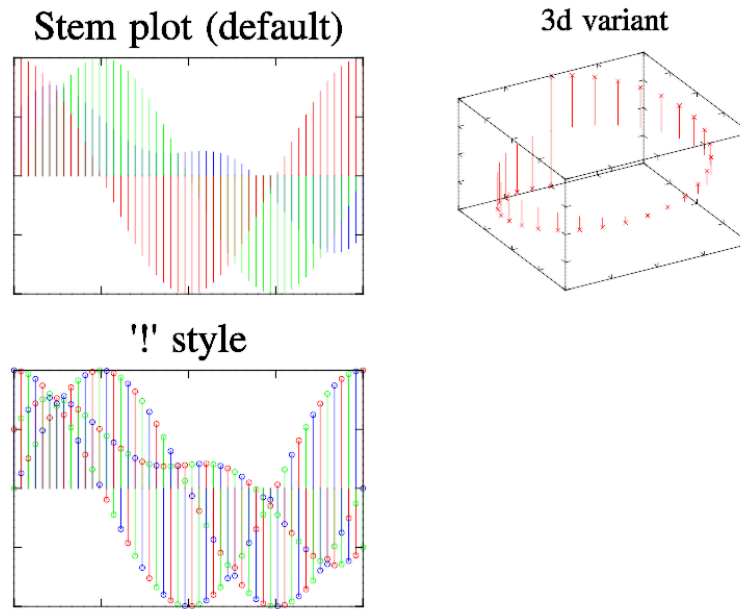
### 2.5.7 Stem sample

Function [stem], page 197, draw vertical bars. It is most attractive if markers are drawn too. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mglS_prepare1d(&y); gr->SetOrigin(0,0,0);
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");
    gr->SubPlot(2,2,0,""); gr->Title("Stem plot (default)");
    gr->Box(); gr->Stem(y);

    gr->SubPlot(2,2,1); gr->Title("3d variant"); gr->Rotate(50,60);
    gr->Box(); gr->Stem(xc,yc,z,"rx");

    gr->SubPlot(2,2,2,""); gr->Title("'i' style");
    gr->Box(); gr->Stem(y,"o!rgb");
    return 0;
}
```



### 2.5.8 Bars sample

Function [bars], page 198, 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:

```
int sample(mglGraph *gr)
{
    mglData ys(10,3); ys.Modify("0.8*sin(pi*(2*x+y/2))+0.2*rnd");
    gr->SetOrigin(0,0,0);
    gr->SubPlot(3,2,0,""); gr->Title("Bars plot (default)");
    gr->Box(); gr->Bars(ys);

    gr->SubPlot(3,2,1,""); gr->Title("2 colors");
    gr->Box(); gr->Bars(ys,"cbgGyr");

    gr->SubPlot(3,2,4,""); gr->Title("'\\#' style");
    gr->Box(); gr->Bars(ys,"#");

    gr->SubPlot(3,2,5); gr->Title("3d variant");
    gr->Rotate(50,60); gr->Box();
    mglData yc(30), xc(30), z(30); z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");
    gr->Bars(xc,yc,z,"r");

    gr->SetRanges(-1,1,-3,3);
    gr->SubPlot(3,2,2,""); gr->Title("'a' style");
    gr->Box(); gr->Bars(ys,"a");

    gr->SubPlot(3,2,3,""); gr->Title("'f' style");
```

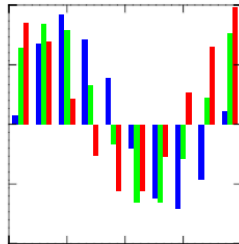


```

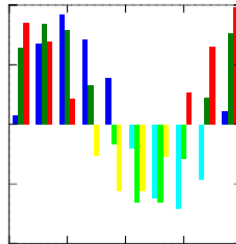
gr->Box();  gr->Bars(ys,"f");
return 0;
}

```

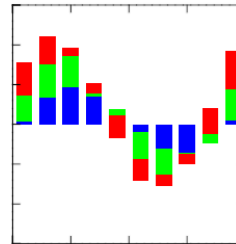
Bars plot (default)



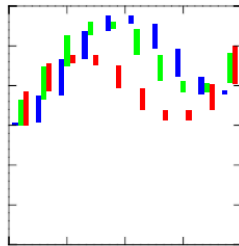
2 colors



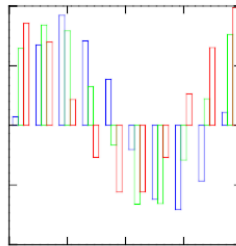
'a' style



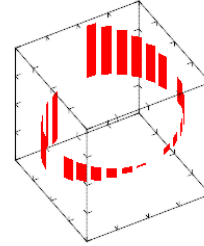
'f' style



'#' style



3d variant



### 2.5.9 Barh sample

Function [barh], page 198, is the similar to Bars but draw horizontal bars. The sample code is:

```

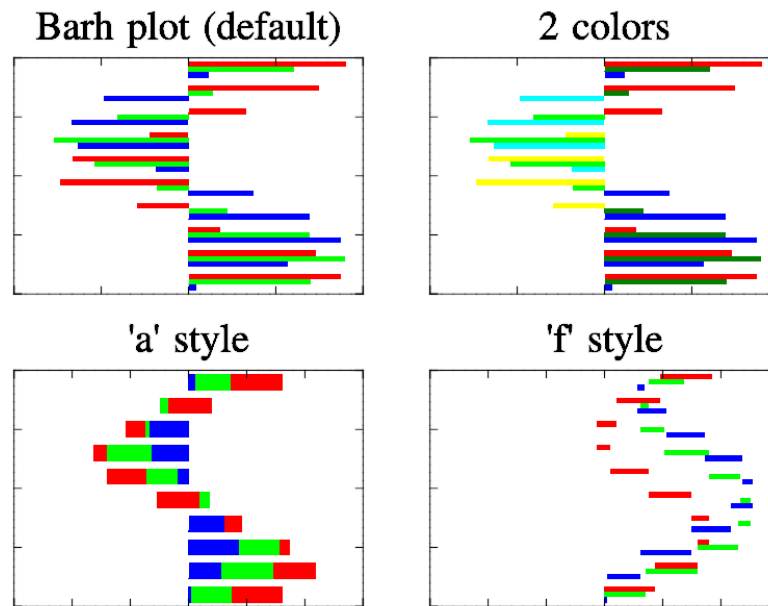
int sample(mglGraph *gr)
{
    mglData ys(10,3); ys.Modify("0.8*sin(pi*(2*x+y/2))+0.2*rnd");
    gr->SetOrigin(0,0,0);
    gr->SubPlot(2,2,0,""); gr->Title("Barh plot (default)");
    gr->Box(); gr->Barh(ys);

    gr->SubPlot(2,2,1,""); gr->Title("2 colors");
    gr->Box(); gr->Barh(ys,"cbgGyr");

    gr->SetRanges(-3,3,-1,1);
    gr->SubPlot(2,2,2,""); gr->Title("'a' style");
    gr->Box(); gr->Barh(ys,"a");

    gr->SubPlot(2,2,3,""); gr->Title("'f' style");
    gr->Box(); gr->Barh(ys,"f");
    return 0;
}

```



### 2.5.10 Cones sample

Function [cones], page 199, is similar to Bars but draw cones. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData ys(10,3);  ys.Modify("0.8*sin(pi*(2*x+y/2))+0.2*rnd");
    gr->Light(true);   gr->SetOrigin(0,0,0);
    gr->SubPlot(3,2,0); gr->Title("Cones plot");
    gr->Rotate(50,60); gr->Box();  gr->Cones(ys);

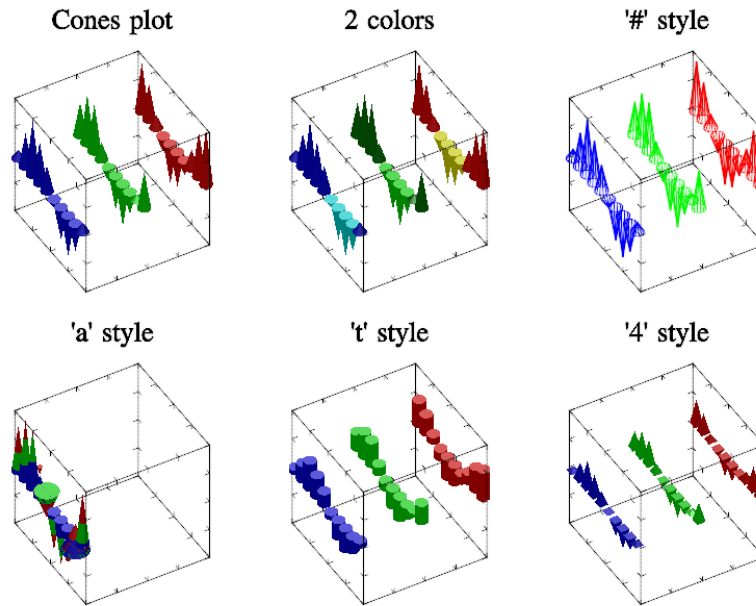
    gr->SubPlot(3,2,1); gr->Title("2 colors");
    gr->Rotate(50,60);  gr->Box();  gr->Cones(ys,"cbgGyr");

    gr->SubPlot(3,2,2); gr->Title("'#' style");
    gr->Rotate(50,60);  gr->Box();  gr->Cones(ys,"#");

    gr->SubPlot(3,2,3); gr->Title("'a' style");
    gr->SetRange('z',-2,2); // increase range since summation can exceed [-1,1]
    gr->Rotate(50,60);  gr->Box();  gr->Cones(ys,"a");

    gr->SubPlot(3,2,4); gr->Title("'t' style");
    gr->Rotate(50,60);  gr->Box();  gr->Cones(ys,"t");

    gr->SubPlot(3,2,5); gr->Title("'4' style");
    gr->Rotate(50,60);  gr->Box();  gr->Cones(ys,"4");
    return 0;
}
```



### 2.5.11 Chart sample

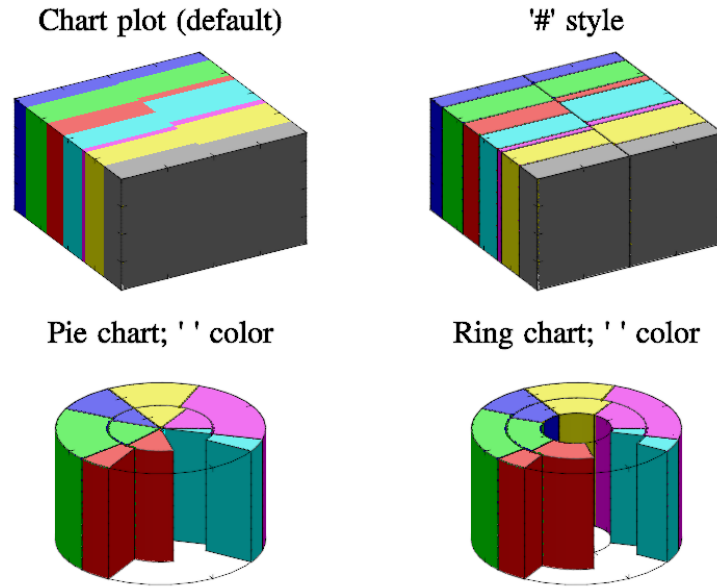
Function [chart], page 199, draw colored boxes with width proportional to data values. Use ' ' for empty box. Plot looks most attractive in polar coordinates – well known pie chart. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData ch(7,2); for(int i=0;i<7*2;i++) ch.a[i]=mgl_rnd()+0.1;
    gr->SubPlot(2,2,0); gr->Title("Chart plot (default)");
    gr->Light(true); gr->Rotate(50,60); gr->Box(); gr->Chart(ch);

    gr->SubPlot(2,2,1); gr->Title("'\\#' style");
    gr->Rotate(50,60); gr->Box(); gr->Chart(ch,"#");

    gr->SubPlot(2,2,2); gr->Title("Pie chart; ' ' color");
    gr->SetFunc("(y+1)/2*cos(pi*x)","(y+1)/2*sin(pi*x)","");
    gr->Rotate(50,60); gr->Box(); gr->Chart(ch,"bgr cmy#");

    gr->SubPlot(2,2,3); gr->Title("Ring chart; ' ' color");
    gr->SetFunc("(y+2)/3*cos(pi*x)","(y+2)/3*sin(pi*x)","");
    gr->Rotate(50,60); gr->Box(); gr->Chart(ch,"bgr cmy#");
    return 0;
}
```

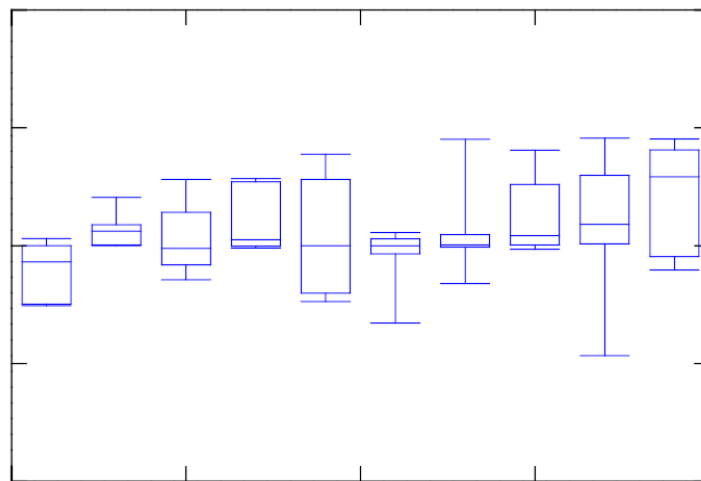


### 2.5.12 BoxPlot sample

Function [boxplot], page 199, draw box-and-whisker diagram. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a(10,7); a.Modify("(2*rnd-1)^3/2");
    gr->SubPlot(1,1,0,""); gr->Title("Boxplot plot");
    gr->Box(); gr->BoxPlot(a);
    return 0;
}
```

## Boxplot plot

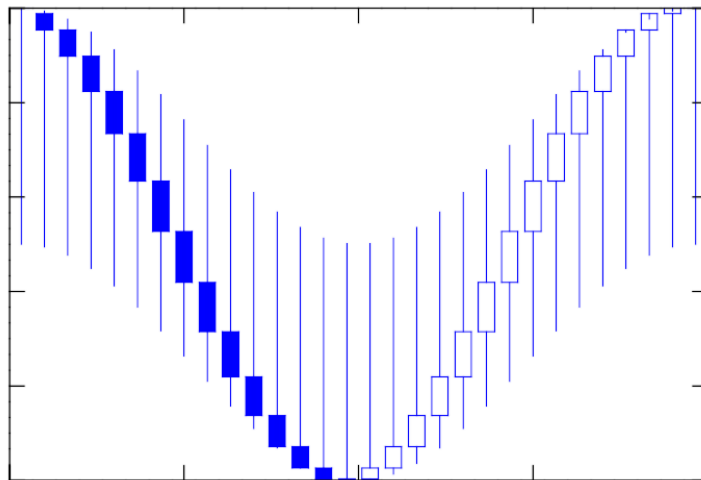


### 2.5.13 Candle sample

Function [candle], page 200, 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:

```
int sample(mglGraph *gr)
{
    mglData y(30); gr->Fill(y,"sin(pi*x/2)^2");
    mglData y1(30); gr->Fill(y1,"v/2",y);
    mglData y2(30); gr->Fill(y2,"(1+v)/2",y);
    gr->SubPlot(1,1,0,""); gr->Title("Candle plot (default)");
    gr->SetRange('y',0,1); gr->Box(); gr->Candle(y,y1,y2);
    return 0;
}
```

## Candle plot (default)



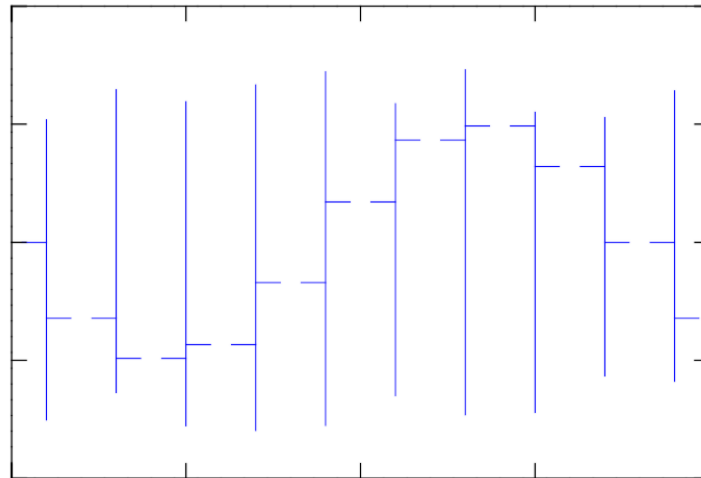
### 2.5.14 OHLC sample

Function [ohlc], page 201, 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:

```
int sample(mglGraph *gr)
{
    mglData o(10), h(10), l(10), c(10);
    gr->Fill(o,"0.5*sin(pi*x)"); gr->Fill(c,"0.5*sin(pi*(x+2/9))");
    gr->Fill(l,"0.3*rand-0.8"); gr->Fill(h,"0.3*rand+0.5");
    gr->SubPlot(1,1,0,""); gr->Title("OHLC plot");
    gr->Box(); gr->OHLC(o,h,l,c);
    return 0;
}
```

```
}
```

## OHLC plot



### 2.5.15 Error sample

Function [error], page 201, draw error boxes around the points. You can draw default boxes or semi-transparent symbol (like marker, see Section 3.3 [Line styles], page 147). Also you can set individual color for each box. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mgl_prepare1d(&y);
    mglData x0(10), y0(10), ex0(10), ey0(10);
    mreal x;
    for(int i=0;i<10;i++)
    {
        x = i/9.;
        x0.a[i] = 2*x-1 + 0.1*mgl_rnd()-0.05;
        y0.a[i] = 0.7*sin(2*M_PI*x)+0.5*cos(3*M_PI*x)+0.2*sin(M_PI*x)+0.2*mgl_rnd()-0.1;
        ey0.a[i]=0.2; ex0.a[i]=0.1;
    }

    gr->SubPlot(2,2,0,""); gr->Title("Error plot (default)");
    gr->Box(); gr->Plot(y.SubData(-1,0)); gr->Error(x0,y0,ex0,ey0,"ko");

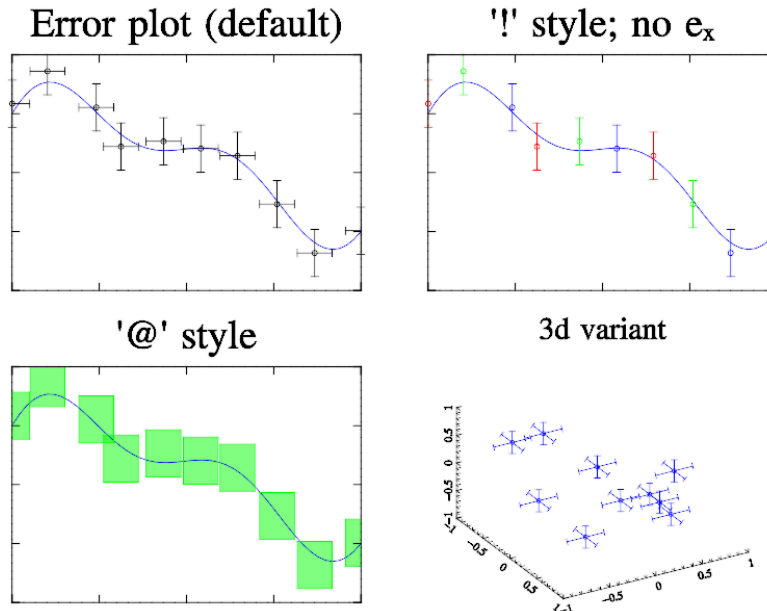
    gr->SubPlot(2,2,1,""); gr->Title("'!' style; no e_x");
    gr->Box(); gr->Plot(y.SubData(-1,0)); gr->Error(x0,y0,ey0,"o!rgb");

    gr->SubPlot(2,2,2,""); gr->Title("'\\' style");
    gr->Box(); gr->Plot(y.SubData(-1,0)); gr->Error(x0,y0,ex0,ey0,"@", "alpha 0.5");
```

```

gr->SubPlot(2,2,3); gr->Title("3d variant"); gr->Rotate(50,60);
for(int i=0;i<10;i++)
    gr->Error(mglPoint(2*mgl_rnd()-1,2*mgl_rnd()-1,2*mgl_rnd()-1),
              mglPoint(0.2,0.2,0.2),"bo");
gr->Axis();
return 0;
}

```



Additionally, you can use solid large "marks" instead of error boxes by selecting proper style.

```

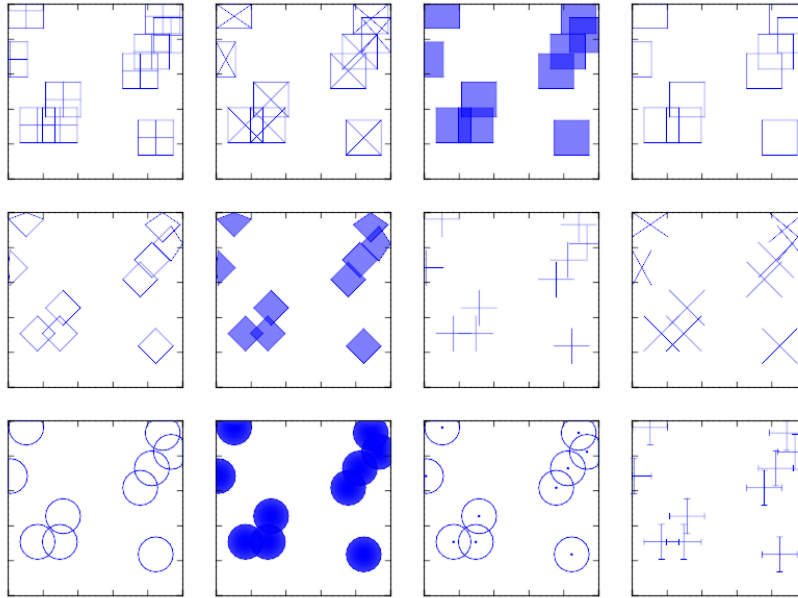
int sample(mglGraph *gr)
{
    mglData x0(10), y0(10), ex(10), ey(10);
    for(int i=0;i<10;i++)
    { x0.a[i] = mgl_rnd(); y0.a[i] = mgl_rnd(); ey.a[i] = ex.a[i] = 0.1; }
    gr->SetRanges(0,1,0,1); gr->Alpha(true);
    gr->SubPlot(4,3,0,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#+@");
    gr->SubPlot(4,3,1,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#x@");
    gr->SubPlot(4,3,2,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#s@","alpha 0.5");
    gr->SubPlot(4,3,3,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#s@");
    gr->SubPlot(4,3,4,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#d@");
    gr->SubPlot(4,3,5,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#d@","alpha 0.5");
    gr->SubPlot(4,3,6,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#+@");
    gr->SubPlot(4,3,7,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#x@");
    gr->SubPlot(4,3,8,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#o@");
    gr->SubPlot(4,3,9,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#o@","alpha 0.5");
    gr->SubPlot(4,3,10,""); gr->Box(); gr->Error(x0,y0,ex,ey,"#.@");
}

```

```

gr->SubPlot(4,3,11,""); gr->Box(); gr->Error(x0,y0,ex,ey);
}

```



### 2.5.16 Mark sample

Function [mark], page 201, draw markers at points. It is mostly the same as Plot but marker size can be variable. The sample code is:

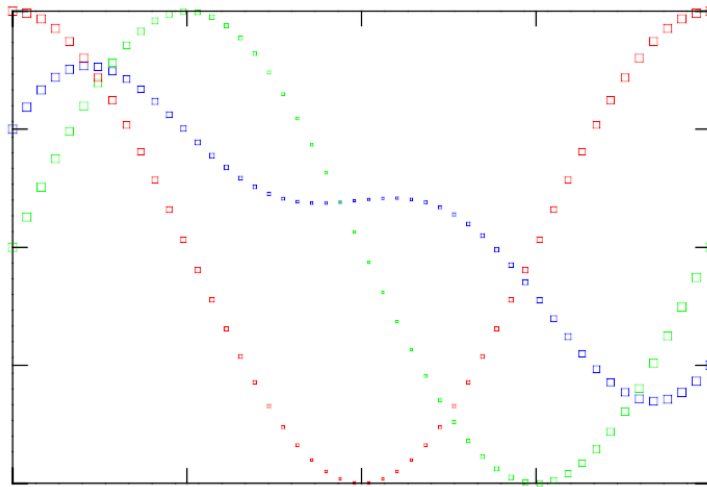
```

int sample(mglGraph *gr)
{
    mglData y,y1; mgls_prepare1d(&y,&y1);
    gr->SubPlot(1,1,0,""); gr->Title("Mark plot (default)");
    gr->Box(); gr->Mark(y,y1,"s");
    return 0;
}

```



## Mark plot (default)

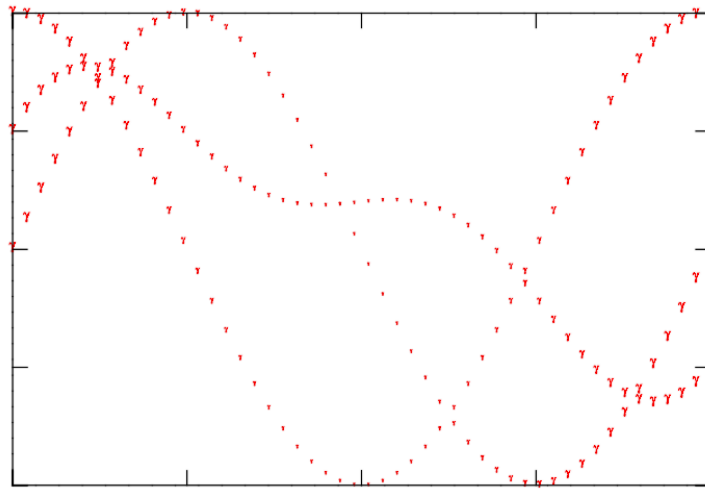


### 2.5.17 TextMark sample

Function [textmark], page 202, like Mark but draw text instead of markers. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y,y1; mgl_prepare1d(&y,&y1);
    gr->SubPlot(1,1,0,""); gr->Title("TextMark plot (default)");
    gr->Box(); gr->TextMark(y,y1,"\\gamma","r");
    return 0;
}
```

## TextMark plot (default)

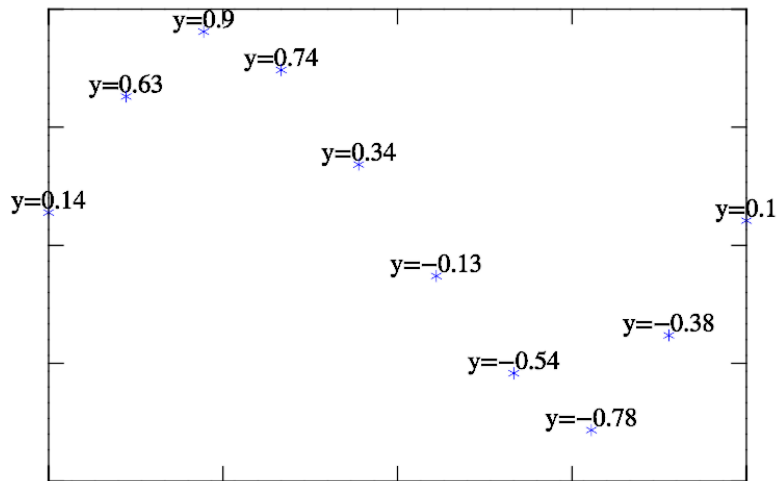


### 2.5.18 Label sample

Function [label], page 203, 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:

```
int sample(mglGraph *gr)
{
    mglData ys(10); ys.Modify("0.8*sin(pi*2*x)+0.2*rnd");
    gr->SubPlot(1,1,0,""); gr->Title("Label plot");
    gr->Box(); gr->Plot(ys," *"); gr->Label(ys,"y=%y");
    return 0;
}
```

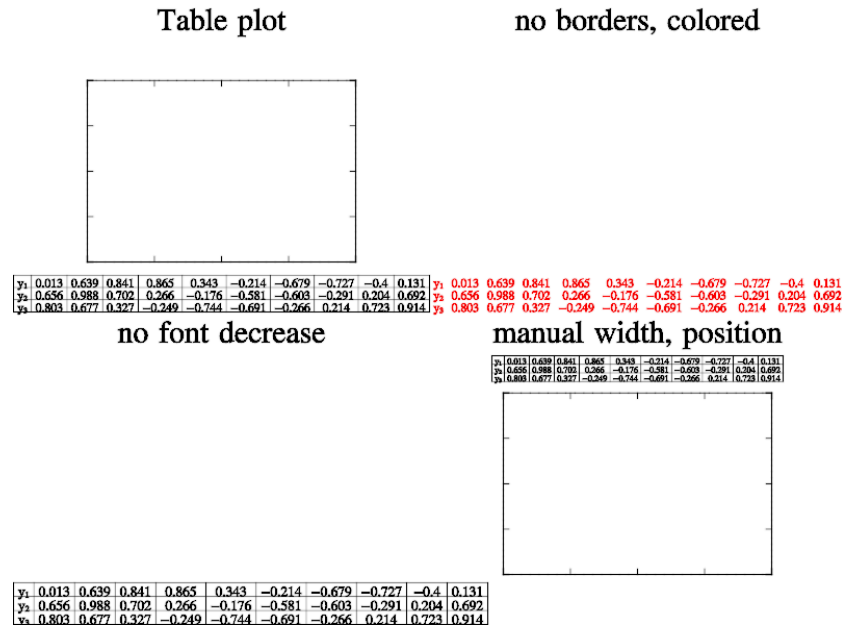
# Label plot



## 2.5.19 Table sample

Function [table], page 204, draw table with data values. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData ys(10,3); ys.Modify("0.8*sin(pi*(2*x+y/2))+0.2*rnd");
    gr->SubPlot(2,2,0); gr->Title("Table plot");
    gr->Table(ys,"y_1\nty_2\nty_3"); gr->Box();
    gr->SubPlot(2,2,1); gr->Title("no borders, colored");
    gr->Table(ys,"y_1\nty_2\nty_3","r|");
    gr->SubPlot(2,2,2); gr->Title("no font decrease");
    gr->Table(ys,"y_1\nty_2\nty_3","#");
    gr->SubPlot(2,2,3); gr->Title("manual width, position");
    gr->Table(0.5, 0.95, ys,"y_1\nty_2\nty_3","#", "value 0.7"); gr->Box();
    return 0;
}
```



### 2.5.20 Tube sample

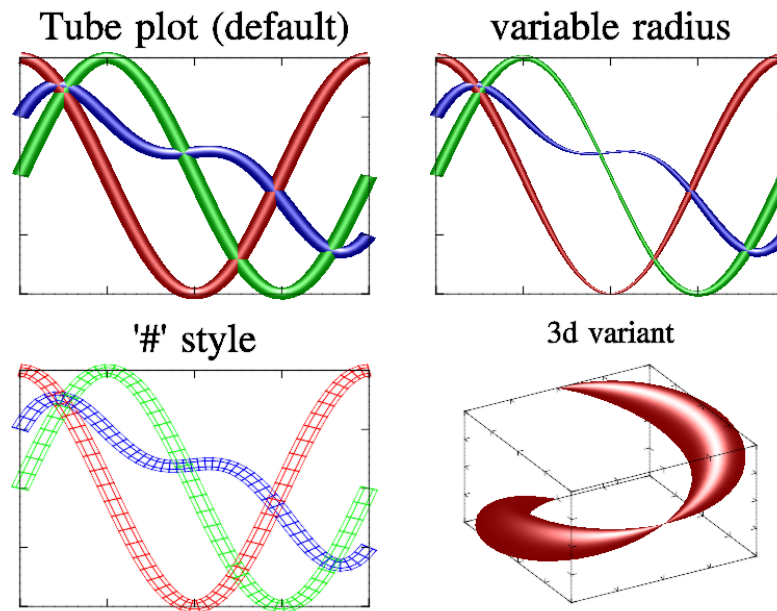
Function [tube], page 205, draw tube with variable radius. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y,y1,y2;  mglS_prepare1d(&y,&y1,&y2); y1/=20;
    gr->SubPlot(2,2,0,"");  gr->Title("Tube plot (default)");
    gr->Light(true);  gr->Box();  gr->Tube(y,0.05);

    gr->SubPlot(2,2,1,"");  gr->Title("variable radius");
    gr->Box();  gr->Tube(y,y1);

    gr->SubPlot(2,2,2,"");  gr->Title("'\\#\' style");
    gr->Box();  gr->Tube(y,0.05,"#");
    mglData yc(50), xc(50), z(50);  z.Modify("2*x-1");
    yc.Modify("sin(pi*(2*x-1))"); xc.Modify("cos(pi*2*x-pi)");

    gr->SubPlot(2,2,3); gr->Title("3d variant");  gr->Rotate(50,60);
    gr->Box();  gr->Tube(xc,yc,z,y2,"r");
    return 0;
}
```



### 2.5.21 Tape sample

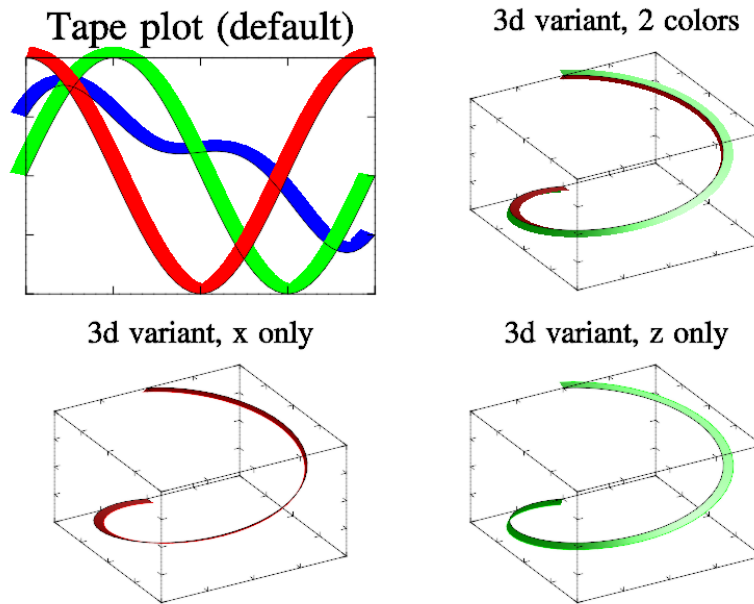
Function [tape], page 196, draw tapes which rotate around the curve as normal and binormal. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mglS_prepare1d(&y);
    mglData xc(50), yc(50), z(50);
    yc.Modify("sin(pi*(2*x-1))");
    xc.Modify("cos(pi*2*x-pi)"); z.Fill(-1,1);
    gr->SubPlot(2,2,0,""); gr->Title("Tape plot (default)");
    gr->Box(); gr->Tape(y); gr->Plot(y,"k");

    gr->SubPlot(2,2,1); gr->Title("3d variant, 2 colors");
    gr->Rotate(50,60); gr->Light(true);
    gr->Box(); gr->Plot(xc,yc,z,"k"); gr->Tape(xc,yc,z,"rg");

    gr->SubPlot(2,2,2); gr->Title("3d variant, x only"); gr->Rotate(50,60);
    gr->Box(); gr->Plot(xc,yc,z,"k");
    gr->Tape(xc,yc,z,"xr"); gr->Tape(xc,yc,z,"xr#");

    gr->SubPlot(2,2,3); gr->Title("3d variant, z only"); gr->Rotate(50,60);
    gr->Box(); gr->Plot(xc,yc,z,"k");
    gr->Tape(xc,yc,z,"zg"); gr->Tape(xc,yc,z,"zg#");
    return 0;
}
```



### 2.5.22 Torus sample

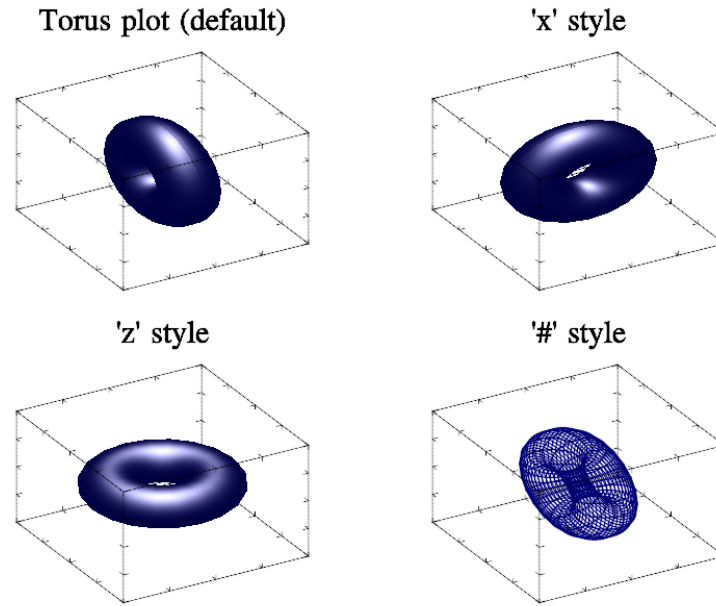
Function [torus], page 205, draw surface of the curve rotation. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y1,y2;  mglS_prepare1d(0,&y1,&y2);
    gr->SubPlot(2,2,0); gr->Title("Torus plot (default)");
    gr->Light(true);  gr->Rotate(50,60);  gr->Box();  gr->Torus(y1,y2);
    if(mini)  return;

    gr->SubPlot(2,2,1); gr->Title("'x' style"); gr->Rotate(50,60);
    gr->Box();  gr->Torus(y1,y2,"x");

    gr->SubPlot(2,2,2); gr->Title("'z' style"); gr->Rotate(50,60);
    gr->Box();  gr->Torus(y1,y2,"z");

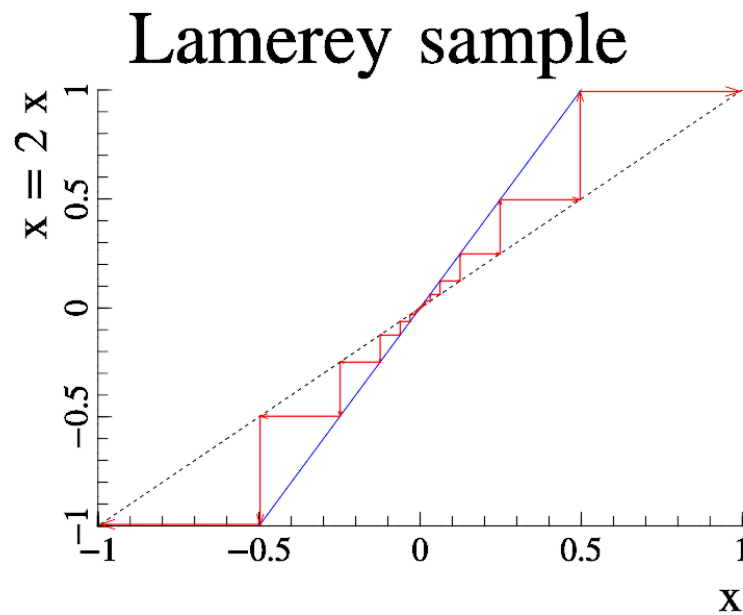
    gr->SubPlot(2,2,3); gr->Title("'\\#' style"); gr->Rotate(50,60);
    gr->Box();  gr->Torus(y1,y2,"#");
    return 0;
}
```



### 2.5.23 Lamerey sample

Function [lamerey], page 206, draw Lamerey diagram. The sample code is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(1,1,0,"<_");
    gr->Title("Lamerey sample");
    gr->Axis(); gr->Label('x',"\\i x");
    gr->Label('y',"\\bar{\\i x} = 2 \\i{x}");
    gr->FPlot("x","k="); gr->FPlot("2*x","b");
    gr->Lamerey( 0.00097,"2*x","rv~");
    gr->Lamerey(-0.00097,"2*x","rv~");
}
```



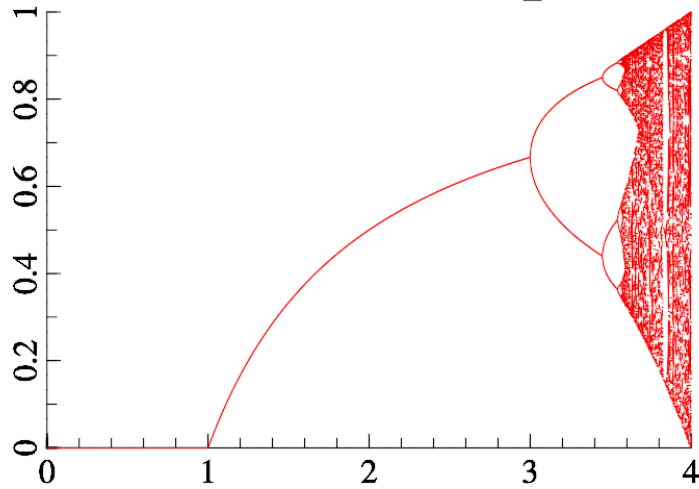
## 2.5.24 Bifurcation sample

Function [bifurcation], page 206, draw Bifurcation diagram for logistic map. The sample code is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(1,1,0,"<_");
    gr->Title("Bifurcation sample");
    gr->SetRanges(0,4,0,1);    gr->Axis();
    gr->Bifurcation(0.005,"x*y*(1-y)","r");
}
```



## Bifurcation sample

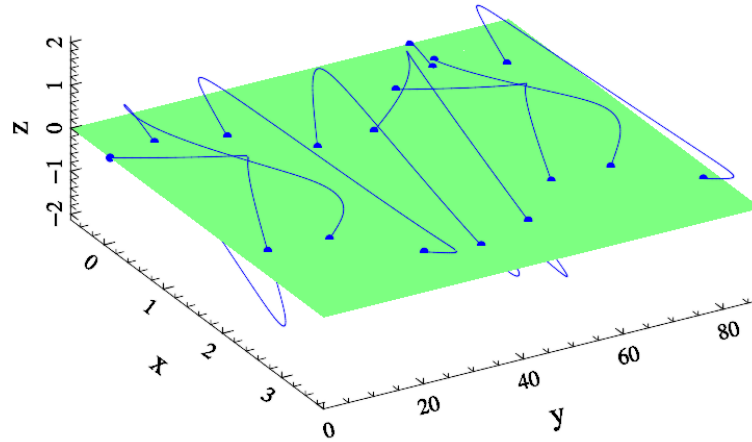


### 2.5.25 Pmap sample

Function [pmap], page 206, draw Poincare map – show intersections of the curve and the surface. The sample code is:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(1,1,0,"<_");
    gr->Title("Poincare map sample");
    mglData ini(3);      ini[0]=0.1;
    mglData r(mglODE("cos(y)+sin(z);cos(z)+sin(x);cos(x)+sin(y)","xyz",ini,0.1,100));
    mglData x(r.SubData(0)),y(r.SubData(1)), z(r.SubData(2));
    gr->Rotate(40,60);  gr->SetRanges(x,y,z);
    gr->Axis(); gr->FSurf("0"); gr->Plot(x,y,z,"b");
    gr->Label('x',"\\i x",0);  gr->Label('y',"\\i y",0);  gr->Label('z',"\\i z",0);
    gr->Pmap(x,y,z,z, "b#o");
}
```

## Poincare map sample



### 2.6 2D samples

This section is devoted to visualization of 2D data arrays. 2D means the data which depend on 2 indexes (parameters) like matrix  $z(i,j)=z(x(i),y(j))$ ,  $i=1\dots n$ ,  $j=1\dots m$  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

```
void mgls_prepare2d(mglData *a, mglData *b=0, mglData *v=0)
{
    register long i,j,n=50,m=40,i0;
    if(a) a->Create(n,m);    if(b) b->Create(n,m);
    if(v) { v->Create(9); v->Fill(-1,1); }
    mreal x,y;
    for(i=0;i<n;i++)    for(j=0;j<m;j++)
    {
        x = i/(n-1.); y = j/(m-1.); i0 = i+n*j;
        if(a) a->a[i0] = 0.6*sin(2*M_PI*x)*sin(3*M_PI*y)+0.4*cos(3*M_PI*x*y);
        if(b) b->a[i0] = 0.6*cos(2*M_PI*x)*cos(3*M_PI*y)+0.4*cos(3*M_PI*x*y);
    }
}
```

or using C functions

```
void mgls_prepare2d(HMDT a, HMDT b=0, HMDT v=0)
{
    register long i,j,n=50,m=40,i0;
    if(a) mgl_data_create(a,n,m,1);
    if(b) mgl_data_create(b,n,m,1);
    if(v) { mgl_data_create(v,9,1,1); mgl_data_fill(v,-1,1,'x'); }
    mreal x,y;
    for(i=0;i<n;i++)    for(j=0;j<m;j++)
```

```

{
  x = i/(n-1.); y = j/(m-1.); i0 = i+n*j;
  if(a) mgl_data_set_value(a, 0.6*sin(2*M_PI*x)*sin(3*M_PI*y)+0.4*cos(3*M_PI*x*y), i,j,0)
  if(b) mgl_data_set_value(b, 0.6*cos(2*M_PI*x)*cos(3*M_PI*y)+0.4*cos(3*M_PI*x*y), i,j,0)
}
}

```

### 2.6.1 Surf sample

Function [surf], page 207, is most standard way to visualize 2D data array. Surf use color scheme for coloring (see Section 3.4 [Color scheme], page 149). You can use ‘#’ style for drawing black meshes on the surface. The sample code is:

```

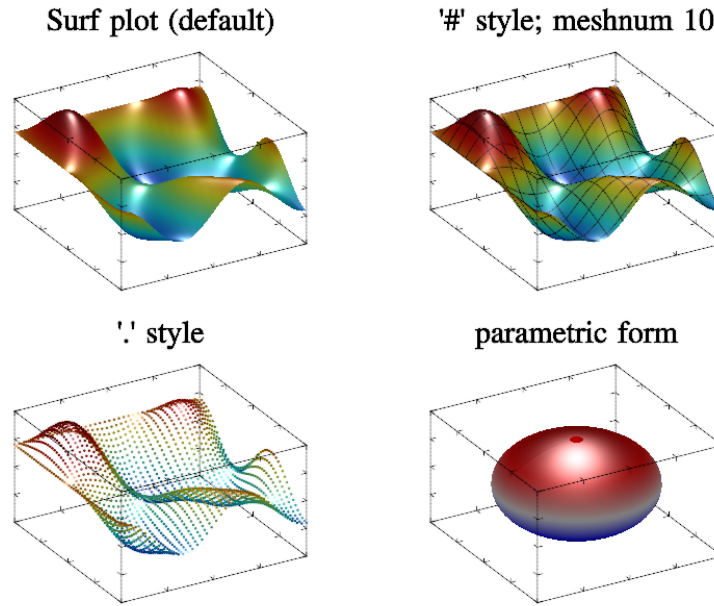
int sample(mglGraph *gr)
{
  mglData a; mgl_prepare2d(&a);
  gr->SubPlot(2,2,0); gr->Title("Surf plot (default)");
  gr->Light(true); gr->Rotate(50,60); gr->Box(); gr->Surf(a);

  gr->SubPlot(2,2,1); gr->Title("'\\#' style; meshnum 10");
  gr->Rotate(50,60); gr->Box(); gr->Surf(a,"#", "meshnum 10");

  gr->SubPlot(2,2,2); gr->Title("'.' style");
  gr->Rotate(50,60); gr->Box(); gr->Surf(a,".");

  gr->SubPlot(2,2,3); gr->Title("parametric form");
  mglData x(50,40),y(50,40),z(50,40);
  gr->Fill(x,"0.8*sin(pi*x)*sin(pi*(y+1)/2)");
  gr->Fill(y,"0.8*cos(pi*x)*sin(pi*(y+1)/2)");
  gr->Fill(z,"0.8*cos(pi*(y+1)/2)");
  gr->Rotate(50,60); gr->Box(); gr->Surf(x,y,z,"BbwrR");
  return 0;
}

```

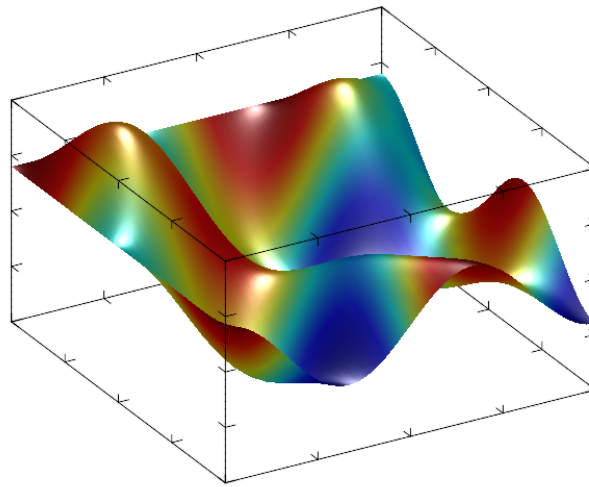


### 2.6.2 SurfC sample

Function [surfC], page 217, is similar to [surf], page 207, but its coloring is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mglS_prepare2d(&a,&b);
    gr->Title("SurfC plot");  gr->Rotate(50,60);
    gr->Light(true);  gr->Box();  gr->SurfC(a,b);
    return 0;
}
```

## SurfC plot

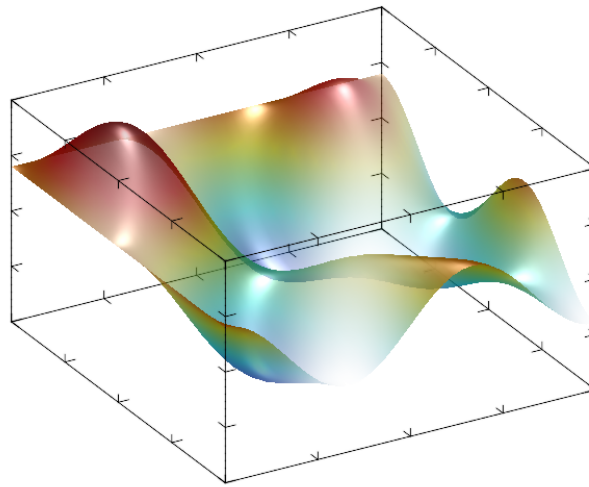


### 2.6.3 SurfA sample

Function [surfa], page 218, is similar to [surf], page 207, but its transparency is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b; mglS_prepare2d(&a,&b);
    gr->Title("SurfA plot"); gr->Rotate(50,60);
    gr->Alpha(true); gr->Light(true);
    gr->Box(); gr->SurfA(a,b);
    return 0;
}
```

## SurfA plot

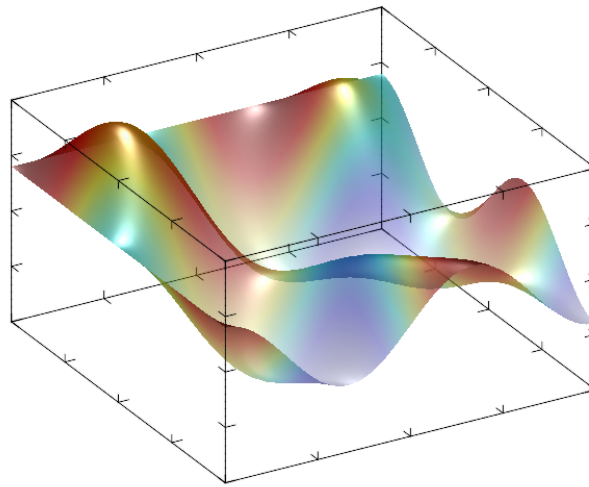


### 2.6.4 SurfCA sample

Function [surfca], page 219, is similar to [surf], page 207, but its color and transparency is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b; mgl_prepare2d(&a,&b);
    gr->Title("SurfCA plot"); gr->Rotate(50,60);
    gr->Alpha(true); gr->Light(true);
    gr->Box(); gr->SurfCA(a,b,a);
    return 0;
}
```

## SurfCA plot

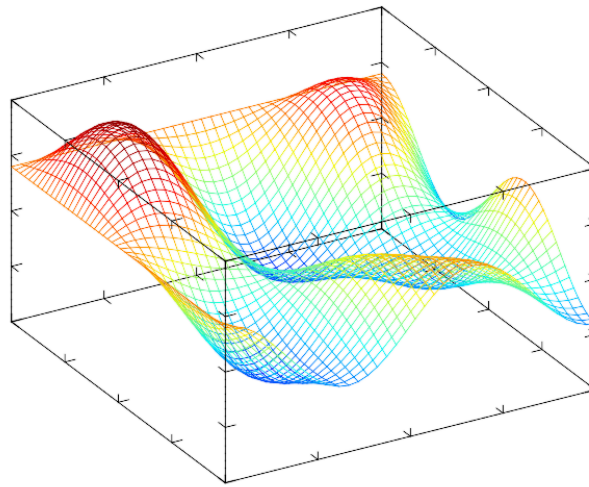


### 2.6.5 Mesh sample

Function [mesh], page 207, draw wired surface. You can use [meshnum], page 160, for changing number of lines to be drawn. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a;  mglS_prepare2d(&a);
    gr->Title("Mesh plot"); gr->Rotate(50,60);
    gr->Box();  gr->Mesh(a);
    return 0;
}
```

## Mesh plot



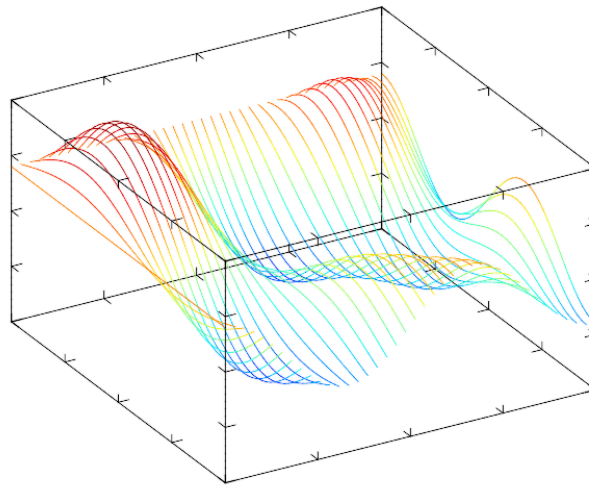
### 2.6.6 Fall sample

Function [fall], page 208, draw waterfall surface. You can use [meshnum], page 160, for changing number of lines to be drawn. Also you can use 'x' style for drawing lines in other direction. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a;  mgl_prepare2d(&a);
    gr->Title("Fall plot"); gr->Rotate(50,60);
    gr->Box();  gr->Fall(a);
    return 0;
}
```



## Fall plot

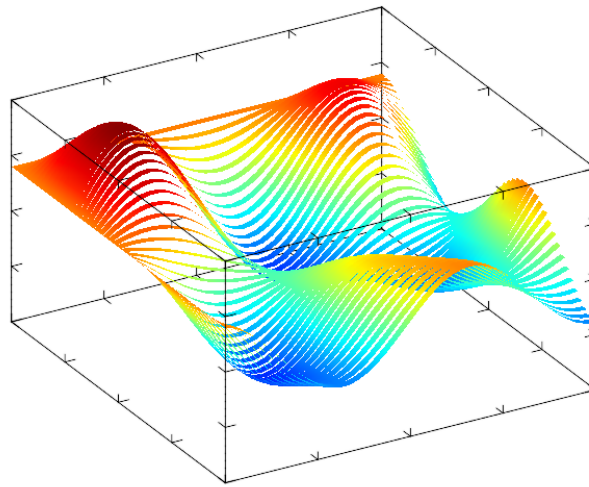


### 2.6.7 Belt sample

Function [belt], page 208, draw surface by belts. You can use 'x' style for drawing lines in other direction. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a;  mglS_prepare2d(&a);
    gr->Title("Belt plot"); gr->Rotate(50,60);
    gr->Box();  gr->Belt(a);
    return 0;
}
```

## Belt plot



### 2.6.8 Boxs sample

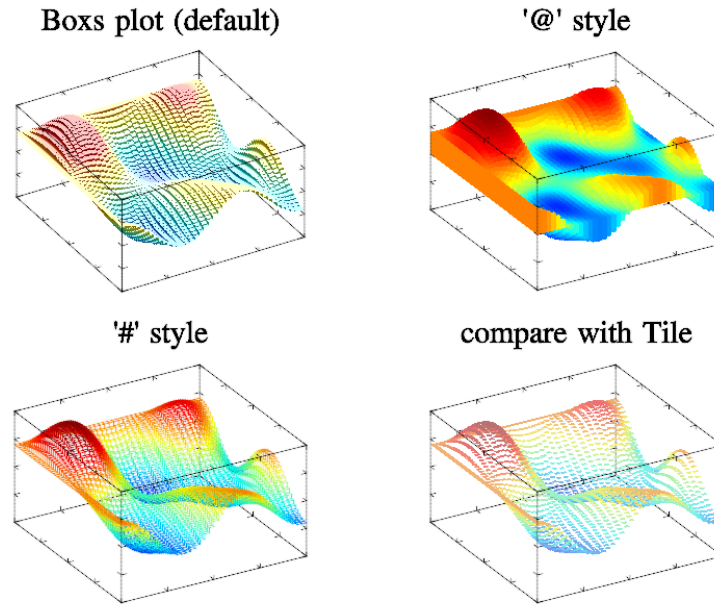
Function [boxs], page 208, draw surface by boxes. You can use '#' for drawing wire plot. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a; mgl_prepare2d(&a);
    gr->SetOrigin(0,0,0); gr->Light(true);
    gr->SubPlot(2,2,0); gr->Title("Boxs plot (default)");
    gr->Rotate(40,60); gr->Box(); gr->Boxs(a);

    gr->SubPlot(2,2,1); gr->Title("'\\@' style");
    gr->Rotate(50,60); gr->Box(); gr->Boxs(a,"@");

    gr->SubPlot(2,2,2); gr->Title("'\\#' style");
    gr->Rotate(50,60); gr->Box(); gr->Boxs(a,"#");

    gr->SubPlot(2,2,3); gr->Title("compare with Tile");
    gr->Rotate(50,60); gr->Box(); gr->Tile(a);
    return 0;
}
```

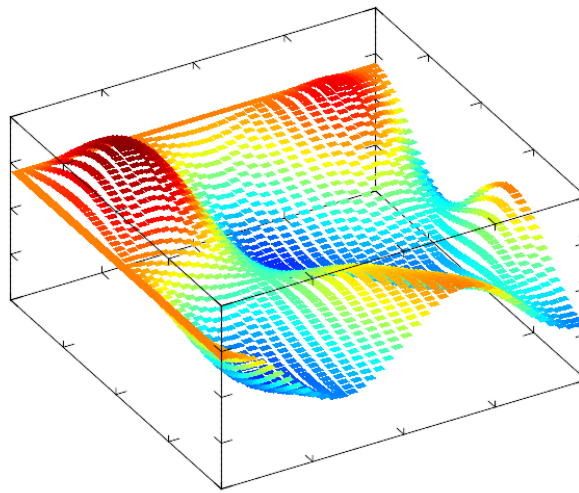


### 2.6.9 Tile sample

Function [tile], page 209, draw surface by tiles. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a; mglS_prepare2d(&a);
    gr->Title("Tile plot");
    gr->Rotate(40,60); gr->Box(); gr->Tile(a);
    return 0;
}
```

### Tile plot

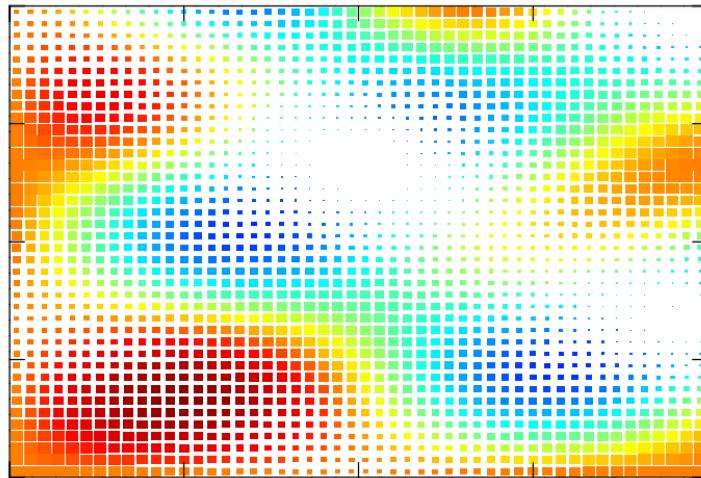


### 2.6.10 TileS sample

Function [tiles], page 220, is similar to [tile], page 209, but tile sizes is determined by another data. This allows one to simulate transparency of the plot. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mglS_prepare2d(&a,&b);
    gr->SubPlot(1,1,0,""); gr->Title("TileS plot");
    gr->Box();  gr->TileS(a,b);
    return 0;
}
```

## TileS plot



### 2.6.11 Dens sample

Function [dens], page 209, draw density plot for surface. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,a1(30,40,3);  mglS_prepare2d(&a);
    gr->Fill(a1,"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/");
    gr->SubPlot(2,2,0,""); gr->Title("Dens plot (default)");
    gr->Box();  gr->Dens(a);

    gr->SubPlot(2,2,1); gr->Title("3d variant");
    gr->Rotate(50,60);  gr->Box();  gr->Dens(a);

    gr->SubPlot(2,2,2,"");  gr->Title("'\\#' style; meshnum 10");
    gr->Box();  gr->Dens(a,"#", "meshnum 10");

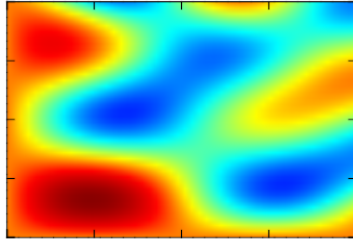
    gr->SubPlot(2,2,3); gr->Title("several slices");
```

```

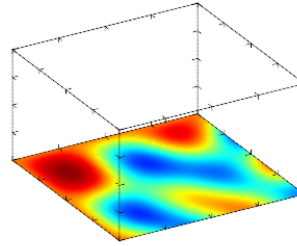
gr->Rotate(50,60);    gr->Box();  gr->Dens(a1);
return 0;
}

```

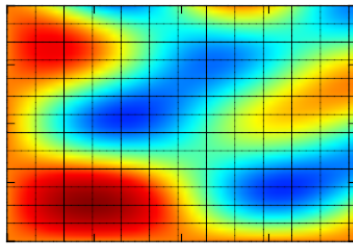
Dens plot (default)



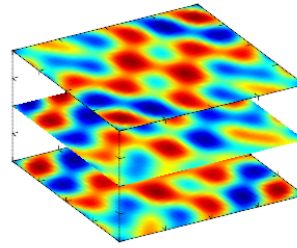
3d variant



'#' style; meshnum 10



several slices



### 2.6.12 Cont sample

Function [cont], page 209, 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:

```

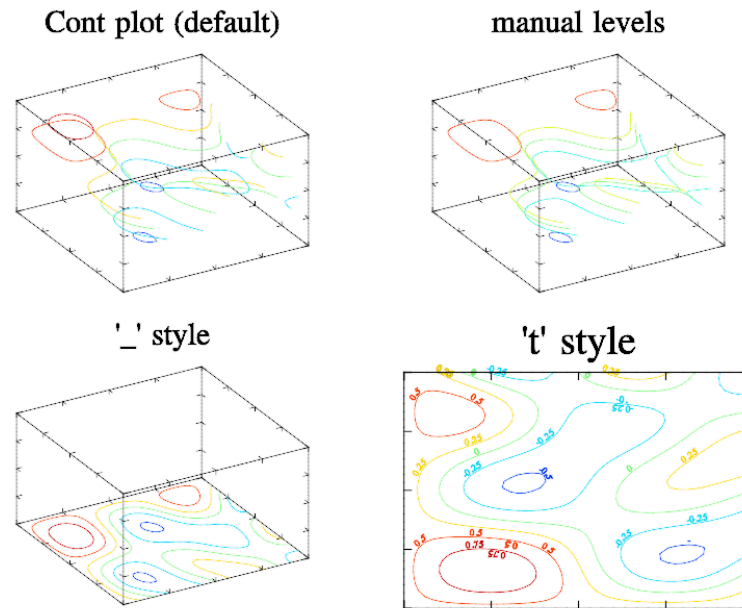
int sample(mglGraph *gr)
{
    mglData a,v(5); mglS_prepare2d(&a); v.a[0]=-0.5; v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v
    gr->SubPlot(2,2,0); gr->Title("Cont plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->Cont(a);

    gr->SubPlot(2,2,1); gr->Title("manual levels");
    gr->Rotate(50,60); gr->Box(); gr->Cont(v,a);

    gr->SubPlot(2,2,2); gr->Title("'\\_' style");
    gr->Rotate(50,60); gr->Box(); gr->Cont(a,"_");

    gr->SubPlot(2,2,3,""); gr->Title("'t' style");
    gr->Box(); gr->Cont(a,"t");
    return 0;
}

```



### 2.6.13 ContF sample

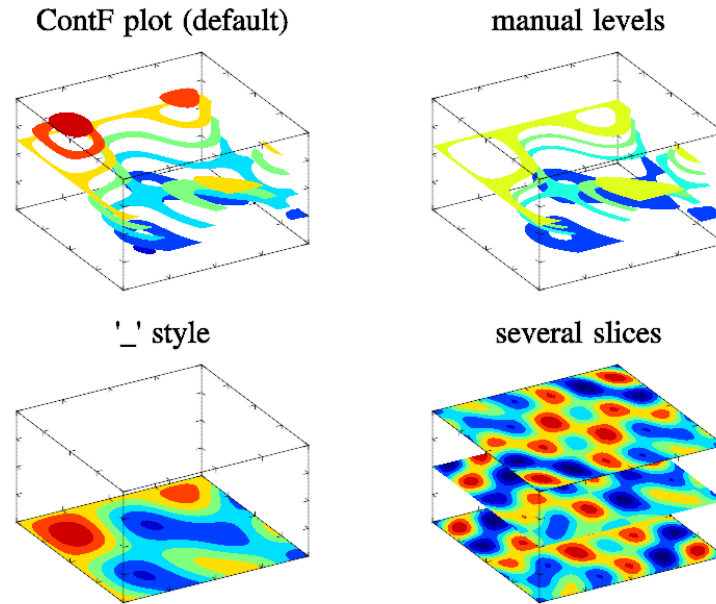
Function [contf], page 210, draw filled contours. You can select automatic (default) or manual levels for contours. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,v(5),a1(30,40,3); mgl_prepare2d(&a); v.a[0]=-0.5;
    v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v.a[4]=0.5;
    gr->SubPlot(2,2,0); gr->Title("ContF plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->ContF(a);

    gr->SubPlot(2,2,1); gr->Title("manual levels");
    gr->Rotate(50,60); gr->Box(); gr->ContF(v,a);

    gr->SubPlot(2,2,2); gr->Title("'\\_' style");
    gr->Rotate(50,60); gr->Box(); gr->ContF(a,"_");

    gr->Fill(a1,"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)");
    gr->SubPlot(2,2,3); gr->Title("several slices");
    gr->Rotate(50,60); gr->Box(); gr->ContF(a1);
    return 0;
}
```



### 2.6.14 ContD sample

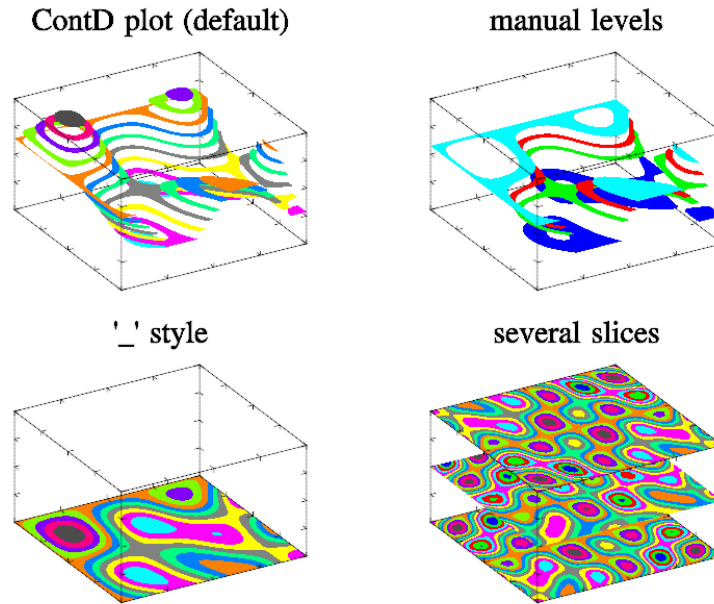
Function [contd], page 211, is similar to ContF but with manual contour colors. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,v(5),a1(30,40,3); mglS_prepare2d(&a); v.a[0]=-0.5;
    v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v.a[4]=0.5;
    gr->SubPlot(2,2,0); gr->Title("ContD plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->ContD(a);

    gr->SubPlot(2,2,1); gr->Title("manual levels");
    gr->Rotate(50,60); gr->Box(); gr->ContD(v,a);

    gr->SubPlot(2,2,2); gr->Title("'\\_' style");
    gr->Rotate(50,60); gr->Box(); gr->ContD(a,"_");

    gr->Fill(a1,"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/");
    gr->SubPlot(2,2,3); gr->Title("several slices");
    gr->Rotate(50,60); gr->Box(); gr->ContD(a1);
    return 0;
}
```



### 2.6.15 ContV sample

Function [contv], page 211, draw vertical cylinders (belts) at contour lines. The sample code is:

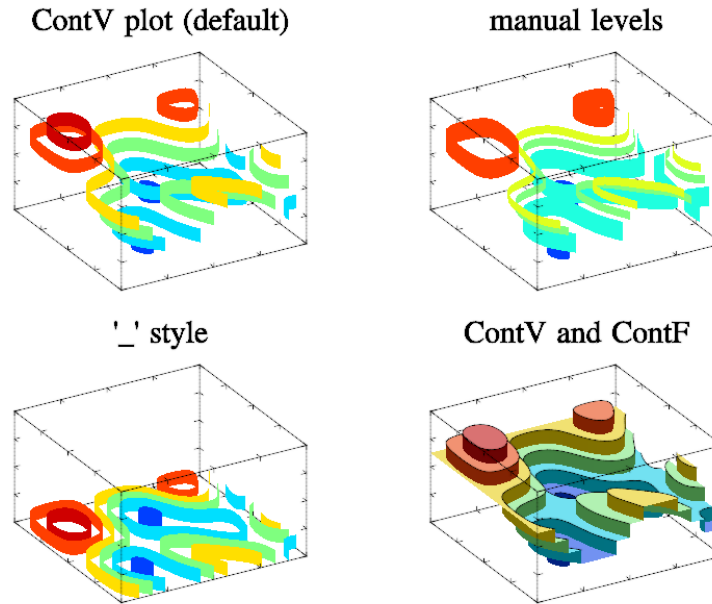
```
int sample(mglGraph *gr)
{
    mglData a,v(5); mgl_prepare2d(&a); v.a[0]=-0.5;
    v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v.a[4]=0.5;
    gr->SubPlot(2,2,0); gr->Title("ContV plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->ContV(a);

    gr->SubPlot(2,2,1); gr->Title("manual levels");
    gr->Rotate(50,60); gr->Box(); gr->ContV(v,a);

    gr->SubPlot(2,2,2); gr->Title("'_' style");
    gr->Rotate(50,60); gr->Box(); gr->ContV(a,"_");

    gr->SubPlot(2,2,3); gr->Title("ContV and ContF");
    gr->Rotate(50,60); gr->Box(); gr->Light(true);
    gr->ContV(a); gr->ContF(a); gr->Cont(a,"k");
    return 0;
}
```





### 2.6.16 Axial sample

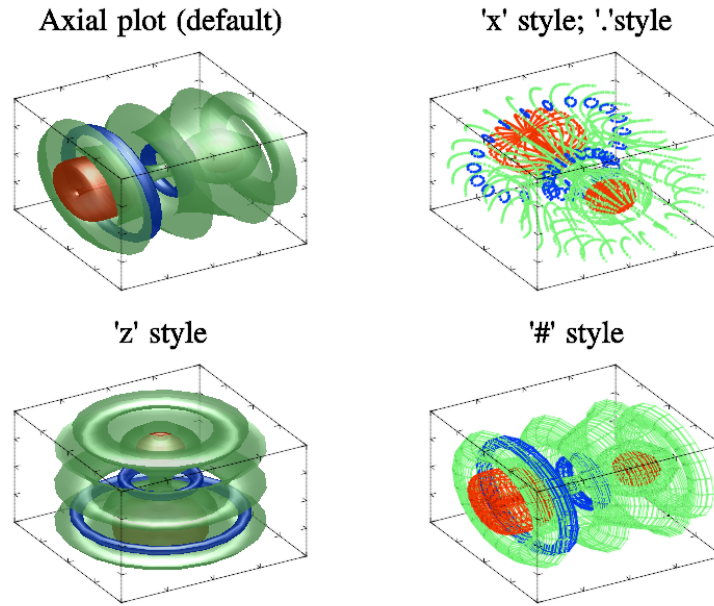
Function [axial], page 212, draw surfaces of rotation for contour lines. You can draw wire surfaces ('#' style) or ones rotated in other directions ('x', 'z' styles). The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a;  mgls_prepare2d(&a);
    gr->SubPlot(2,2,0); gr->Title("Axial plot (default)");
    gr->Light(true);  gr->Alpha(true);  gr->Rotate(50,60);
    gr->Box();  gr->Axial(a);

    gr->SubPlot(2,2,1); gr->Title("'x' style; '.' style"); gr->Rotate(50,60);
    gr->Box();  gr->Axial(a,"x.");

    gr->SubPlot(2,2,2); gr->Title("'z' style"); gr->Rotate(50,60);
    gr->Box();  gr->Axial(a,"z");

    gr->SubPlot(2,2,3); gr->Title("'\\#' style"); gr->Rotate(50,60);
    gr->Box();  gr->Axial(a,"#");
    return 0;
}
```

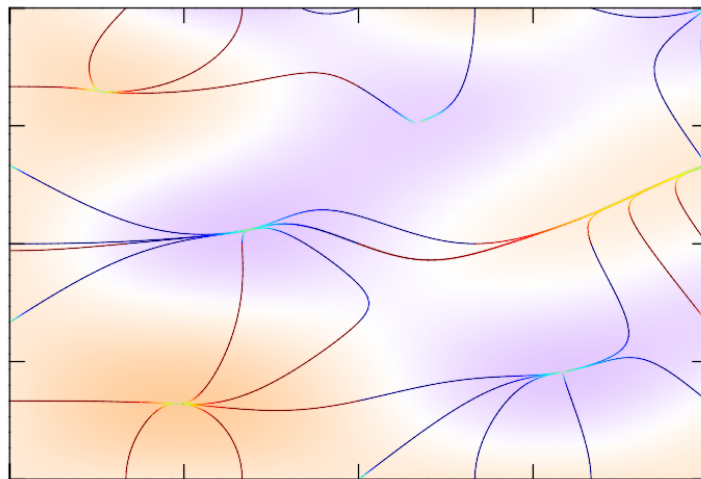


### 2.6.17 Grad sample

Function [grad], page 225, draw gradient lines for matrix. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a; mglS_prepare2d(&a);
    gr->SubPlot(1,1,0,""); gr->Title("Grad plot");
    gr->Box(); gr->Grad(a); gr->Dens(a,"{u8}w{q8}");
    return 0;
}
```

## Grad plot



## 2.7 3D samples

This section is devoted to visualization of 3D data arrays. 3D means the data which depend on 3 indexes (parameters) like tensor  $a(i,j,k)=a(x(i),y(j),x(k))$ ,  $i=1\dots n$ ,  $j=1\dots m$ ,  $k=1\dots l$  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

```
void mgl_prepare3d(mglData *a, mglData *b=0)
{
    register long i,j,k,n=61,m=50,l=40,i0;
    if(a) a->Create(n,m,l);    if(b) b->Create(n,m,l);
    mreal x,y,z;
    for(i=0;i<n;i++) for(j=0;j<m;j++) for(k=0;k<l;k++)
    {
        x=2*i/(n-1.)-1; y=2*j/(m-1.)-1; z=2*k/(l-1.)-1; i0 = i+n*(j+m*k);
        if(a) a->a[i0] = -2*(x*x + y*y + z*z*z*z - z*z - 0.1);
        if(b) b->a[i0] = 1-2*tanh((x+y)*(x+y));
    }
}

or using C functions

void mgl_prepare3d(HMDT a, HMDT b=0)
{
    register long i,j,k,n=61,m=50,l=40,i0;
    if(a) mgl_data_create(a,n,m,l);
    if(b) mgl_data_create(b,n,m,l);
    mreal x,y,z;
    for(i=0;i<n;i++) for(j=0;j<m;j++) for(k=0;k<l;k++)
    {
        x=2*i/(n-1.)-1; y=2*j/(m-1.)-1; z=2*k/(l-1.)-1; i0 = i+n*(j+m*k);
        if(a) mgl_data_set_value(a, -2*(x*x + y*y + z*z*z*z - z*z - 0.1), i,j,k);
        if(b) mgl_data_set_value(b, 1-2*tanh((x+y)*(x+y)), i,j,k);
    }
}
```

### 2.7.1 Surf3 sample

Function [surf3], page 213, 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:

```
int sample(mglGraph *gr)
{
    mglData c; mgl_prepare3d(&c);
    gr->Light(true);    gr->Alpha(true);
    gr->SubPlot(2,2,0); gr->Title("Surf3 plot (default)");
    gr->Rotate(50,60);  gr->Box();  gr->Surf3(c);

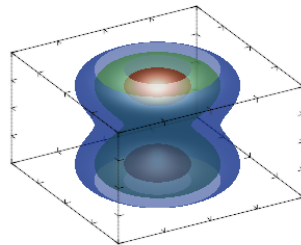
    gr->SubPlot(2,2,1); gr->Title("'\\#' style");
    gr->Rotate(50,60);  gr->Box();  gr->Surf3(c,"#");
}
```

```

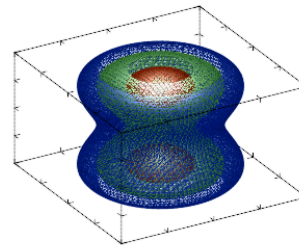
gr->SubPlot(2,2,2); gr->Title("'.' style");
gr->Rotate(50,60); gr->Box(); gr->Surf3(c,".");
return 0;
}

```

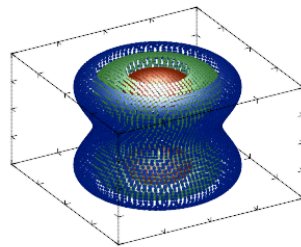
Surf3 plot (default)



'#' style



'.' style



### 2.7.2 Surf3C sample

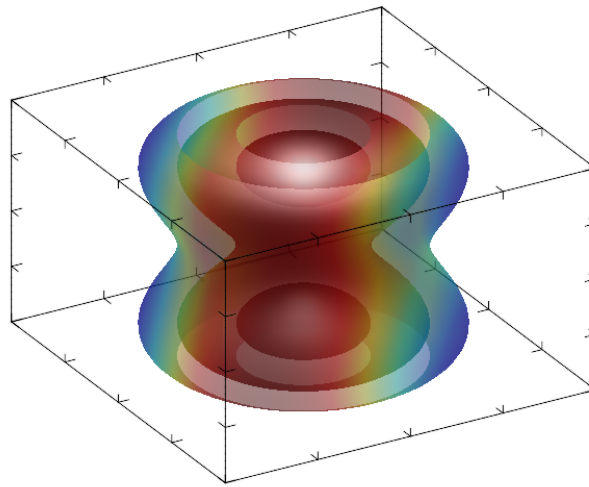
Function [surf3c], page 217, is similar to [surf3], page 213, but its coloring is determined by another data. The sample code is:

```

int sample(mglGraph *gr)
{
    mglData c,d; mgl_prepare3d(&c,&d);
    gr->Title("Surf3C plot"); gr->Rotate(50,60);
    gr->Light(true); gr->Alpha(true);
    gr->Box(); gr->Surf3C(c,d);
    return 0;
}

```

## Surf3C plot

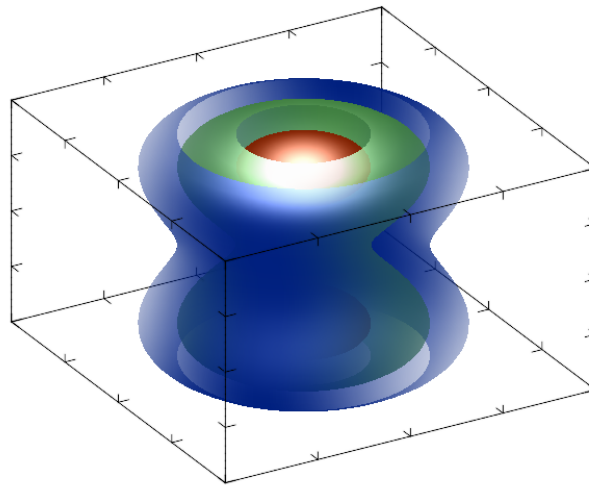


### 2.7.3 Surf3A sample

Function [surf3a], page 218, is similar to [surf3], page 213, but its transparency is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c,d;  mglS_prepare3d(&c,&d);
    gr->Title("Surf3A plot"); gr->Rotate(50,60);
    gr->Light(true);  gr->Alpha(true);
    gr->Box();  gr->Surf3A(c,d);
    return 0;
}
```

## Surf3A plot

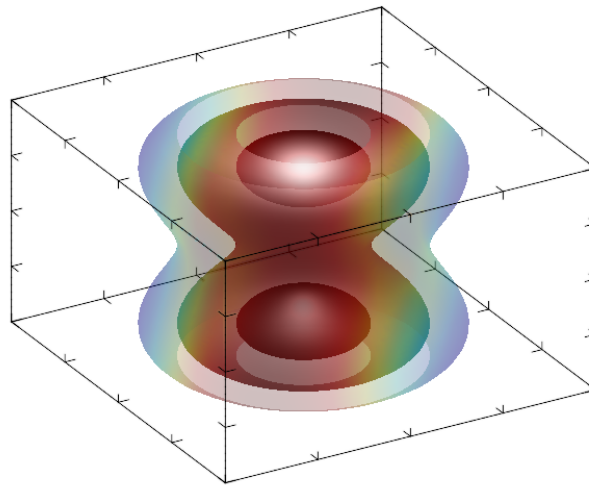


### 2.7.4 Surf3CA sample

Function [surf3ca], page 220, is similar to [surf3], page 213, but its color and transparency is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c,d;  mglS_prepare3d(&c,&d);
    gr->Title("Surf3CA plot"); gr->Rotate(50,60);
    gr->Light(true);  gr->Alpha(true);
    gr->Box();  gr->Surf3CA(c,d,c);
    return 0;
}
```

## Surf3CA plot



### 2.7.5 Cloud sample

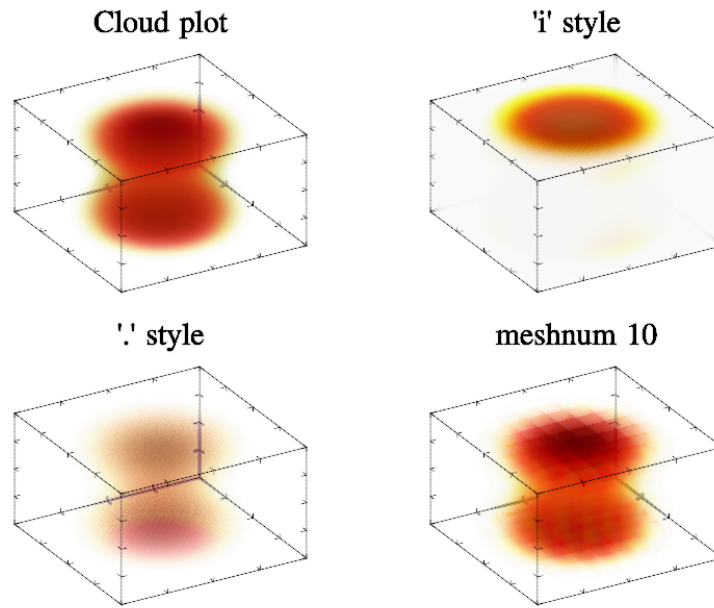
Function [cloud], page 214, draw cloud-like object which is less transparent for higher data values. Similar plot can be created using many (about 10-20) `Surf3A(a,a)` isosurfaces. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c;  mglS_prepare3d(&c);
    gr->SubPlot(2,2,0); gr->Title("Cloud plot");
    gr->Rotate(50,60);  gr->Alpha(true);
    gr->Box();  gr->Cloud(c,"wyrRk");

    gr->SubPlot(2,2,1); gr->Title("'i' style");
    gr->Rotate(50,60);  gr->Box();  gr->Cloud(c,"iwyrRk");

    gr->SubPlot(2,2,2); gr->Title("'.' style");
    gr->Rotate(50,60);  gr->Box();  gr->Cloud(c,".wyrRk");

    gr->SubPlot(2,2,3); gr->Title("meshnum 10");
    gr->Rotate(50,60);  gr->Box();  gr->Cloud(c,"wyrRk","meshnum 10");
    return 0;
}
```



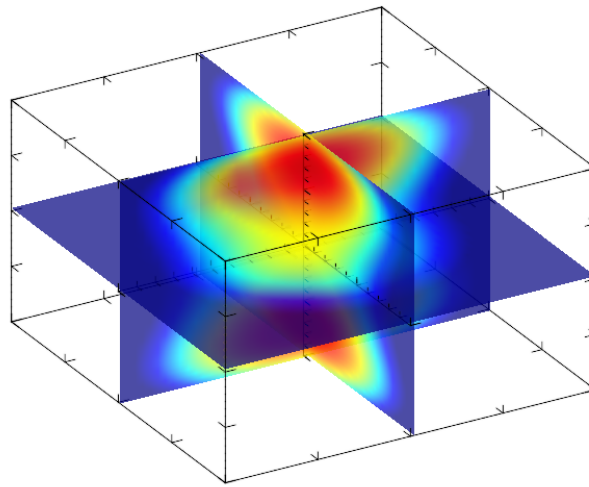
### 2.7.6 Dens3 sample

Function [dens3], page 214, draw just usual density plot but at slices of 3D data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c; mgl_prepare3d(&c);
    gr->Title("Dens3 sample"); gr->Rotate(50,60);
    gr->Alpha(true); gr->SetAlphaDef(0.7);
    gr->SetOrigin(0,0,0); gr->Axis("_xyz"); gr->Box();
    gr->Dens3(c,"x"); gr->Dens3(c); gr->Dens3(c,"z");
    return 0;
}
```



## Dens3 sample

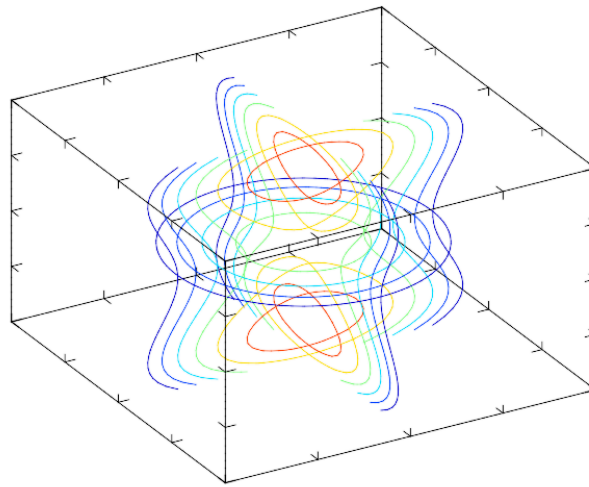


### 2.7.7 Cont3 sample

Function [cont3], page 215, draw just usual contour lines but at slices of 3D data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c; mgl_prepare3d(&c);
    gr->Title("Cont3 sample"); gr->Rotate(50,60);
    gr->Alpha(true); gr->SetAlphaDef(0.7);
    gr->SetOrigin(0,0,0); gr->Axis("_xyz"); gr->Box();
    gr->Cont3(c,"x"); gr->Cont3(c); gr->Cont3(c,"z");
    return 0;
}
```

## Cont3 sample

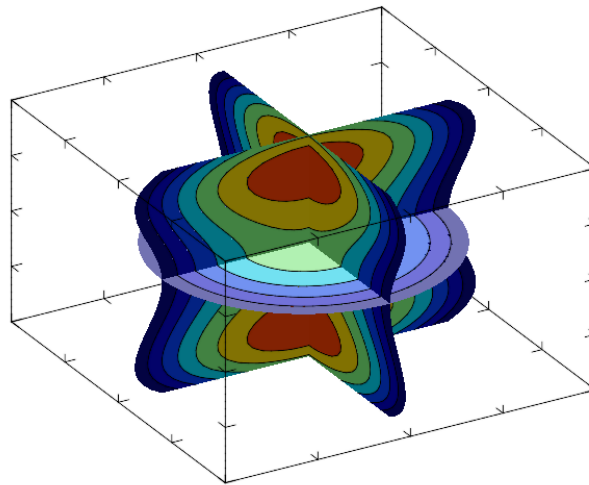


### 2.7.8 ContF3 sample

Function [contf3], page 215, draw just usual filled contours but at slices of 3D data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c; mglS_prepare3d(&c);
    gr->Title("ContF3 sample"); gr->Rotate(50,60);
    gr->Alpha(true); gr->SetAlphaDef(0.7);
    gr->SetOrigin(0,0,0); gr->Axis("_xyz"); gr->Box();
    gr->ContF3(c,"x"); gr->ContF3(c); gr->ContF3(c,"z");
    gr->Cont3(c,"kx"); gr->Cont3(c,"k"); gr->Cont3(c,"kz");
    return 0;
}
```

## ContF3 sample

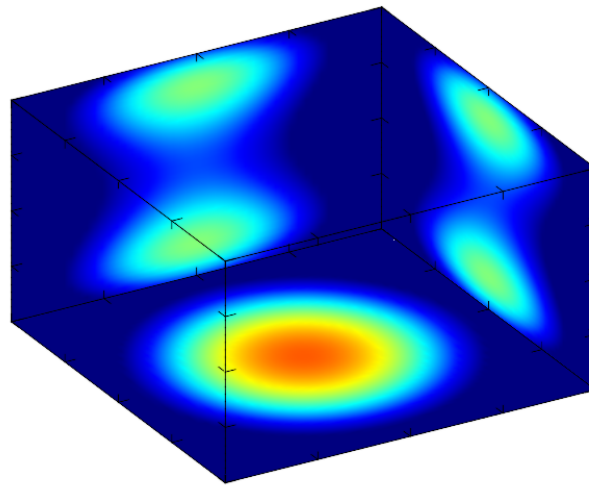


### 2.7.9 Dens projection sample

Functions [densz], page 227, [densy], page 227, [densx], page 227, draw density plot on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c;  mgl_prepare3d(&c);
    gr->Title("Dens[XYZ] sample");  gr->Rotate(50,60);
    gr->Box();  gr->DensX(c.Sum("x"),0,-1);
    gr->DensY(c.Sum("y"),0,1);  gr->DensZ(c.Sum("z"),0,-1);
    return 0;
}
```

## Dens[XYZ] sample

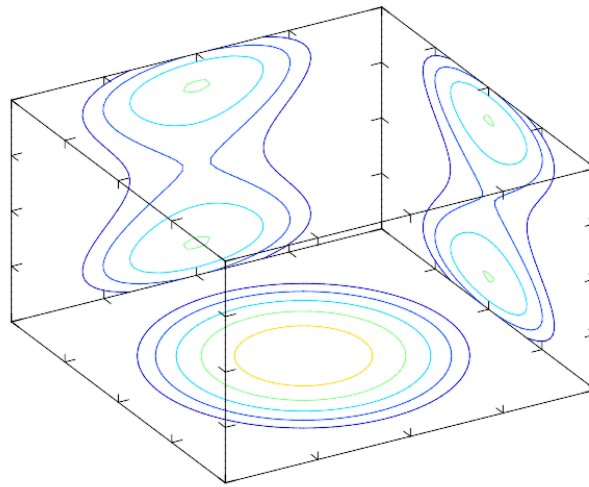


### 2.7.10 Cont projection sample

Functions [contz], page 227, [conty], page 227, [contx], page 227, draw contour lines on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c;  mglS_prepare3d(&c);
    gr->Title("Cont[XYZ] sample");  gr->Rotate(50,60);
    gr->Box();  gr->ContX(c.Sum("x"),"",-1);
    gr->ContY(c.Sum("y"),"",1);  gr->ContZ(c.Sum("z"),"",-1);
    return 0;
}
```

## Cont[XYZ] sample

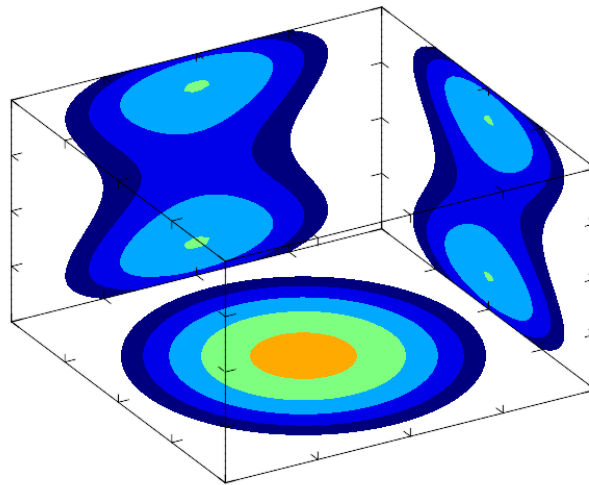


### 2.7.11 ContF projection sample

Functions [contfz], page 228, [contfy], page 228, [contfx], page 228, draw filled contours on plane perpendicular to corresponding axis. One of possible application is drawing projections of 3D field. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c;  mgl_prepare3d(&c);
    gr->Title("Cont[XYZ] sample");  gr->Rotate(50,60);
    gr->Box();  gr->ContFX(c.Sum("x"),"",-1);
    gr->ContFY(c.Sum("y"),"",1);  gr->ContFZ(c.Sum("z"),"",-1);
    return 0;
}
```

## ContF[XYZ] sample



### 2.7.12 TriPlot and QuadPlot

Function [triplot], page 230, and [quadplot], page 231, 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:

```
int sample(mglGraph *gr)
{
    mreal 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};
    mreal xc[] = {-1,1,-1,1,-1,1,-1,1}, yc[] = {-1,-1,1,1,-1,-1,1,1}, zc[] = {-1,-1,-1,-1,1,1,1,1};
    mglData qq(6,4,q), xx(8,xc), yy(8,yc), zz(8,zc);
    gr->Light(true); //gr->Alpha(true);
    gr->SubPlot(2,2,0); gr->Title("QuadPlot sample"); gr->Rotate(50,60);
    gr->QuadPlot(qq,xx,yy,zz,"yr");
    gr->QuadPlot(qq,xx,yy,zz,"k#");
    gr->SubPlot(2,2,2); gr->Title("QuadPlot coloring"); gr->Rotate(50,60);
    gr->QuadPlot(qq,xx,yy,zz,yy,"yr");
    gr->QuadPlot(qq,xx,yy,zz,"k#");

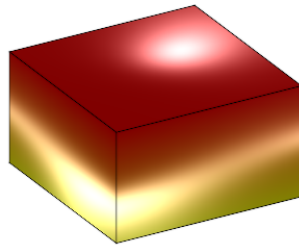
    mreal t[] = {0,1,2, 0,1,3, 0,2,3, 1,2,3};
    mreal xt[] = {-1,1,0,0}, yt[] = {-1,-1,1,0}, zt[] = {-1,-1,-1,1};
    mglData tt(4,3,t), uu(4,xt), vv(4,yt), ww(4,zt);
    gr->SubPlot(2,2,1); gr->Title("TriPlot sample"); gr->Rotate(50,60);
    gr->TriPlot(tt,uu,vv,ww,"b");
    gr->TriPlot(tt,uu,vv,ww,"k#");
    gr->SubPlot(2,2,3); gr->Title("TriPlot coloring"); gr->Rotate(50,60);
    gr->TriPlot(tt,uu,vv,ww,vv,"cb");
    gr->TriPlot(tt,uu,vv,ww,"k#");
    gr->TriCont(tt,uu,vv,ww,"B");
}
```

```

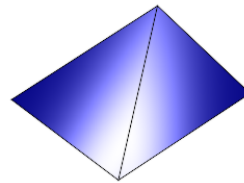
    return 0;
}

```

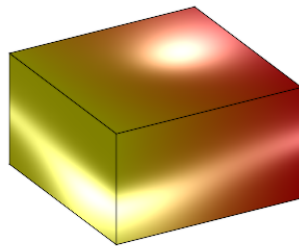
QuadPlot sample



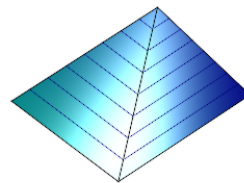
TriPlot sample



QuadPlot coloring



TriPlot coloring



### 2.7.13 Dots sample

Function [dots], page 231, is another way to draw irregular points. **Dots** use color scheme for coloring (see Section 3.4 [Color scheme], page 149). The sample code is:

```

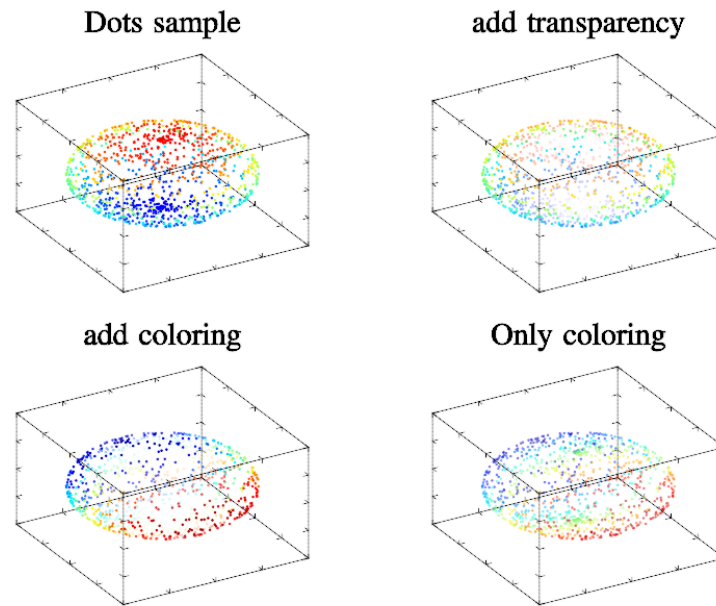
int sample(mglGraph *gr)
{
    int i, n=2000;
    mglData x(n),y(n),z(n),c(n);
    for(i=0;i<n;i++)
    {
        mreal t=M_PI*(mgl_rnd()-0.5), f=2*M_PI*mgl_rnd();
        x.a[i] = 0.9*cos(t)*cos(f);
        y.a[i] = 0.9*cos(t)*sin(f);
        z.a[i] = 0.6*sin(t);
        c.a[i] = cos(2*t);
    }
    gr->SubPlot(2,2,0); gr->Title("Dots sample"); gr->Rotate(50,60);
    gr->Box(); gr->Dots(x,y,z);
    gr->Alpha(true);
    gr->SubPlot(2,2,1); gr->Title("add transparency"); gr->Rotate(50,60);
    gr->Box(); gr->Dots(x,y,z,c);
    gr->SubPlot(2,2,2); gr->Title("add coloring"); gr->Rotate(50,60);
    gr->Box(); gr->Dots(x,y,z,x,c);
    gr->SubPlot(2,2,3); gr->Title("Only coloring");gr->Rotate(50,60);
    gr->Box(); gr->Tens(x,y,z,x,".");
}

```

```

    return 0;
}

```



### 2.7.14 IFS sample

Commands [ifs2d], page 281, and [ifs3d], page 281, generate points for fractals using iterated function system in 2d and 3d cases correspondingly. The sample codes are:

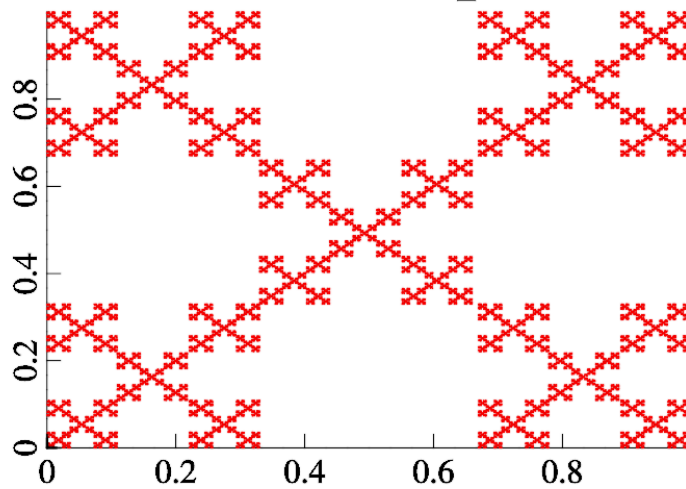
```

int sample(mglGraph *gr)
{
    mglData A;
    A.SetList(35, 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.,0.33,0.3
                0.33,0.,0.,0.33,0.,0.67,0.2, 0.33,0.,0.,0.33,0.67,0.67,0.2);
    A.Rearrange(7);
    mglData f(mglIFS2d(A,100000));
    gr->SubPlot(1,1,0,"<_");
    gr->Title("IFS 2d sample");
    gr->SetRanges(f.SubData(0), f.SubData(1));
    gr->Axis(); gr->Plot(f.SubData(0), f.SubData(1),"r#o ","size 0.05");
}

```

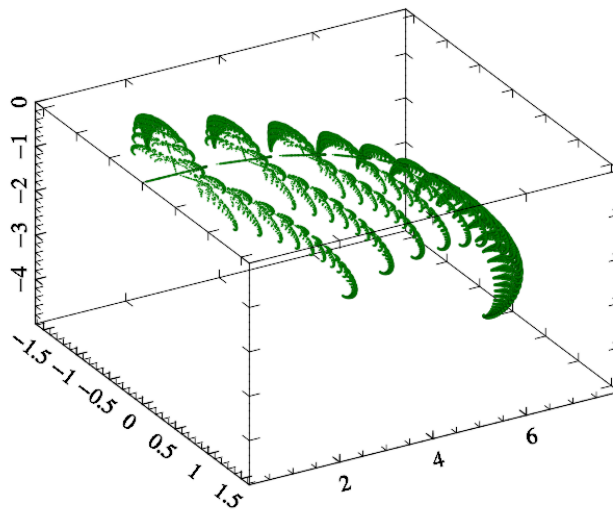


## IFS 2d sample



```
int sample(mglGraph *gr)
{
    mglData A;
    A.SetList(52, 0.,0.,0.,0.,.18,0.,0.,0.,0.,0.,0.,0.,.01, .85,0.,0.,0.,.85,.1,0.,-0.1,0.85,
        .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.
    A.Rearrange(13);
    mglData f(mglIFS3d(A,100000));
    gr->Title("IFS 3d sample");
    gr->SetRanges(f.SubData(0), f.SubData(1), f.SubData(2));
    gr->Rotate(50,60); gr->Axis(); gr->Box();
    gr->Dots(f.SubData(0), f.SubData(1), f.SubData(2),"G#o","size 0.05");
}
```

## IFS 3d sample



## 2.8 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

```
void mgl_prepare2v(mglData *a, mglData *b)
{
    register long i,j,n=20,m=30,i0;
    if(a) a->Create(n,m);    if(b) b->Create(n,m);
    mreal x,y;
    for(i=0;i<n;i++)    for(j=0;j<m;j++)
    {
        x=i/(n-1.); y=j/(m-1.); i0 = i+n*j;
        if(a) a->a[i0] = 0.6*sin(2*M_PI*x)*sin(3*M_PI*y)+0.4*cos(3*M_PI*x*y);
        if(b) b->a[i0] = 0.6*cos(2*M_PI*x)*cos(3*M_PI*y)+0.4*cos(3*M_PI*x*y);
    }
}

void mgl_prepare3v(mglData *ex, mglData *ey, mglData *ez)
{
    register long i,j,k,n=10,i0;
    if(!ex || !ey || !ez) return;
```

```

ex->Create(n,n,n); ey->Create(n,n,n); ez->Create(n,n,n);
mreal x,y,z, r1,r2;
for(i=0;i<n;i++) for(j=0;j<n;j++) for(k=0;k<n;k++)
{
    x=2*i/(n-1.)-1; y=2*j/(n-1.)-1; z=2*k/(n-1.)-1; i0 = i+n*(j+k*n);
    r1 = pow(x*x+y*y+(z-0.3)*(z-0.3)+0.03,1.5);
    r2 = pow(x*x+y*y+(z+0.3)*(z+0.3)+0.03,1.5);
    ex->a[i0]=0.2*x/r1 - 0.2*x/r2;
    ey->a[i0]=0.2*y/r1 - 0.2*y/r2;
    ez->a[i0]=0.2*(z-0.3)/r1 - 0.2*(z+0.3)/r2;
}
}

```

or using C functions

```

void mgls_prepare2v(HMDT a, HMDT b)
{
    register long i,j,n=20,m=30,i0;
    if(a) mgl_data_create(a,n,m,1);
    if(b) mgl_data_create(b,n,m,1);
    mreal x,y;
    for(i=0;i<n;i++) for(j=0;j<m;j++)
    {
        x=i/(n-1.); y=j/(m-1.); i0 = i+n*j;
        if(a) mgl_data_set_value(a, 0.6*sin(2*M_PI*x)*sin(3*M_PI*y)+0.4*cos(3*M_PI*x*y), i,j,0);
        if(b) mgl_data_set_value(b, 0.6*cos(2*M_PI*x)*cos(3*M_PI*y)+0.4*cos(3*M_PI*x*y), i,j,0);
    }
}

void mgls_prepare3v(HMDT ex, HMDT ey, HMDT ez)
{
    register long i,j,k,n=10,i0;
    if(!ex || !ey || !ez) return;
    mgl_data_create(ex,n,n,n);
    mgl_data_create(ey,n,n,n);
    mgl_data_create(ez,n,n,n);
    mreal x,y,z, r1,r2;
    for(i=0;i<n;i++) for(j=0;j<n;j++) for(k=0;k<n;k++)
    {
        x=2*i/(n-1.)-1; y=2*j/(n-1.)-1; z=2*k/(n-1.)-1; i0 = i+n*(j+k*n);
        r1 = pow(x*x+y*y+(z-0.3)*(z-0.3)+0.03,1.5);
        r2 = pow(x*x+y*y+(z+0.3)*(z+0.3)+0.03,1.5);
        mgl_data_set_value(ex, 0.2*x/r1 - 0.2*x/r2, i,j,k);
        mgl_data_set_value(ey, 0.2*y/r1 - 0.2*y/r2, i,j,k);
        mgl_data_set_value(ez, 0.2*(z-0.3)/r1 - 0.2*(z+0.3)/r2, i,j,k);
    }
}

```

### 2.8.1 Vect sample

Function [vect], page 222, 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 3.4 [Color scheme], page 149). The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mgl_prepare2v(&a,&b);
    gr->SubPlot(3,2,0,""); gr->Title("Vect plot (default)");
    gr->Box();  gr->Vect(a,b);

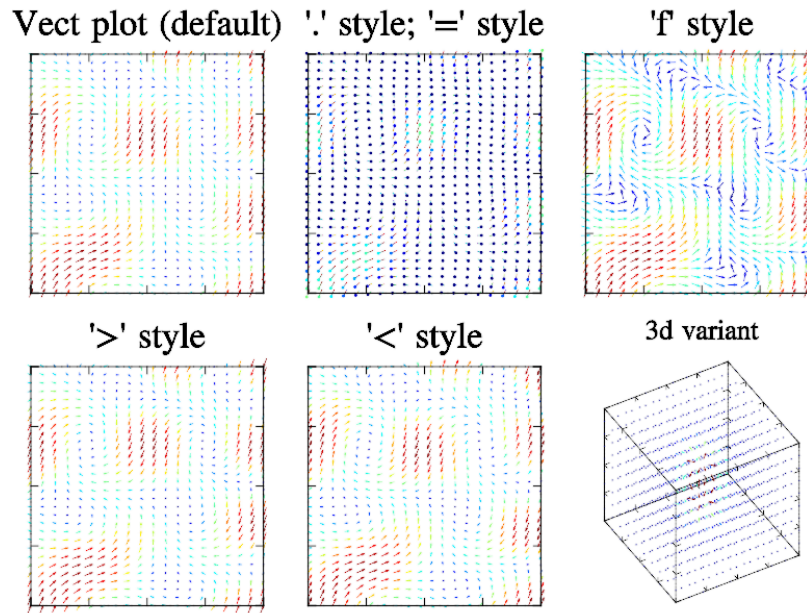
    gr->SubPlot(3,2,1,""); gr->Title("'.' style; '=' style");
    gr->Box();  gr->Vect(a,b,"=");

    gr->SubPlot(3,2,2,""); gr->Title("'f' style");
    gr->Box();  gr->Vect(a,b,"f");

    gr->SubPlot(3,2,3,""); gr->Title(">' style");
    gr->Box();  gr->Vect(a,b,">");

    gr->SubPlot(3,2,4,""); gr->Title("<' style");
    gr->Box();  gr->Vect(a,b,"<");

    mglData ex,ey,ez; mgl_prepare3v(&ex,&ey,&ez);
    gr->SubPlot(3,2,5); gr->Title("3d variant"); gr->Rotate(50,60);
    gr->Box();  gr->Vect(ex,ey,ez);
    return 0;
}
```

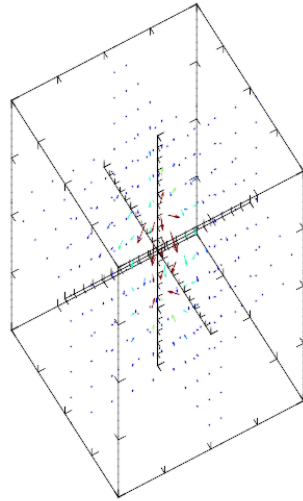
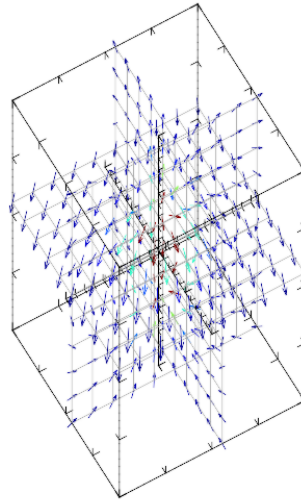


### 2.8.2 Vect3 sample

Function [vect3], page 223, draw just usual vector field plot but at slices of 3D data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData ex,ey,ez; mgl_prepare3v(&ex,&ey,&ez);
    gr->SubPlot(2,1,0); gr->Title("Vect3 sample");    gr->SetOrigin(0,0,0);
    gr->Rotate(50,60);    gr->Axis("_xyz");    gr->Box();
    gr->Vect3(ex,ey,ez,"x");    gr->Vect3(ex,ey,ez);    gr->Vect3(ex,ey,ez,"z");

    gr->SubPlot(2,1,1);    gr->Title("'f' style");
    gr->Rotate(50,60);    gr->Axis("_xyz");    gr->Box();
    gr->Vect3(ex,ey,ez,"fx"); gr->Vect3(ex,ey,ez,"f");gr->Vect3(ex,ey,ez,"fz");
    gr->Grid3(ex,"Wx");    gr->Grid3(ex,"W");    gr->Grid3(ex,"Wz");
    return 0;
}
```

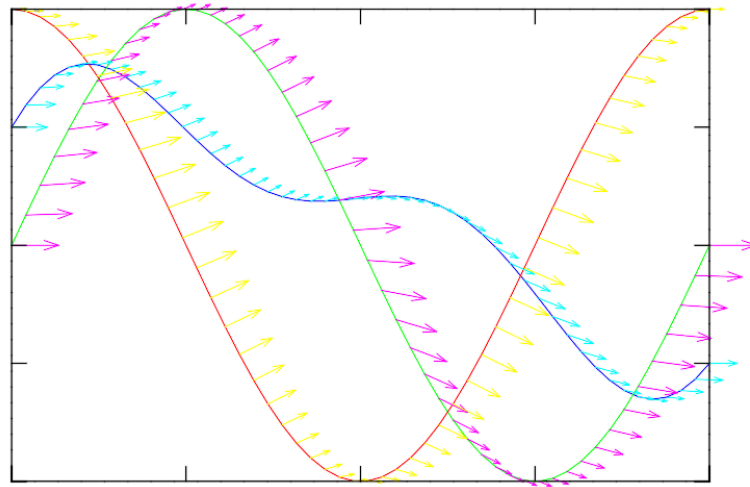
**Vect3 sample****'f' style**

### 2.8.3 Traj sample

Function [traj], page 222, is 1D analogue of Vect. It draw vectors from specified points. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData x,y,y1,y2;  mgl_prepare1d(&y,&y1,&y2,&x);
    gr->SubPlot(1,1,0,""); gr->Title("Traj plot");
    gr->Box();  gr->Plot(x,y);  gr->Traj(x,y,y1,y2);
    return 0;
}
```

# Traj plot



## 2.8.4 Flow sample

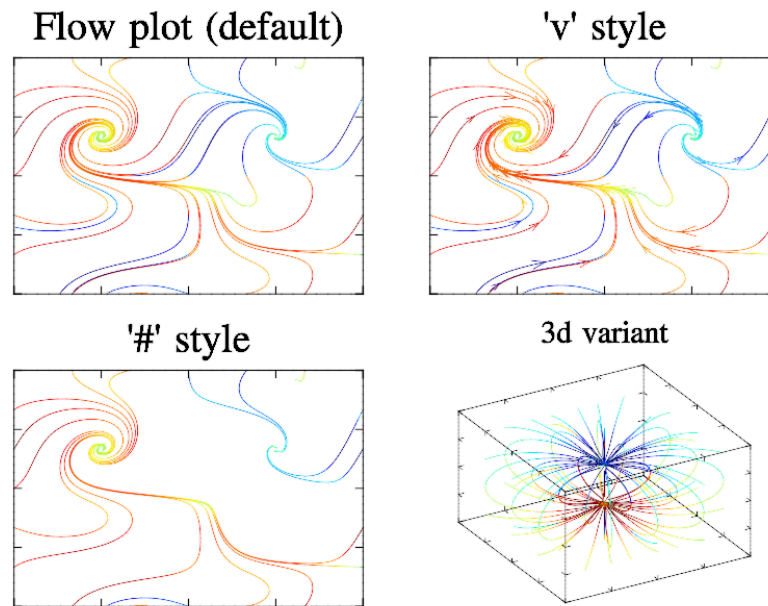
Function [flow], page 224, is another standard way to visualize vector fields – it draw lines (threads) which is tangent to local vector field direction. MathGL draw threads from edges of bounding box and from central slices. Sometimes it is not most appropriate variant – you may want to use FlowP to specify manual position of threads. Flow use color scheme for coloring (see Section 3.4 [Color scheme], page 149). At this warm color corresponds to normal flow (like attractor), cold one corresponds to inverse flow (like source). The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mgl_prepare2v(&a,&b);
    gr->SubPlot(2,2,0,""); gr->Title("Flow plot (default)");
    gr->Box();  gr->Flow(a,b);

    gr->SubPlot(2,2,1,"");  gr->Title("'v' style");
    gr->Box();  gr->Flow(a,b,"v");

    gr->SubPlot(2,2,2,"");  gr->Title("'\\#' style");
    gr->Box();  gr->Flow(a,b,"#");

    mglData ex,ey,ez; mgl_prepare3v(&ex,&ey,&ez);
    gr->SubPlot(2,2,3); gr->Title("3d variant");  gr->Rotate(50,60);
    gr->Box();  gr->Flow(ex,ey,ez);
    return 0;
}
```



### 2.8.5 Pipe sample

Function [pipe], page 226, is similar to [flow], page 224, but draw pipes (tubes) which radius is proportional to the amplitude of vector field. **Pipe** use color scheme for coloring (see Section 3.4 [Color scheme], page 149). At this warm color corresponds to normal flow (like attractor), cold one corresponds to inverse flow (like source). The sample code is:

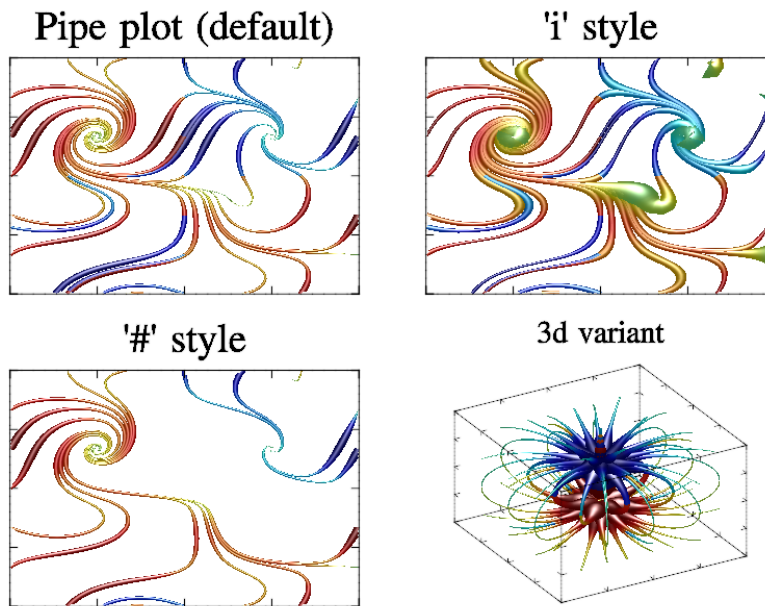
```
int sample(mglGraph *gr)
{
    mglData a,b; mgl_prepare2v(&a,&b);
    gr->SubPlot(2,2,0,""); gr->Title("Pipe plot (default)");
    gr->Light(true); gr->Box(); gr->Pipe(a,b);

    gr->SubPlot(2,2,1,""); gr->Title("'i' style");
    gr->Box(); gr->Pipe(a,b,"i");

    gr->SubPlot(2,2,2,""); gr->Title("'\\#' style");
    gr->Box(); gr->Pipe(a,b,"#");

    mglData ex,ey,ez; mgl_prepare3v(&ex,&ey,&ez);
    gr->SubPlot(2,2,3); gr->Title("3d variant"); gr->Rotate(50,60);
    gr->Box(); gr->Pipe(ex,ey,ez,"",0.1);
    return 0;
}
```



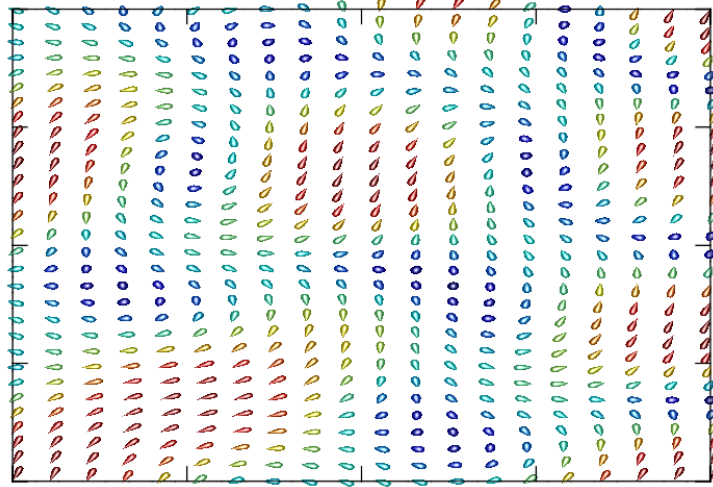


### 2.8.6 Dew sample

Function [dew], page 224, is similar to `Vect` but use drops instead of arrows. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mglS_prepare2v(&a,&b);
    gr->SubPlot(1,1,0,""); gr->Title("Dew plot");
    gr->Box();  gr->Light(true);  gr->Dew(a,b);
    return 0;
}
```

# Dew plot



## 2.9 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.

### 2.9.1 “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 `surf` 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:

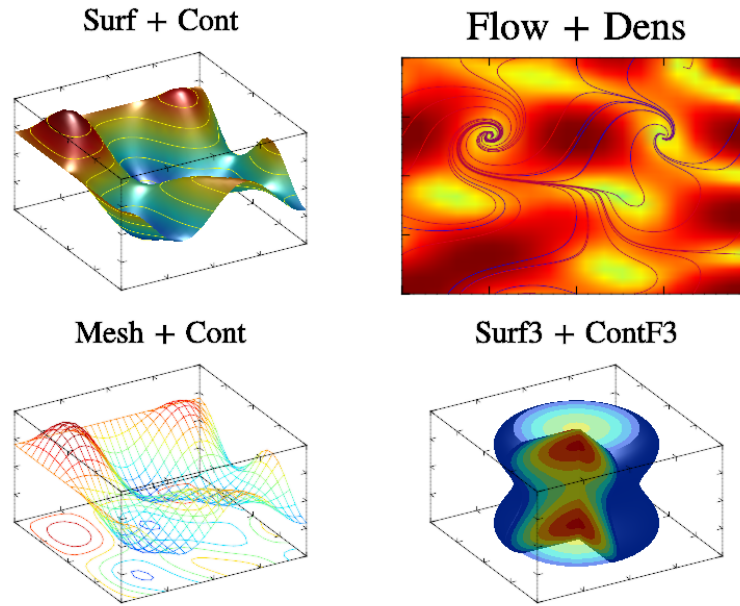
```
int sample(mglGraph *gr)
{
    mglData a,b,d;  mglS_prepare2v(&a,&b);  d = a;
    for(int i=0;i<a.nx*a.ny;i++)  d.a[i] = hypot(a.a[i],b.a[i]);
    mglData c;  mglS_prepare3d(&c);
    mglData v(10);  v.Fill(-0.5,1);

    gr->SubPlot(2,2,1,"");  gr->Title("Flow + Dens");
    gr->Flow(a,b,"br");  gr->Dens(d,"BbcyrR");  gr->Box();

    gr->SubPlot(2,2,0);  gr->Title("Surf + Cont");  gr->Rotate(50,60);
    gr->Light(true);  gr->Surf(a);  gr->Cont(a,"y");  gr->Box();

    gr->SubPlot(2,2,2);  gr->Title("Mesh + Cont");  gr->Rotate(50,60);
    gr->Box();  gr->Mesh(a);  gr->Cont(a,"_");

    gr->SubPlot(2,2,3);  gr->Title("Surf3 + ContF3");gr->Rotate(50,60);
    gr->Box();  gr->ContF3(v,c,"z",0);  gr->ContF3(v,c,"x");  gr->ContF3(v,c);
    gr->SetCutBox(mglPoint(0,-1,-1), mglPoint(1,0,1.1));
    gr->ContF3(v,c,"z",c.nz-1);  gr->Surf3(-0.5,c);
    return 0;
}
```



### 2.9.2 Transparency and lighting

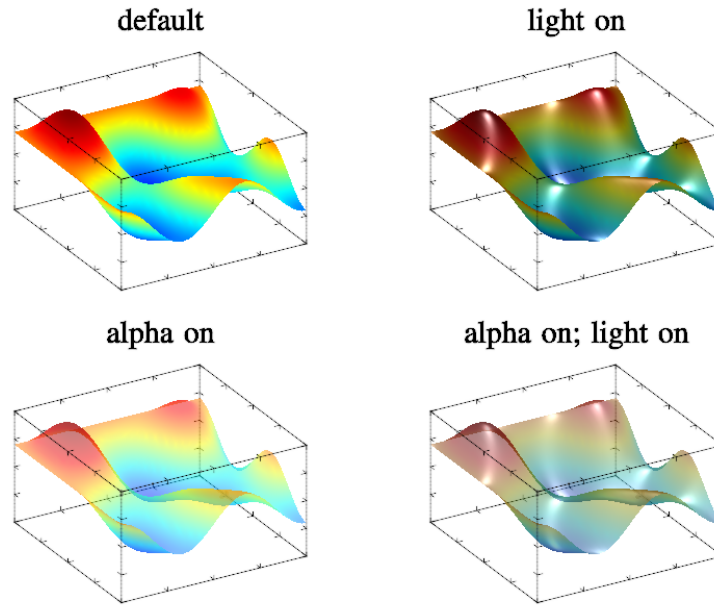
Here I want to show how transparency and lighting both and separately change the look of a surface. So, there is code and picture for that:

```
int sample(mglGraph *gr)
{
    mglData a;  mglS_prepare2d(&a);
    gr->SubPlot(2,2,0); gr->Title("default"); gr->Rotate(50,60);
    gr->Box();  gr->Surf(a);

    gr->SubPlot(2,2,1); gr->Title("light on");  gr->Rotate(50,60);
    gr->Box();  gr->Light(true);  gr->Surf(a);

    gr->SubPlot(2,2,3); gr->Title("alpha on; light on");  gr->Rotate(50,60);
    gr->Box();  gr->Alpha(true);  gr->Surf(a);

    gr->SubPlot(2,2,2); gr->Title("alpha on");  gr->Rotate(50,60);
    gr->Box();  gr->Light(false); gr->Surf(a);
    return 0;
}
```



### 2.9.3 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 function `[alpha]`, page 158. 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 3.7 [Command options], page 153). 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.

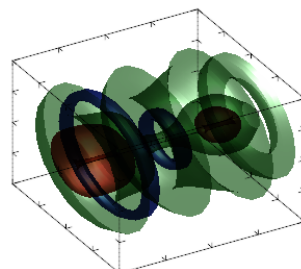
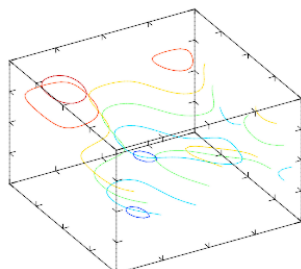
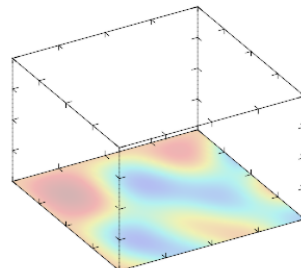
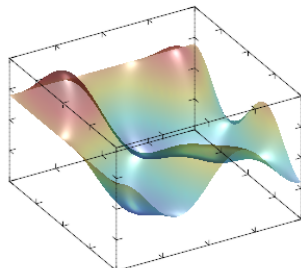
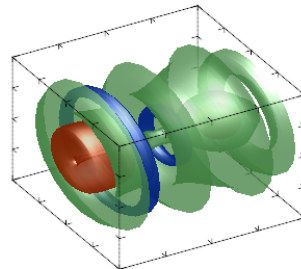
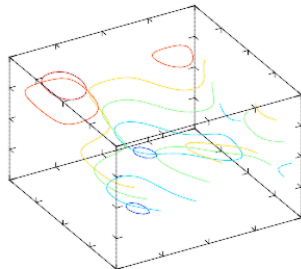
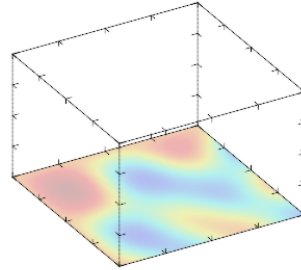
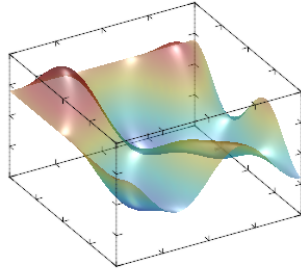
As example I shall show several plots for different types of transparency. The code is the same except the values of `SetTranspType` function:

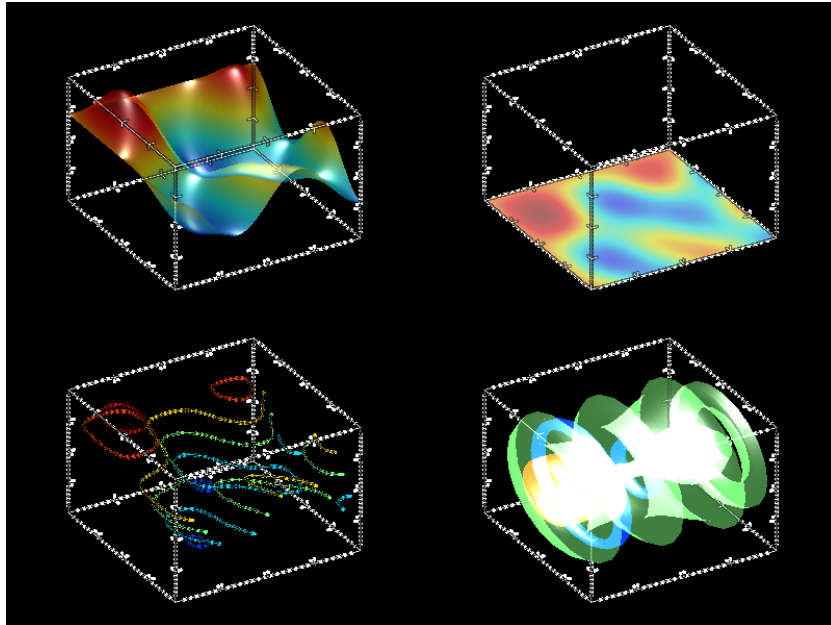
```
int sample(mglGraph *gr)
{
    gr->Alpha(true); gr->Light(true);
    mglData a; mgl_s_prepare2d(&a);
    gr->SetTranspType(0); gr->Clf();
    gr->SubPlot(2,2,0); gr->Rotate(50,60); gr->Surf(a); gr->Box();
```

```

gr->SubPlot(2,2,1); gr->Rotate(50,60); gr->Dens(a); gr->Box();
gr->SubPlot(2,2,2); gr->Rotate(50,60); gr->Cont(a); gr->Box();
gr->SubPlot(2,2,3); gr->Rotate(50,60); gr->Axial(a); gr->Box();
return 0;
}

```





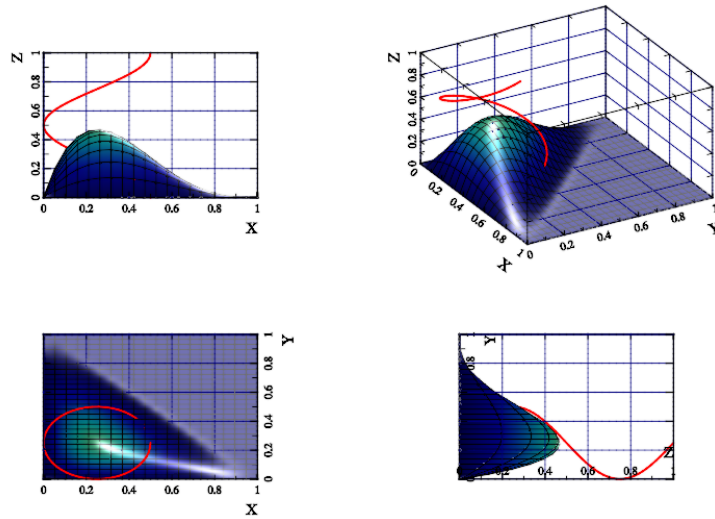
### 2.9.4 Axis projection

You can easily make 3D plot and draw its x-,y-,z-projections (like in CAD) by using [ternary], page 168, function with arguments: 4 for Cartesian, 5 for Ternary and 6 for Quaternary coordinates. The sample code is:

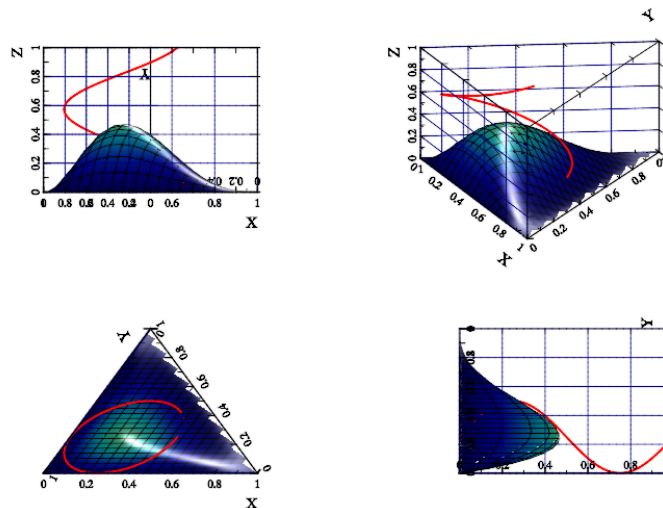
```
int sample(mglGraph *gr)
{
    gr->SetRanges(0,1,0,1,0,1);
    mglData x(50),y(50),z(50),rx(10),ry(10), a(20,30);
    a.Modify("30*x*y*(1-x-y)^2*(x+y<1)");
    x.Modify("0.25*(1+cos(2*pi*x))");
    y.Modify("0.25*(1+sin(2*pi*x))");
    rx.Modify("rnd"); ry.Modify("(1-v)*rnd",rx);
    z.Modify("x");

    gr->Title("Projection sample");
    gr->Ternary(4);
    gr->Rotate(50,60);      gr->Light(true);
    gr->Plot(x,y,z,"r2");   gr->Surf(a,"#");
    gr->Axis(); gr->Grid(); gr->Box();
    gr->Label('x',"X",1);   gr->Label('y',"Y",1);   gr->Label('z',"Z",1);
}
```

## Projection sample



## Projection sample (ternary)



### 2.9.5 Adding fog

MathGL can add a fog to the image. Its switching on is rather simple – just use [fog], page 159, function. There is the only feature – fog is applied for whole image. Not to particular subplot. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a; mglS_prepare2d(&a);
    gr->Title("Fog sample");
    gr->Light(true); gr->Rotate(50,60); gr->Fog(1); gr->Box();
```

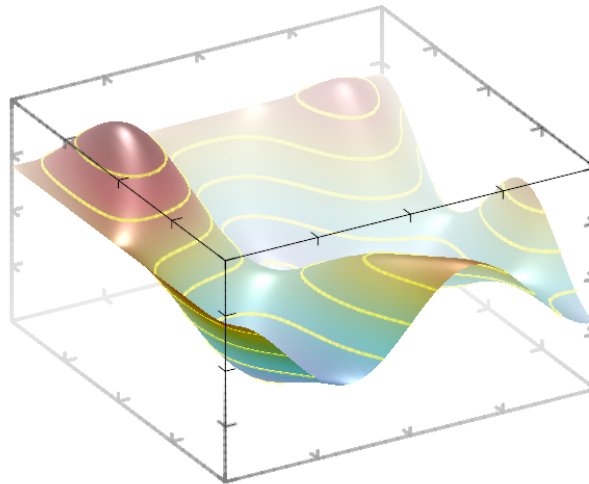


```

gr->Surf(a);  gr->Cont(a,"y");
return 0;
}

```

## Fog sample



### 2.9.6 Lighting sample

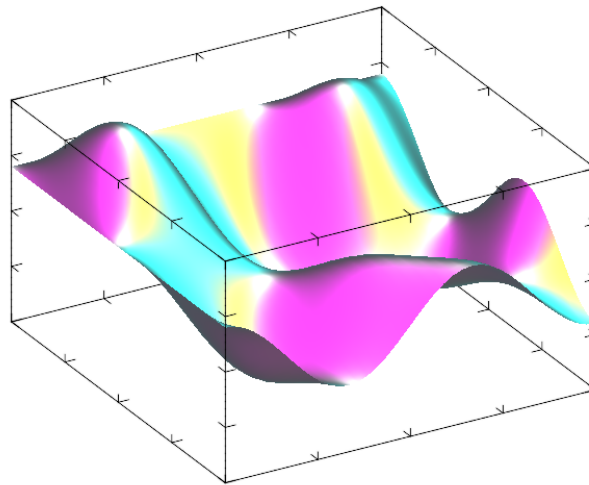
In contrast to the most of other programs, MathGL supports several (up to 10) light sources. Moreover, the color each of them can be different: white (this is usual), yellow, red, cyan, green and so on. The use of several light sources may be interesting for the highlighting of some peculiarities of the plot or just to make an amusing picture. Note, each light source can be switched on/off individually. The sample code is:

```

int sample(mglGraph *gr)
{
    mglData a;  mgl_prepare2d(&a);
    gr->Title("Several light sources");
    gr->Rotate(50,60);  gr->Light(true);
    gr->AddLight(1,mglPoint(0,1,0),'c');
    gr->AddLight(2,mglPoint(1,0,0),'y');
    gr->AddLight(3,mglPoint(0,-1,0),'m');
    gr->Box();  gr->Surf(a,"h");
    return 0;
}

```

## Several light sources



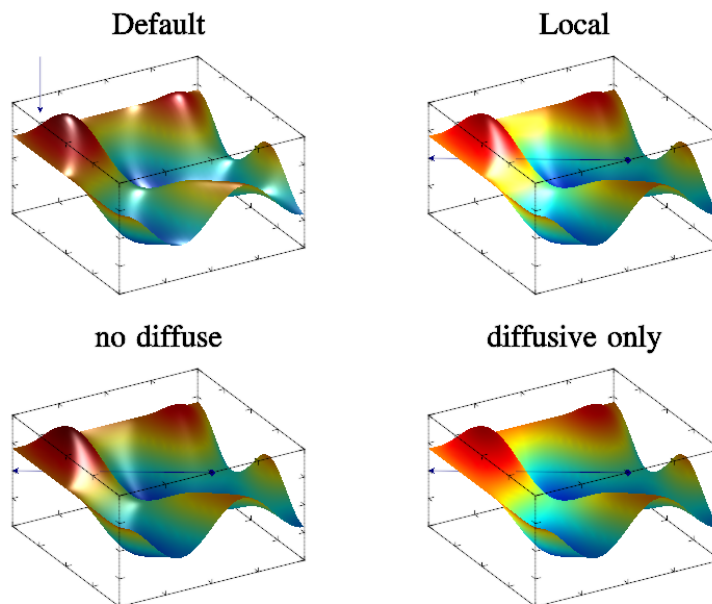
Additionally, you can use local light sources and set to use [diffuse], page 159, reflection instead of specular one (by default) or both kinds. Note, I use [attachlight], page 159, command to keep light settings relative to subplot.

```
int sample(mglGraph *gr)
{
    gr->Light(true); gr->AttachLight(true);
    gr->SubPlot(2,2,0); gr->Title("Default"); gr->Rotate(50,60);
    gr->Line(mglPoint(-1,-0.7,1.7),mglPoint(-1,-0.7,0.7),"BA"); gr->Box(); gr->Surf(a);█

    gr->SubPlot(2,2,1); gr->Title("Local"); gr->Rotate(50,60);
    gr->AddLight(0,mglPoint(1,0,1),mglPoint(-2,-1,-1));
    gr->Line(mglPoint(1,0,1),mglPoint(-1,-1,0),"BA0"); gr->Box(); gr->Surf(a);█

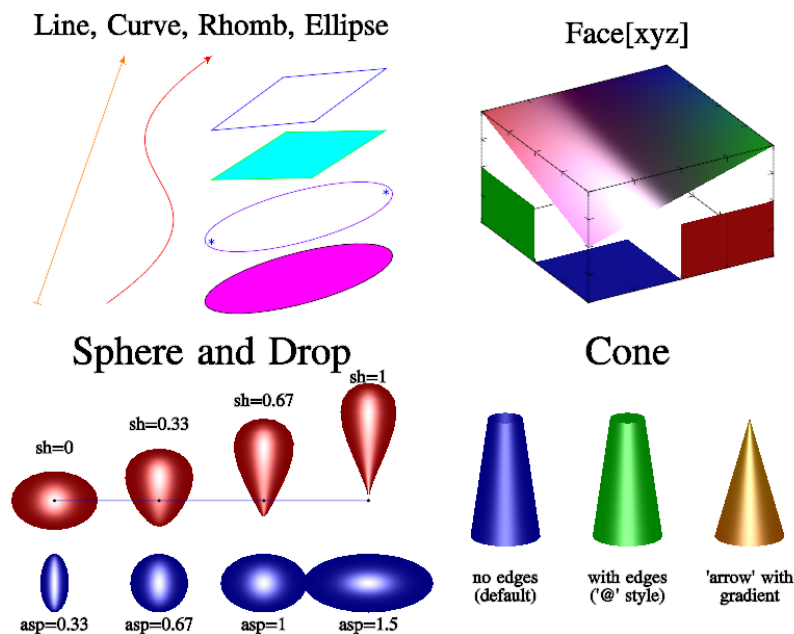
    gr->SubPlot(2,2,2); gr->Title("no diffuse"); gr->Rotate(50,60);
    gr->SetDiffuse(0);
    gr->Line(mglPoint(1,0,1),mglPoint(-1,-1,0),"BA0"); gr->Box(); gr->Surf(a);█

    gr->SubPlot(2,2,3); gr->Title("diffusive only"); gr->Rotate(50,60);
    gr->SetDiffuse(0.5);
    gr->AddLight(0,mglPoint(1,0,1),mglPoint(-2,-1,-1),'w',0);
    gr->Line(mglPoint(1,0,1),mglPoint(-1,-1,0),"BA0"); gr->Box(); gr->Surf(a);█
}
```



### 2.9.7 Using primitives

MathGL provide a set of functions for drawing primitives (see Section 4.7 [Primitives], page 184). Primitives are low level object, which used by most of plotting functions. Picture below demonstrate some of commonly used primitives.



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 186. The sample code is:

```
int sample(mglGraph *gr)
{
```

```

gr->Alpha(true);  gr->Light(true);

gr->SubPlot(2,2,0,"");  gr->Title("Methane, CH4");
gr->StartGroup("Methane");
gr->Rotate(60,120);
gr->Sphere(mglPoint(0,0,0),0.25,"k");
gr->Drop(mglPoint(0,0,0),mglPoint(0,0,1),0.35,"h",1,2);
gr->Sphere(mglPoint(0,0,0.7),0.25,"g");
gr->Drop(mglPoint(0,0,0),mglPoint(-0.94,0,-0.33),0.35,"h",1,2);
gr->Sphere(mglPoint(-0.66,0,-0.23),0.25,"g");
gr->Drop(mglPoint(0,0,0),mglPoint(0.47,0.82,-0.33),0.35,"h",1,2);
gr->Sphere(mglPoint(0.33,0.57,-0.23),0.25,"g");
gr->Drop(mglPoint(0,0,0),mglPoint(0.47,-0.82,-0.33),0.35,"h",1,2);
gr->Sphere(mglPoint(0.33,-0.57,-0.23),0.25,"g");
gr->EndGroup();

gr->SubPlot(2,2,1,"");  gr->Title("Water, H2O");
gr->StartGroup("Water");
gr->Rotate(60,100);
gr->StartGroup("Water_O");
gr->Sphere(mglPoint(0,0,0),0.25,"r");
gr->EndGroup();
gr->StartGroup("Water_Bond_1");
gr->Drop(mglPoint(0,0,0),mglPoint(0.3,0.5,0),0.3,"m",1,2);
gr->EndGroup();
gr->StartGroup("Water_H_1");
gr->Sphere(mglPoint(0.3,0.5,0),0.25,"g");
gr->EndGroup();
gr->StartGroup("Water_Bond_2");
gr->Drop(mglPoint(0,0,0),mglPoint(0.3,-0.5,0),0.3,"m",1,2);
gr->EndGroup();
gr->StartGroup("Water_H_2");
gr->Sphere(mglPoint(0.3,-0.5,0),0.25,"g");
gr->EndGroup();
gr->EndGroup();

gr->SubPlot(2,2,2,"");  gr->Title("Oxygen, O2");
gr->StartGroup("Oxygen");
gr->Rotate(60,120);
gr->Drop(mglPoint(0,0.5,0),mglPoint(0,-0.3,0),0.3,"m",1,2);
gr->Sphere(mglPoint(0,0.5,0),0.25,"r");
gr->Drop(mglPoint(0,-0.5,0),mglPoint(0,0.3,0),0.3,"m",1,2);
gr->Sphere(mglPoint(0,-0.5,0),0.25,"r");
gr->EndGroup();

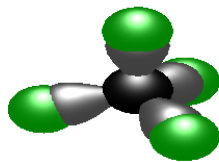
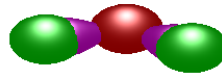
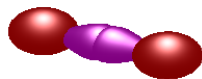
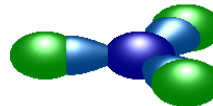
gr->SubPlot(2,2,3,"");  gr->Title("Ammonia, NH3");
gr->StartGroup("Ammonia");

```

```

gr->Rotate(60,120);
gr->Sphere(mglPoint(0,0,0),0.25,"b");
gr->Drop(mglPoint(0,0,0),mglPoint(0.33,0.57,0),0.32,"n",1,2);
gr->Sphere(mglPoint(0.33,0.57,0),0.25,"g");
gr->Drop(mglPoint(0,0,0),mglPoint(0.33,-0.57,0),0.32,"n",1,2);
gr->Sphere(mglPoint(0.33,-0.57,0),0.25,"g");
gr->Drop(mglPoint(0,0,0),mglPoint(-0.65,0,0),0.32,"n",1,2);
gr->Sphere(mglPoint(-0.65,0,0),0.25,"g");
gr->EndGroup();
return 0;
}

```

Methane, CH<sub>4</sub>Water, H<sub>2</sub>OOxygen, O<sub>2</sub>Ammonia, NH<sub>3</sub>

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:

```

int sample(mglGraph *gr)
{
    double xx[3]={-0.3,0,0.3}, yy[3]={0.3,-0.3,0.3}, ee[3]={0.7,0.7,0.7};
    mglData x(3,xx), y(3,yy), e(3,ee);
    gr->Title("Venn-like diagram"); gr->Alpha(true);
    gr->Error(x,y,e,e,"!rgb@#o");
    return 0;
}

```

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 186, function:

```

int sample(mglGraph *gr)
{
    gr->Title("Venn-like diagram"); gr->Alpha(true);
    gr->Circle(mglPoint(-0.3,0.3),0.7,"rr@");
}

```

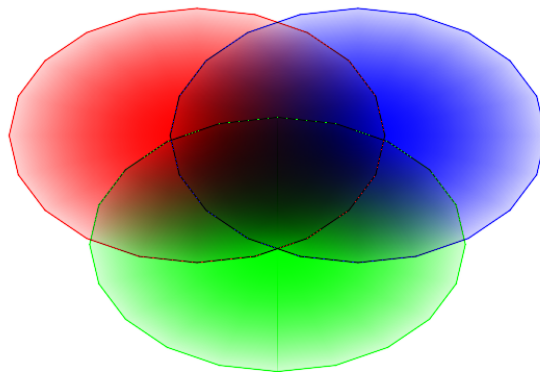
```

gr->Circle(mglPoint(0,-0.3),0.7,"gg@");
gr->Circle(mglPoint( 0.3,0.3),0.7,"bb@");
return 0;
}

```

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.

## Venn-like diagram



### 2.9.8 STFA sample

Short-time Fourier Analysis ([stfa], page 221) 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:

```

int sample(mglGraph *gr)
{
    mglData a(2000), b(2000);
    gr->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)");
    gr->SubPlot(1, 2, 0,"<_"); gr->Title("Initial signal");
    gr->Plot(a);
    gr->Axis();
    gr->Label('x', "\\i t");

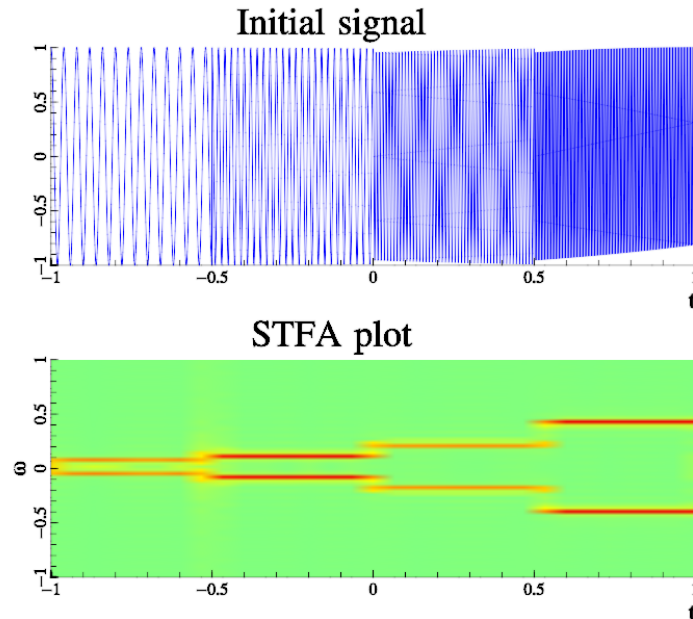
    gr->SubPlot(1, 2, 1,"<_"); gr->Title("STFA plot");
    gr->STFA(a, b, 64);
    gr->Axis();
}

```

```

gr->Label('x', "\\i t");
gr->Label('y', "\\omega", 0);
return 0;
}

```



### 2.9.9 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 [surf], page 217, [surfa], page 218) 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 221. 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:

```

int sample(mglGraph *gr)
{
    mglData a(50, 40), b(50, 40);
    gr->Puts(mglPoint(0, 0), "\\to", ":C", -1.4);
    gr->SetRanges(-1,1,-1,1,-2,2);

    gr->SubPlot(2, 1, 0);
    gr->Fill(a,"x"); gr->Fill(b,"y");
}

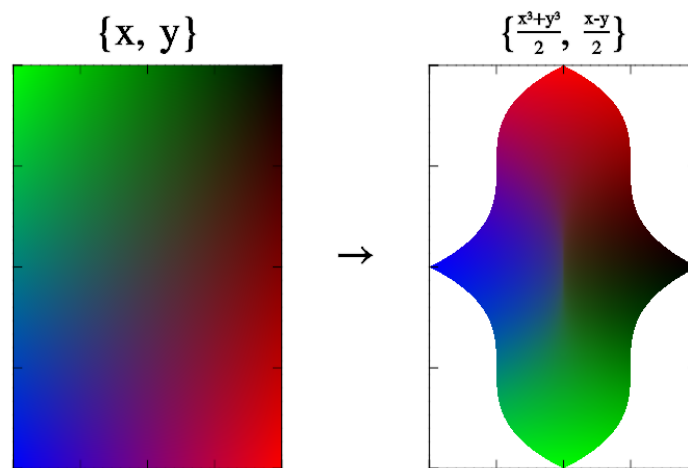
```

```

gr->Puts(mglPoint(0, 1.1), "\\{x, y\\}", ":C", -2);    gr->Box();
gr->Map(a, b, "brgk");

gr->SubPlot(2, 1, 1);
gr->Fill(a, "(x^3+y^3)/2");  gr->Fill(b, "(x-y)/2");
gr->Puts(mglPoint(0, 1.1), "\\{\\frac{x^3+y^3}{2}, \\frac{x-y}{2}\\}", ":C", -2);
gr->Box();
gr->Map(a, b, "brgk");
return 0;
}

```



### 2.9.10 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 6.8 [Interpolation], page 271, and `mglGSpline()` in Section 6.11 [Global functions], page 277);
2. functions `[subdata]`, page 264, and `[evaluate]`, page 265, for indirect access to data elements;
3. functions `[refill]`, page 260, `[gspline]`, page 261, and `[datagrid]`, page 259, which fill regular (rectangular) data array by interpolated values.

The usage of first category is rather straightforward and don't need any special comments.

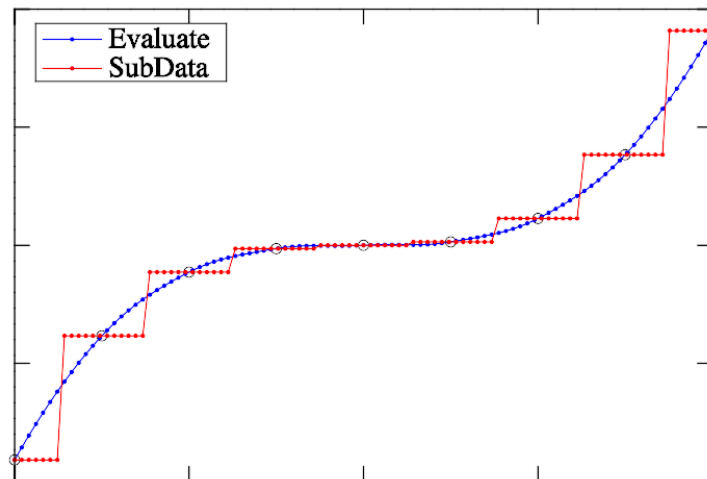
There is difference in indirect access functions. Function `[subdata]`, page 264, use step-like interpolation to handle correctly single `nan` values in the data array. Contrary, function `[evaluate]`, page 265, use local spline interpolation, which give smoother output but spread `nan` values. So, `[subdata]`, page 264, should be used for specific data elements



(for example, for given column), and [evaluate], page 265, should be used for distributed elements (i.e. consider data array as some field). Following sample illustrates this difference:

```
int sample(mglGraph *gr)
{
    gr->SubPlot(1,1,0,""); gr->Title("SubData vs Evaluate");
    mglData in(9), arg(99), e, s;
    gr->Fill(in,"x^3/1.1"); gr->Fill(arg,"4*x+4");
    gr->Plot(in,"ko "); gr->Box();
    e = in.Evaluate(arg); gr->Plot(e,"b.", "legend 'Evaluate'");
    s = in.SubData(arg); gr->Plot(s,"r.", "legend 'SubData'");
    gr->Legend(2);
}
```

## SubData vs Evaluate



Example of [datagrid], page 259, usage is done in Section 2.9.11 [Making regular data], page 127. Here I want to show the peculiarities of [refill], page 260, and [gspline], page 261, 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 265, function, [refill], page 260, and [gspline], page 261, can interpolate non-equidistantly distributed data. At this both functions [refill], page 260, and [gspline], page 261, provide continuity of 2nd derivatives along coordinate(s). However, [refill], page 260, is slower but give better (from human point of view) result than global spline [gspline], page 261, due to more advanced algorithm. Following sample illustrates this difference:

```
int sample(mglGraph *gr)
{
    mglData x(10), y(10), r(100);
    x.Modify("0.5+rnd"); x.CumSum("x"); x.Norm(-1,1);
    y.Modify("sin(pi*v)/1.5",x);
}
```

```

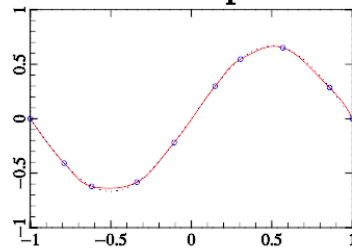
gr->SubPlot(2,2,0,"<_"); gr->Title("Refill sample");
gr->Axis(); gr->Box(); gr->Plot(x,y,"o ");
gr->Refill(r,x,y); // or you can use r.Refill(x,y,-1,1);
gr->Plot(r,"r"); gr->FPlot("sin(pi*x)/1.5","B:");
gr->SubPlot(2,2,1,"<_");gr->Title("Global spline");
gr->Axis(); gr->Box(); gr->Plot(x,y,"o ");
r.RefillGS(x,y,-1,1); gr->Plot(r,"r");
gr->FPlot("sin(pi*x)/1.5","B:");

gr->Alpha(true); gr->Light(true);
mglData z(10,10), xx(10,10), yy(10,10), rr(100,100);
y.Modify("0.5+rand"); y.CumSum("x"); y.Norm(-1,1);
for(int i=0;i<10;i++) for(int j=0;j<10;j++)
    z.a[i+10*j] = sin(M_PI*x.a[i]*y.a[j])/1.5;
gr->SubPlot(2,2,2); gr->Title("2d regular"); gr->Rotate(40,60);
gr->Axis(); gr->Box(); gr->Mesh(x,y,z,"k");
gr->Refill(rr,x,y,z); gr->Surf(rr);

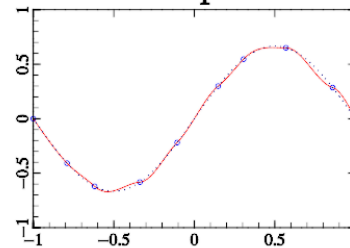
gr->Fill(xx,"(x+1)/2*cos(y*pi/2-1)");
gr->Fill(yy,"(x+1)/2*sin(y*pi/2-1)");
for(int i=0;i<10*10;i++)
    z.a[i] = sin(M_PI*xx.a[i]*yy.a[i])/1.5;
gr->SubPlot(2,2,3); gr->Title("2d non-regular"); gr->Rotate(40,60);
gr->Axis(); gr->Box(); gr->Plot(xx,yy,z,"ko ");
gr->Refill(rr,xx,yy,z); gr->Surf(rr);
}

```

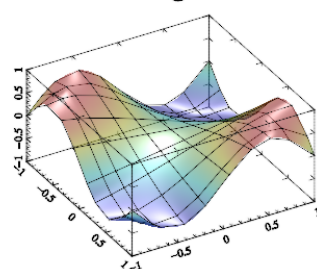
Refill sample



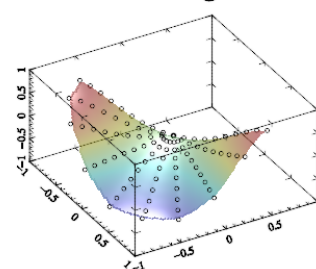
Global spline



2d regular



2d non-regular



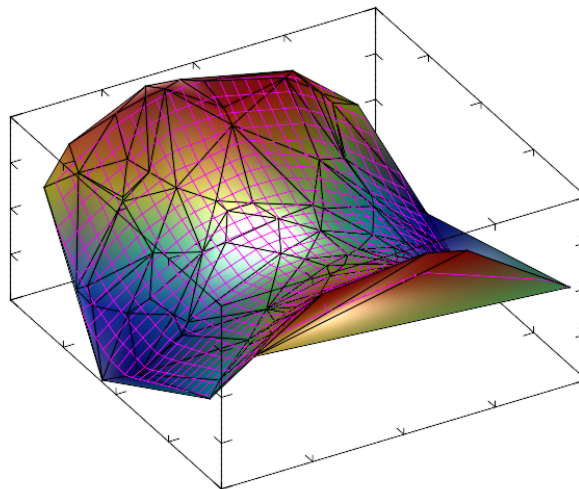
### 2.9.11 Making regular data

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 231, 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 280, function to obtain list of vertexes for triangles. Later this list can be used in functions like [triplot], page 230, or [tricont], page 230. Second way consist in usage of [datagrid], page 259, 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 3.7 [Command options], page 153) to change default axis range as well as in other plotting functions.

```
int sample(mglGraph *gr)
{
    mglData x(100), y(100), z(100);
    gr->Fill(x,"2*rnd-1"); gr->Fill(y,"2*rnd-1"); gr->Fill(z,"v^2-w^2",x,y);
    // first way - plot triangular surface for points
    mglData d = mglTriangulation(x,y);
    gr->Title("Triangulation");
    gr->Rotate(40,60); gr->Box(); gr->Light(true);
    gr->TriPlot(d,x,y,z); gr->TriPlot(d,x,y,z,"#k");
    // second way - make regular data and plot it
    mglData g(30,30);
    gr->DataGrid(g,x,y,z); gr->Mesh(g,"m");
}
```

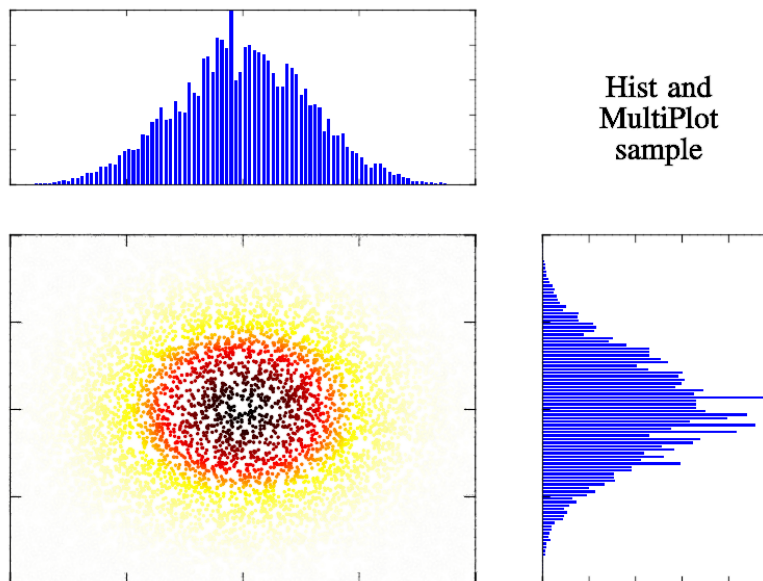
## Triangulation



### 2.9.12 Making histogram

Using the [hist], page 266, 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 266, usage:

```
int sample(mglGraph *gr)
{
    mglData x(10000), y(10000), z(10000); gr->Fill(x,"2*rnd-1");
    gr->Fill(y,"2*rnd-1"); gr->Fill(z,"exp(-6*(v^2+w^2))",x,y);
    mglData xx=gr->Hist(x,z), yy=gr->Hist(y,z); xx.Norm(0,1);
    yy.Norm(0,1);
    gr->MultiPlot(3,3,3,2,2,""); gr->SetRanges(-1,1,-1,1,0,1);
    gr->Box(); gr->Dots(x,y,z,"wyrRk");
    gr->MultiPlot(3,3,0,2,1,""); gr->SetRanges(-1,1,0,1);
    gr->Box(); gr->Bars(xx);
    gr->MultiPlot(3,3,5,1,2,""); gr->SetRanges(0,1,-1,1);
    gr->Box(); gr->Barh(yy);
    gr->SubPlot(3,3,2);
    gr->Puts(mglPoint(0.5,0.5),"Hist and\nMultiPlot\nsample","a",-6);
    return 0;
}
```



Hist and  
MultiPlot  
sample

### 2.9.13 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:

```
mglData dat(100), in(100); //data to be fitted and ideal data
gr->Fill(dat,"0.4*rnd+0.1+sin(2*pi*x)");
gr->Fill(in,"0.3+sin(2*pi*x)");

and plot it to see that data we will fit

gr->Title("Fitting sample");
gr->SetRange('y',-2,2); gr->Box(); gr->Plot(dat, "k. ");
gr->Axis(); gr->Plot(in, "b");
gr->Puts(mglPoint(0, 2.2), "initial: y = 0.3+sin(2\\pi x)", "b");
```

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)'

```
mreal ini[3] = {1,1,3};
mglData Ini(3,ini);
mglData res = gr->Fit(dat, "a+b*sin(c*x)", "abc", Ini);
```

Now display it

```
gr->Plot(res, "r");
gr->Puts(mglPoint(-0.9, -1.3), "fitted:", "r:L");
gr->PutsFit(mglPoint(0, -1.8), "y = ", "r");
```

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.

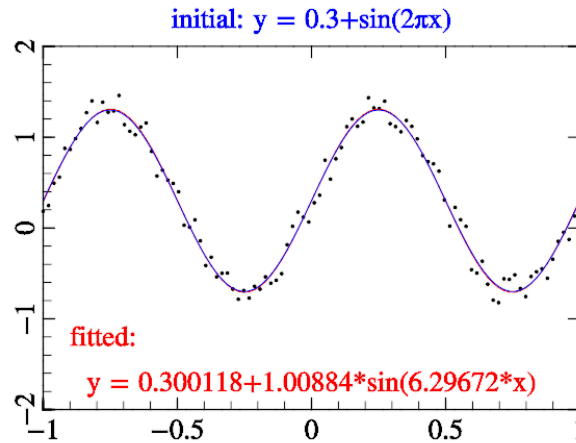
The full sample code for nonlinear fitting is:

```
int sample(mglGraph *gr)
{
    mglData dat(100), in(100);
    gr->Fill(dat,"0.4*rnd+0.1+sin(2*pi*x)");
    gr->Fill(in,"0.3+sin(2*pi*x)");
    mreal ini[3] = {1,1,3};
    mglData Ini(3,ini);

    mglData res = gr->Fit(dat, "a+b*sin(c*x)", "abc", Ini);

    gr->Title("Fitting sample");
    gr->SetRange('y',-2,2); gr->Box(); gr->Plot(dat, "k. ");
    gr->Axis(); gr->Plot(res, "r"); gr->Plot(in, "b");
    gr->Puts(mglPoint(-0.9, -1.3), "fitted:", "r:L");
    gr->PutsFit(mglPoint(0, -1.8), "y = ", "r");
    gr->Puts(mglPoint(0, 2.2), "initial: y = 0.3+sin(2\\pi x)", "b");
    return 0;
}
```

## Fitting sample



### 2.9.14 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 278, for ray tracing, [pde], page 278, for PDE solving, [qo2d], page 279, for beam tracing in 2D case (see Section 6.11 [Global functions], page 277). 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 278, 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<sup>2</sup>+q<sup>2</sup>-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

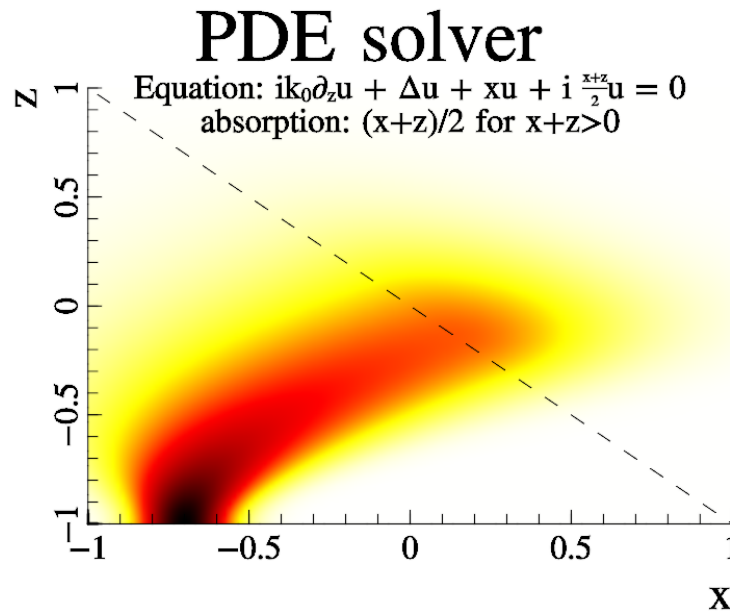
look like  $du/dz = ik_0 H(x, y, \hat{p}, \hat{q}, |u|)[u]$ . Dependence on field amplitude ' $u=|u|$ ' allows one to solve nonlinear problems too. For example, for nonlinear Shrodinger equation you may set `ham="p^2 + q^2 - u^2"`. Also you may specify imaginary part for wave absorption, like `ham = "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  $k_0=k_0$ . 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 ' $x*p \rightarrow x*d/dx$ ') are excluded. For example, in 2D case this operator is effectively  $H = f(p, z) + g(x, z, u)$ . However commutable combinations (like ' $x*q \rightarrow x*d/dy$ ') 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:

```
int sample(mglGraph *gr)
{
    mglData a,re(128),im(128);
    gr->Fill(re,"exp(-48*(x+0.7)^2)");
    a = gr->PDE("p^2+q^2-x-1+i*0.5*(z+x)*(z>-x)", re, im, 0.01, 30);
    a.Transpose("yxz");
    gr->SubPlot(1,1,0,"<_"); gr->Title("PDE solver");
    gr->SetRange('c',0,1); gr->Dens(a,"wyrRk");
    gr->Axis(); gr->Label('x', "\\i x"); gr->Label('y', "\\i z");
    gr->FPlot("-x", "k|");
    gr->Puts(mglPoint(0, 0.85), "absorption: (x+z)/2 for x+z>0");
    gr->Puts(mglPoint(0,1.1),"Equation: ik_0\\partial_zu + \\Delta u + x\\cdot u + i \\frac{x}{2}u");
    return 0;
}
```



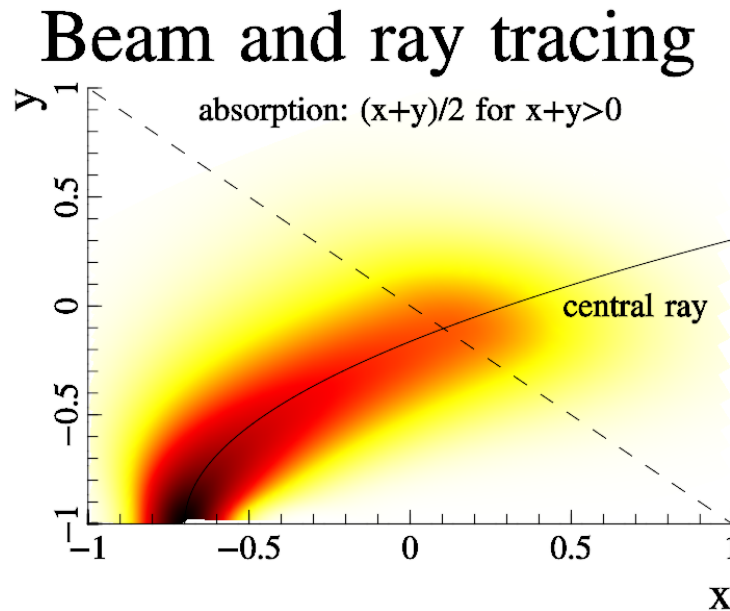
The last example is example of beam tracing. Beam tracing equation is special kind of PDE equation written in coordinates accompanied to a ray. Generally this is the same parameters and limitation as for PDE solving but the coordinates are defined by the ray and by parameter of grid width  $w$  in direction 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:  $K \gg w$ . 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:

```
int sample(mglGraph *gr)
{
    mglData r, xx, yy, a, im(128), re(128);
    const char *ham = "p^2+q^2-x-1+i*0.5*(y+x)*(y>-x)";
    r = mglRay(ham, mglPoint(-0.7, -1), mglPoint(0, 0.5), 0.02, 2);
    gr->SubPlot(1,1,0,"<_"); gr->Title("Beam and ray tracing");
    gr->Plot(r.SubData(0), r.SubData(1), "k");
    gr->Axis(); gr->Label('x', "\\i x"); gr->Label('y', "\\i z");

    // now start beam tracing
    gr->Fill(re,"exp(-48*x^2)");
    a = mglQO2d(ham, re, im, r, xx, yy, 1, 30);
    gr->SetRange('c',0, 1);
    gr->Dens(xx, yy, a, "wyrRk");
    gr->FPlot("-x", "k|");
    gr->Puts(mglPoint(0, 0.85), "absorption: (x+y)/2 for x+y>0");
    gr->Puts(mglPoint(0.7, -0.05), "central ray");
    return 0;
}
```





## 2.9.15 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 224, function(s), which automatically select several points and draw flow threads. If the ODE have an integral of motion (like Hamiltonian  $H(x, x') = \text{const}$  for dissipation-free case) then you can use [cont], page 209, 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 278, 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  $H = y^2 + 2 * x^3 - x^2 = \text{Const}$ . 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.

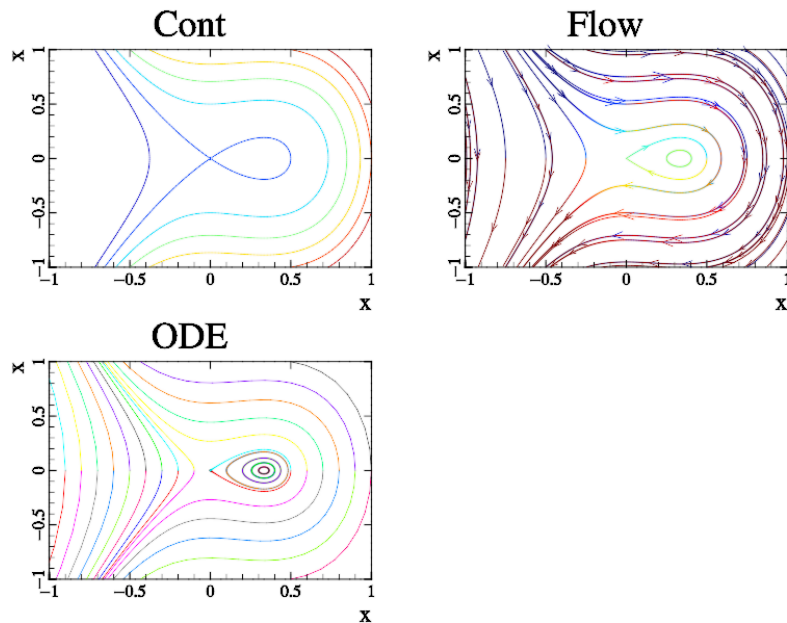
So, let draw this phase plain by 3 different methods. First, draw isolines for  $H = y^2 + 2 * x^3 - x^2 = \text{Const}$  – this is simplest for ODE without dissipation. Next, draw flow threads – this is straightforward way, but the automatic choice of starting points is not always optimal. Finally, use [ode], page 278, to check the above plots. At this we need to run [ode], page 278, 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 224) the plot, but this is a more complicated. The sample code is:

```
int sample(mglGraph *gr)
{
```

```

gr->SubPlot(2,2,0,"<_"); gr->Title("Cont"); gr->Box();
gr->Axis(); gr->Label('x',"x"); gr->Label('y',"\\dot{x}");
mglData f(100,100); gr->Fill(f,"y^2+2*x^3-x^2-0.5");
gr->Cont(f);
gr->SubPlot(2,2,1,"<_"); gr->Title("Flow"); gr->Box();
gr->Axis(); gr->Label('x',"x"); gr->Label('y',"\\dot{x}");
mglData fx(100,100), fy(100,100);
gr->Fill(fx,"x-3*x^2"); gr->Fill(fy,"y");
gr->Flow(fy,fx,"v","value 7");
gr->SubPlot(2,2,2,"<_"); gr->Title("ODE"); gr->Box();
gr->Axis(); gr->Label('x',"x"); gr->Label('y',"\\dot{x}");
for(double x=-1;x<1;x+=0.1)
{
    mglData in(2), r; in.a[0]=x;
    r = mglODE("y;x-3*x^2","xy",in);
    gr->Plot(r.SubData(0), r.SubData(1));
    r = mglODE("-y;-x+3*x^2","xy",in);
    gr->Plot(r.SubData(0), r.SubData(1));
}
}

```



### 2.9.16 Pulse properties

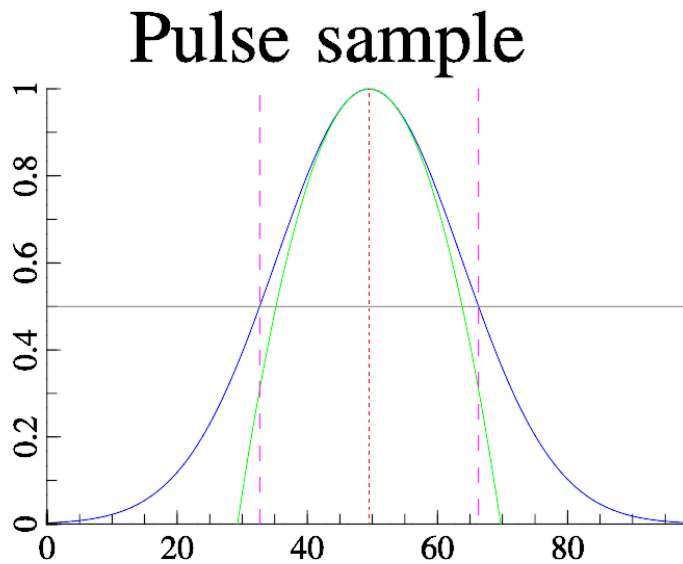
There is common task in optics to determine properties of wave pulses or wave beams. MathGL provide special function [pulse], page 268, which return the pulse properties (maximal value, center of mass, width and so on). Its usage is rather simple. Here I just illustrate it on the example of Gaussian pulse, where all parameters are obvious.

```
void sample(mglGraph *gr)
```

```

{
    gr->SubPlot(1,1,0,"<_"); gr->Title("Pulse sample");
    // first prepare pulse itself
    mglData a(100); gr->Fill(a,"exp(-6*x^2)");
    // get pulse parameters
    mglData b(a.Pulse('x'));
    // positions and widths are normalized on the number of points. So, set proper axis scale
    gr->SetRanges(0, a.nx-1, 0, 1);
    gr->Axis(); gr->Plot(a); // draw pulse and axis
    // now visualize found pulse properties
    double m = b[0]; // maximal amplitude
    // approximate position of maximum
    gr->Line(mglPoint(b[1],0), mglPoint(b[1],m),"r=");
    // width at half-maximum (so called FWHM)
    gr->Line(mglPoint(b[1]-b[3]/2,0), mglPoint(b[1]-b[3]/2,m),"m|");
    gr->Line(mglPoint(b[1]+b[3]/2,0), mglPoint(b[1]+b[3]/2,m),"m|");
    gr->Line(mglPoint(0,m/2), mglPoint(a.nx-1,m/2),"h");
    // parabolic approximation near maximum
    char func[128]; sprintf(func,"%g*(1-((x-%g)/%g)^2)",b[0],b[1],b[2]);
    gr->FPlot(func,"g");
}

```



### 2.9.17 Using MGL parser

Sometimes you may prefer to use MGL scripts in yours code. It is simpler (especially in comparison with C/Fortran interfaces) and provide faster way to plot the data with annotations, labels and so on. Class `mglParse` (see Section 7.4 [mglParse class], page 291, parse MGL scripts in C++. It have also the corresponding interface for C/Fortran.

The key function here is `mglParse::Parse()` (or `mgl_parse()` for C/Fortran) which execute one command per string. At this the detailed information about the possible errors or warnings is passed as function value. Or you may execute the whole script as long string with lines separated by `'\n'`. Functions `mglParse::Execute()` and `mgl_parse_text()` perform it. Also you may set the values of parameters `'$0'...' $9'` for the script by functions `mglParse::AddParam()` or `mgl_add_param()`, allow/disable picture resizing, check “once” status and so on. The usage is rather straight-forward.

The only non-obvious thing is data transition between script and yours program. There are 2 stages: add or find variable; and set data to variable. In C++ you may use functions `mglParse::AddVar()` and `mglParse::FindVar()` which return pointer to `mglData`. In C/Fortran the corresponding functions are `mgl_add_var()`, `mgl_find_var()`. This data pointer is valid until next `Parse()` or `Execute()` call. Note, you **must not delete or free** the data obtained from these functions!

So, some simple example at the end. Here I define a data array, create variable, put data into it and plot it. The C++ code looks like this:

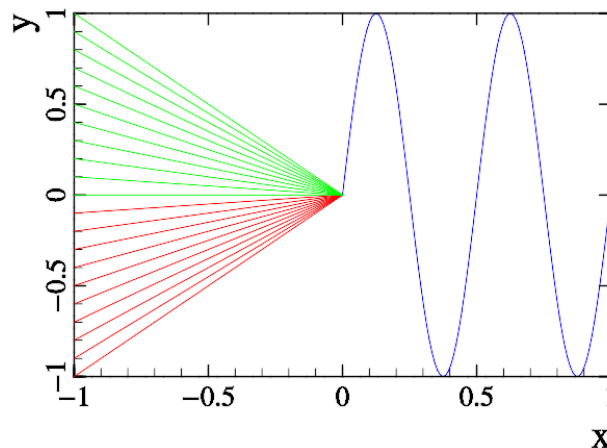
```
int sample(mglGraph *gr)
{
    gr->Title("MGL parser sample");
    mreal a[100];    // let a_i = sin(4*pi*x), x=0...1
    for(int i=0;i<100;i++)a[i]=sin(4*M_PI*i/99);
    mglParse *parser = new mglParse;
    mglData *d = parser->AddVar("dat");
    d->Set(a,100); // set data to variable
    parser->Execute(gr, "plot dat; xrange 0 1\nbox\naxis");
    // you may break script at any line do something
    // and continue after that
    parser->Execute(gr, "xlabel 'x'\nylabel 'y'\nbox");
    // also you may use cycles or conditions in script
    parser->Execute(gr, "for $0 -1 1 0.1\nif $0<0\n"
        "line 0 0 -1 $0 'r':else:line 0 0 -1 $0 'g'\n"
        "endif\nnext");
    delete parser;
    return 0;
}
```

The code in C/Fortran looks practically the same:

```
int sample(HMGL gr)
{
    mgl_title(gr, "MGL parser sample", "", -2);
    double a[100];    // let a_i = sin(4*pi*x), x=0...1
    int i;
    for(i=0;i<100;i++) a[i]=sin(4*M_PI*i/99);
    HMPR parser = mgl_create_parser();
    HMDT d = mgl_parser_add_var(parser, "dat");
    mgl_data_set_double(d,a,100,1,1);    // set data to variable
    mgl_parse_text(gr, parser, "plot dat; xrange 0 1\nbox\naxis");
    // you may break script at any line do something
```

```
// and continue after that
mgl_parse_text(gr, parser, "xlabel 'x'\nylabel 'y'");
// also you may use cycles or conditions in script
mgl_parse_text(gr, parser, "for $0 -1 1 0.1\nif $0<0\n"
    "line 0 0 -1 $0 'r':else:line 0 0 -1 $0 'g'\n"
    "endif\nnext");
mgl_write_png(gr, "test.png", ""); // don't forgot to save picture
return 0;
}
```

## MGL parser sample



### 2.9.18 Using options

Section 3.7 [Command options], page 153, 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:

```
void template(mglGraph *gr)
{
    mglData a(31,41);
    gr->Fill(a,"-pi*x*exp(-(y+1)^2-4*x^2)");

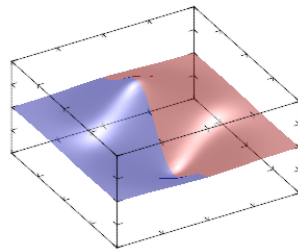
    gr->SubPlot(2,2,0);    gr->Title("Options for coordinates");
    gr->Alpha(true);        gr->Light(true);
    gr->Rotate(40,60);      gr->Box();
    gr->Surf(a,"r","yrange 0 1"); gr->Surf(a,"b","yrange 0 -1");
    if(mini) return;
    gr->SubPlot(2,2,1);    gr->Title("Option 'meshnum'");
}
```

```

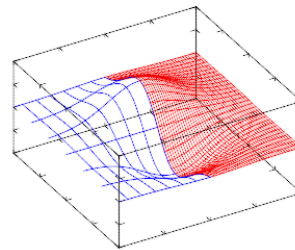
gr->Rotate(40,60);    gr->Box();
gr->Mesh(a,"r","yrange 0 1"); gr->Mesh(a,"b","yrange 0 -1; meshnum 5");
gr->SubPlot(2,2,2);    gr->Title("Option 'alpha'");
gr->Rotate(40,60);    gr->Box();
gr->Surf(a,"r","yrange 0 1; alpha 0.7");
gr->Surf(a,"b","yrange 0 -1; alpha 0.3");
gr->SubPlot(2,2,3,"<_"); gr->Title("Option 'legend'");
gr->FPlot("x^3","r","legend 'y = x^3'");
gr->FPlot("cos(pi*x)","b","legend 'y = cos \\pi x'");
gr->Box();    gr->Axis(); gr->Legend(2,"");
}

```

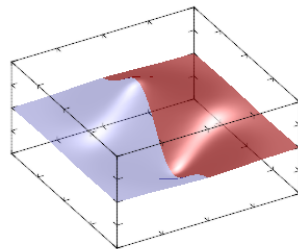
Options for coordinates



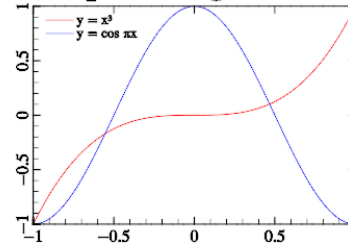
Option 'meshnum'



Option 'alpha'



Option 'legend'



### 2.9.19 “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.

### 2.9.20 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:

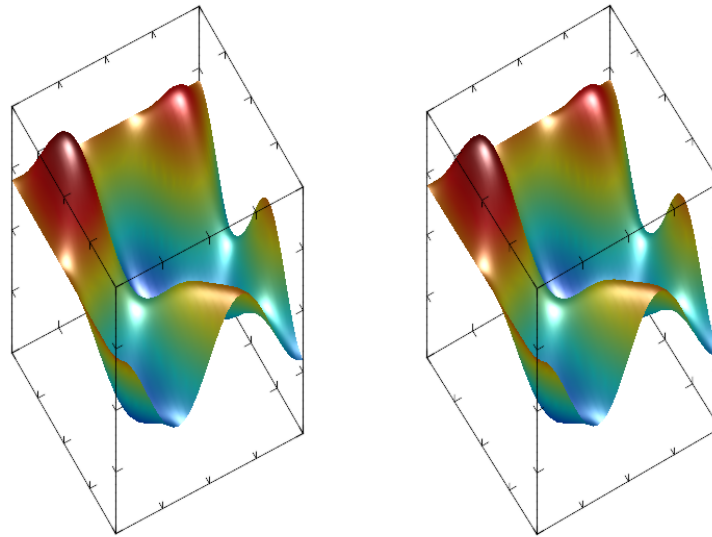
```

int sample(mglGraph *gr)
{
    mglData a; mglS_prepare2d(&a);
    gr->Light(true);

    gr->SubPlot(2,1,0); gr->Rotate(50,60+1);
    gr->Box(); gr->Surf(a);

    gr->SubPlot(2,1,1); gr->Rotate(50,60-1);
    gr->Box(); gr->Surf(a);
    return 0;
}

```



### 2.9.21 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 214, [dew], page 224). You can use [quality], page 175, 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 lowest memory usage looks like this:

```
int sample(mglGraph *gr)
{
    gr->SetQuality(6); // firstly, set to draw directly on bitmap
    for(i=0;i<1000;i++)
        gr->Sphere(mglPoint(mgl_rnd()*2-1,mgl_rnd()*2-1),0.05);
    return 0;
}
```

### 2.9.22 Scanning file

MathGL have possibilities to write textual information into file with variable values. In MGL script you can use [save], page 262, command for that. However, the usual `printf()`; is simple in C/C++ code. For example, lets create some textual file

```
FILE *fp=fopen("test.txt","w");
fprintf(fp,"This is test: 0 -> 1 q\n");
fprintf(fp,"This is test: 1 -> -1 q\n");
fprintf(fp,"This is test: 2 -> 0 q\n");
fclose(fp);
```

It contents look like

```
This is test: 0 -> 1 q
```



```
This is test: 1 -> -1 q
This is test: 2 -> 0 q
```

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 262, for that. The desired values was written using template "This is test: %g -> %g q\n". So, just use

```
mglData a;
a.ScanFile("test.txt","This is test: %g -> %g");
```

and plot it to for assurance

```
gr->SetRanges(a.SubData(0), a.SubData(1));
gr->Axis();      gr->Plot(a.SubData(0),a.SubData(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\n"), because there is no important for us information after the second number in the line.

## 2.10 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 166, function. Check that the data have correct dimensions for selected type of plot. Be sure that `Finish()` is called after the plotting functions (or be sure that the plot is saved to a file). Sometimes the light reflection from flat surfaces (like, `[dens]`, page 209) 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 205, or as a parametrically specified surface by `[surf]`, page 207. See also, Section 2.9 `[Hints]`, page 110. 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.

### Should I know some graphical libraries (like OpenGL) before using the MathGL library?

No. The MathGL library is self-contained and does not require the knowledge of external libraries.

### In which language is the library written? For which languages does it have an interface?

The core of the MathGL library is written in C++. But there are interfaces for: pure C, Fortran, Pascal, Forth, and its own command language MGL. Also there is a large set of interpreted languages, which are supported (Python, Java, ALLEGROCL, CHICKEN, Lisp, CFFI, C#, Guile, Lua, Modula 3, Mzscheme, Ocaml, Octave, Perl, PHP, Pike, R, Ruby, Tcl). These interfaces are written using SWIG (both pure C functions and classes) but only the interface for Python and Octave is included in the build system. The reason is that I don't know any other interpreted languages :(. Note that most other languages can use (link to) the pure C functions.

**How can I use MathGL with Fortran?**

You can use MathGL as is with `gfortran` because it uses by default the AT&T notation for external functions. For other compilers (like Visual Fortran) you have to switch on the AT&T notation manually. The AT&T notation requires that the symbol ‘\_’ is added at the end of each function name, function argument(s) is passed by pointers and the string length(s) is passed at the end of the argument list. For example:

*C function* – `void mgl_fplot(HMGL graph, const char *fy, const char *stl, int n);`

*AT&T function* – `void mgl_fplot_(uintptr_t *graph, const char *fy, const char *stl, int *n, int ly, int ls);`

Fortran users also should add C++ library by the option `-lstdc++`. If library was built with `enable-double=ON` (this default for v.2.1 and later) then all real numbers must be `real*8`. You can make it automatic if use option `-fdefault-real-8`.

**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. You may change it by `setlocale()` function. For example, for Russian text in CP1251 encoding you may use `setlocale(LC_CTYPE, "ru_RU.cp1251");` (under MS Windows the name of locale may differ – `setlocale(LC_CTYPE, "russian_russia.1251")`). I strongly recommend not to use the constant `LC_ALL` in the conversion. Since it also changes the number format, it may lead to mistakes in formula writing and reading of the text in data files. For example, the program will await a ‘,’ as a decimal point but the user will enter ‘.’.

**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 `SetCutBox()` and `CutOff()` define the condition when the points should be omitted (see Section 4.2.5 [Cutting], page 161). Last, you may change the transparency of a part of the plot by the help of functions `[surfa]`, page 218, `[surf3a]`, page 218, (see Section 4.14 [Dual plotting], page 217). In last case the transparency is switched on smoothly.

**I use VisualStudio, CBuilder or some other compiler (not MinGW/gcc). How can I link the MathGL library?**

In version 2.0, main classes (`mglGraph` and `mglData`) contains only `inline` functions and are acceptable for any compiler with the same binary files. However, if you plan to use widget classes (`QMathGL`, `Fl_MathGL`, ...) or to access low-level features (`mglBase`, `mglCanvas`, ...) then you have to recompile MathGL by yours compiler.

Note, that you have to make import library(-ies) \*.lib for provided binary \*.dll. This procedure depend on used compiler – please read documentation for yours compiler. For VisualStudio, it can be done by command `lib.exe /DEF:libmgl.def /OUT:libmgl.lib`.

**How make FLTK/GLUT/Qt window which will display result of my calculations?**

You need to put yours calculations or main event-handling loop in the separate thread. For static image you can give NULL as drawing function and call `Update()` function when you need to redraw it. For more details see Section 2.1.3 [Animation], page 10.

**How I can build MathGL under Windows?**

Generally, it is the same procedure as for Linux or MacOS – see section Section 1.3 [Installation], page 2. The simplest way is using the combination CMake+MinGW. Also you may need some extra libraries like GSL, PNG, JPEG and so on. All of them can be found at <http://gnuwin32.sourceforge.net/packages.html>. After installing all components, just run `cmake-gui` (<http://www.cmake.org/cmake/help/runningcmake.html>) configurator and build the MathGL itself.

**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 into a `mglData` instance by function `[import]`, page 263, and display it by `[dens]`, page 209, function. For example, for black-and-white bitmap you can use the code: `mglData bmp; bmp.Import("fname.png", "wk"); gr->Dens(bmp, "wk");`.

**How can I use MathGL in Qt, FLTK, wxWidgets etc.?**

There are special classes (widgets) for these libraries: `QMathGL` for Qt, `FlMathGL` for FLTK and so on. If you don't find the appropriate class then you can create your own widget that displays a bitmap using `mglCanvas::GetRGB()`.

**How can I create 3D in PDF?**

Just use `WritePRC()` method which also create PDF file if `enable-pdf=ON` at MathGL configure.

**How can I create TeX figure?**

Just use `WriteTEX()` method 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`.

**Can I use MathGL in JavaScript?**

Yes, sample JavaScript file is located in `texinfo/` folder of sources. You should provide JSON data with 3d image for it (can be created by `WriteJSON()` method). Script allows basic manipulation with plot: zoom, rotation, shift. Sample of JavaScript pictures can be found in <http://mathgl.sf.net/json.html>.

**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](http://sourceforge.net/project/showfiles.php?group_id=152187&package_id=267177)). Next, you should load the font files into mglGraph class instance *gr* by the following command: `gr->LoadFont(fontname,path);`. Here *fontname* is the base font name like 'STIX' and *path* sets the location of font files. Use `gr->RestoreFont();` to start using the default font.

**How can I draw tick out of a bounding box?**

Just set a negative value in `[ticklen]`, page 171. For example, use `gr->SetTickLen(-0.1);`.

**How can I prevent text rotation?**

Just use `SetRotatedText(false)`. Also you can use axis style 'U' for disable only tick labels rotation.

**What is \*.so? What is gcc? How I can use make?**

They are standard GNU tools. There is special FAQ about its usage under Windows – <http://www.mingw.org/wiki/FAQ>.

**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 `LoadBackground("fname.png");`.

**How I can reduce "white" edges around bounding box?**

The simplest way is to use `[subplot]`, page 172, style. However, you should be careful if you plan to add `[colorbar]`, page 191, or rotate plot – part of plot can be invisible if you will use non-default `[subplot]`, page 172, 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 184, 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:

```
gr->Surf(x, y, z);
gr->Rasterize(); // make surface as bitmap
gr->Axis();
gr->WriteFrame("fname.eps");
```

**Why I couldn't use name 'I' for variable?**

MathGL support C99 standard, where 'I' is reserved for imaginary unit. If you still need this name, then just use

```
#undef I
```

after including MathGL header files.

**How I can create MPEG video from plots?**

You can save each frame into JPEG with names like 'frame0001.jpg', 'frame0002.jpg', ... Later you can use ImageMagic to convert them into

MPEG video by command `convert frame*.jpg movie.mpg`. See also [MPEG], page 11.

### 3 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 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.

#### 3.1 Coordinate axes

Two axis representations are used in MathGL. The first one consists of normalizing coordinates of data points in axis range (see Section 4.3 [Axis settings], page 166). If `SetCut()` is

**true** then the outlier points are omitted, otherwise they are projected to the bounding box (see Section 4.2.5 [Cutting], page 161). 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 4.3.2 [Curved coordinates], page 167). Finally, the data point is plotted by one of the functions.

The range of  $x$ ,  $y$ ,  $z$ -axis can be specified by `SetRange()` or `SetRanges()` functions. Its origin is specified by `SetOrigin()` function. At this 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 `SetRanges()` function. Also, one can directly set it by call `SetRange('c', ...)`. Use `Colorbar()` function for drawing the colorbar.

The form (appearance) of tick labels is controlled by `SetTicks()` function (see Section 4.3.3 [Ticks], page 168). Function `SetTuneTicks` switches on/off tick enhancing by factoring out a common 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.

## 3.2 Color styles

Base colors are defined by one of symbol `'wkrghbcymhRGBCYMHwlenupqLENUPQ'`. 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.

## 3.3 Line styles

The line style is defined by the string which may contain specifications for color (`'wkrghbcymhRGBCYMHwlenupqLENUPQ'`), dashing style (`'-|;:ji='` or space), width (`'123456789'`) and marks (`'*o+xs.^v<>'` and `#` modifier). If one of the type of information is omitted then default values used with next color from palette (see Section 4.2.7 [Palette and colors], page 162). Note, that internal color counter will be nullified by any change of palette. This includes even hidden change (for example, by

`Box()` or `Axis()` 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’.

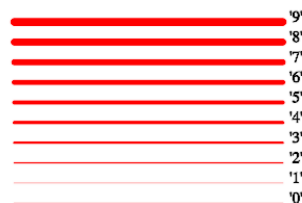
Dashing style has the following meaning: space – no line (usable for plotting only marks), ‘-’ – solid line (#####), ‘|’ – long dashed line (#####-----), ‘;’ – dashed line (####-###-), ‘=’ – small dashed line (##--##--##--##--), ‘:’ – dotted line (#---#---#---#---), ‘j’ – dash-dotted line (#####-###-), ‘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 4.11 [1D plotting], page 194).

• ‘.’ = ‘#.’	—— Solid ‘-’	↔ Style ‘AA’	→ Style ‘A’ or ‘A_’
• ‘+’ = ‘#+’	— — Long Dash ‘ ’	↔ Style ‘VV’	→ Style ‘V’ or ‘V_’
• ‘x’ = ‘#x’	— — — — — Dash ‘;’	↔ Style ‘KK’	→ Style ‘K’ or ‘K_’
• ‘#*’ = ‘#*’	— — — — — Small dash ‘=’	↔ Style ‘II’	→ Style ‘I’ or ‘I_’
• ‘s’ = ‘#s’	— — — — — Dash-dot ‘j’	↔ Style ‘DD’	→ Style ‘D’ or ‘D_’
• ‘d’ = ‘#d’	— — — — — Small dash-dot ‘i’	↔ Style ‘SS’	→ Style ‘S’ or ‘S_’
• ‘o’ = ‘#o’	— — — — — Dots ‘:’	↔ Style ‘OO’	→ Style ‘O’ or ‘O_’
• ‘^’ = ‘#^’	— — — — — None ‘_’	↔ Style ‘TT’	→ Style ‘T’ or ‘T_’
• ‘v’ = ‘#v’	— — — — — Manual ‘{df090}’	↔ Style ‘XX’	→ Style ‘X’ or ‘X_’
• ‘<’ = ‘#<’		↔ Style ‘_’	→ Style ‘_’ or none
• ‘>’ = ‘#>’		↔ Style ‘VA’	→ Style ‘AS’
		↔ Style ‘AV’	→ Style ‘_A’

{r1}	{r3}	{r5}	{r7}	{r9}
b	g	r	h	w
B	G	R	H	W
c	m	y	w	p
C	M	Y	k	P
l	e	n	u	q
L	E	N	U	Q
{xff9966}	{x83CAFF}			





### 3.4 Color scheme

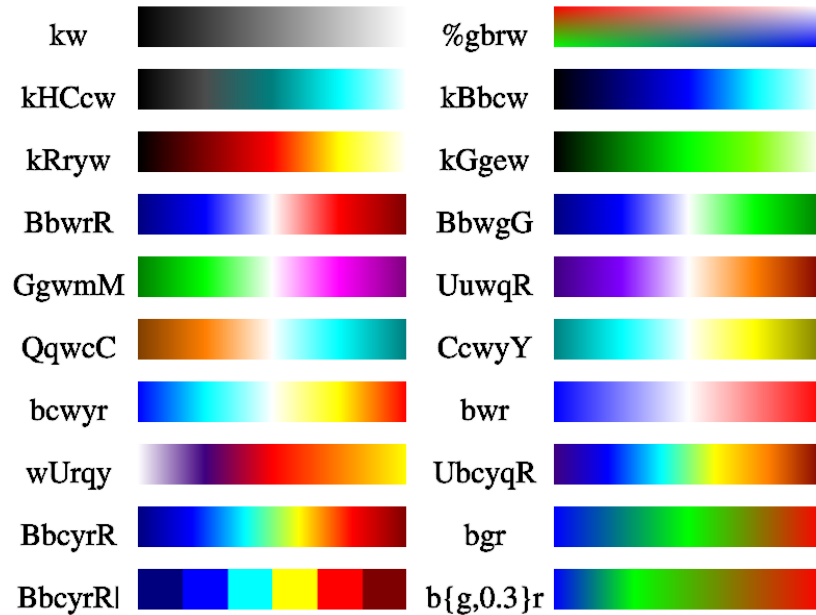
The color scheme is used for determining the color of surfaces, isolines, isosurfaces and so on. The color scheme is defined by the string, which may contain several characters that are color id (see Section 3.3 [Line styles], page 147) or characters ‘#:|’. Symbol ‘#’ switches to mesh drawing or to a 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 3.2 [Color styles], page 147). The argument is the amplitude normalized in color range (see Section 4.3 [Axis settings], page 166). 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 221, 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}.

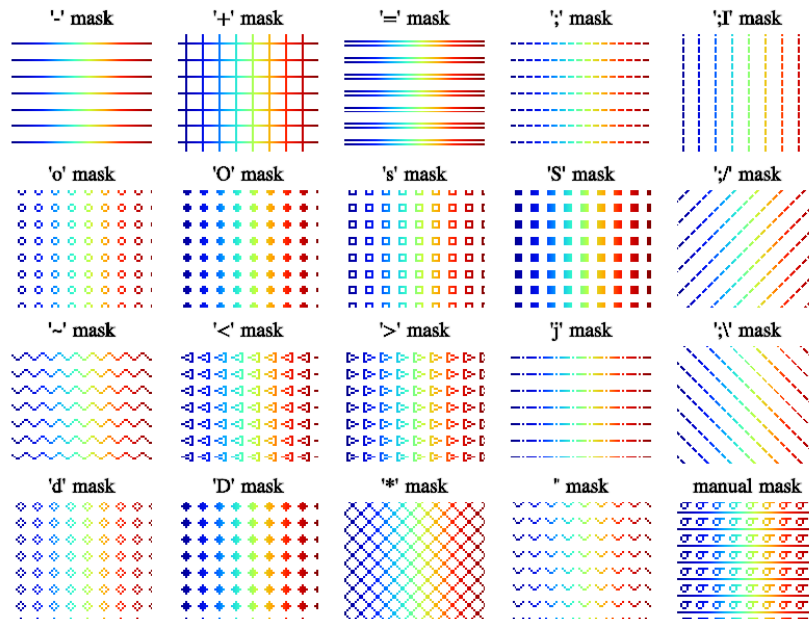
There are several useful combinations. String ‘kw’ corresponds to the simplest gray color scheme where higher values are brighter. String ‘wk’ presents the inverse gray color scheme where higher value is darker. Strings ‘kRryw’, ‘kGgw’, ‘kBbcw’ present the well-known *hot*, *summer* and *winter* color 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 4.3.2 [Curved coordinates], page 167).



When coloring by *coordinate* (used in [map], page 221), 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 ‘-+=;o0sS~<>jdD\*^’ in color scheme. Mask can be rotated by arbitrary angle by command [mask], page 163, 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 163, command. For example, the right-down subplot on the figure above is produced by code

```
gr->SetMask('+', "ff00182424f800"); gr->Dens(a, "3+");
or just use manual mask style (for v.2.3 and later)
gr->Dens(a, "3{s00ff00182424f800}");
```

### 3.5 Font styles

Text style is specified by the string which may contain: color id characters ‘`wkrgbcymhRGCYMHW`’ (see Section 3.2 [Color styles], page 147), 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 3.4 [Color scheme], page 149).

The font styles are: ‘r’ – roman (or regular) font, ‘i’ – italic style, ‘b’ – bold style. 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. 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` – overlined, `\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 3.2 [Color styles], page 147). 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  $\alpha$  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:  $\alpha$  - \alpha,  $\beta$  - \beta,  $\gamma$  - \gamma,  $\delta$  - \delta,  $\epsilon$  - \epsilon,  $\eta$  - \eta,  $\iota$  - \iota,  $\chi$  - \chi,  $\kappa$  - \kappa,  $\lambda$  - \lambda,  $\mu$  - \mu,  $\nu$  - \nu,  $\omicron$  - \omicron,  $\omega$  - \omega,  $\phi$  - \phi,  $\pi$  - \pi,  $\psi$  - \psi,  $\rho$  - \rho,  $\sigma$  - \sigma,  $\theta$  - \theta,  $\tau$  - \tau,  $\upsilon$  - \upsilon,  $\xi$  - \xi,  $\zeta$  - \zeta,  $\varsigma$  - \varsigma,  $\varepsilon$  - \varepsilon,  $\vartheta$  - \vartheta,  $\varphi$  - \varphi,  $\text{A}$  - \text{A},  $\text{B}$  - \text{B},  $\text{G}$  - \text{G},  $\Delta$  - \Delta,  $\text{E}$  - \text{E},

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, ≤ – \leq, ← – \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, ∠ – \angle, ♠ – \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 **SetFontPointSize()**, **SetFontSizeCM()**, **SetFontSizeIN()** (see Section 4.2.6 [Font settings], page 161) allow one to set it in more “common” variables for a given dpi value of the picture.

### 3.6 Textual formulas

MathGL have the fast variant of textual formula evaluation (see Section 6.12 [Evaluate expression], page 282) . 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  $\pi = 3.1415926\dots$ ,  $inf = \infty$

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  $x \neq 0$ , ‘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  $\zeta(s) = \sum_{k=1}^\infty k^{-s}$  for arbitrary  $s \neq 1$ , ‘eta(x)’ – eta function  $\eta(s) = (1-2^{1-s})\zeta(s)$  for arbitrary s, ‘lp(1,x)’ – Legendre polynomial  $P_l(x)$ , ( $|x| \leq 1$ ,  $l \geq 0$ ), ‘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  $x \geq 0$ .

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.

### 3.7 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 2.9.18 [Using options], page 137, 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 158.

**xrange val1 val2** [MGL option]

Sets boundaries of x coordinate change for the plot. See also [xrange], page 166.

**yrange val1 val2** [MGL option]

Sets boundaries of y coordinate change for the plot. See also [yrange], page 166.

- zrange val1 val2** [MGL option]  
 Sets boundaries of z coordinate change for the plot. See also [zrange], page 166.
- cut val** [MGL option]  
 Sets whether to cut or to project the plot points lying outside the bounding box. See also [cut], page 161.
- size val** [MGL option]  
 Sets the size of text, marks and arrows. See also [font], page 161, [marksize], page 160, [arrowsize], page 160.
- meshnum val** [MGL option]  
 Work like [meshnum], page 160, 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 4.11 [1D plotting], page 194, command. See also [legend], page 193.
- value val** [MGL option]  
 Set the value to be used as additional numeric parameter in plotting command.

## 3.8 Interfaces

The MathGL library has interfaces for a set of languages. Most of them are based on the C interface via SWIG tool. There are Python, Java, Octave, Lisp, C#, Guile, Lua, Modula 3, Ocaml, Perl, PHP, Pike, R, Ruby, and Tcl interfaces. Also there is a Fortran interface which has a similar set of functions, but slightly different types of arguments (integers instead of pointers). These functions are marked as [C function].

Some of the languages listed above support classes (like C++ or Python). The name of functions for them is the same as in C++ (see Chapter 4 [MathGL core], page 157, and Chapter 6 [Data processing], page 251) and marked like [Method on mglGraph].

Finally, a special command language MGL (see Chapter 7 [MGL scripts], page 285) was written for a faster access to plotting functions. Corresponding scripts can be executed separately (by UDAV, mglconv, mglview and so on) or from the C/C++/Python/... code (see Section 7.4 [mglParse class], page 291).

### 3.8.1 C/Fortran interface

The C interface is a base for many other interfaces. It contains the pure C functions for most of the methods of MathGL classes. In distinction to C++ classes, C functions must have an argument HMGL (for graphics) and/or HMDT (for data arrays), which specifies the object for drawing or manipulating (changing). So, firstly, the user has to create this object by the function `mgl_create_*`() and has to delete it after the use by function `mgl_delete_*`().

All C functions are described in the header file `#include <mgl2/mgl_cf.h>` and use variables of the following types:

- HMGL — Pointer to class `mglGraph` (see Chapter 4 [MathGL core], page 157).
- HCDT — Pointer to class `const mglDataA` (see Chapter 6 [Data processing], page 251) — constant data array.

- **HMDT** — Pointer to class **mglData** (see Chapter 6 [Data processing], page 251) — data array of real numbers.
- **HADT** — Pointer to class **mglDataC** (see Chapter 6 [Data processing], page 251) — data array of complex numbers.
- **HMPR** — Pointer to class **mglParse** (see Section 7.4 [mglParse class], page 291) — MGL script parsing.
- **HMEX** — Pointer to class **mglExpr** (see Section 6.12 [Evaluate expression], page 282) — textual formulas for real numbers.
- **HMAX** — Pointer to class **mglExprC** (see Section 6.12 [Evaluate expression], page 282) — textual formulas for complex numbers.

These variables contain identifiers for graphics drawing objects and for the data objects.

Fortran functions/subroutines have the same names as C functions. However, there is a difference. Variable of type **HMGL**, **HMDT** must be an integer with sufficient size (**integer\*4** in the 32-bit operating system or **integer\*8** in the 64-bit operating system). All C functions of type **void** are subroutines in Fortran, which are called by operator **call**. The exceptions are functions, which return variables of types **HMGL** or **HMDT**. These functions should be declared as integer in Fortran code. Also, one should keep in mind that strings in Fortran are denoted by ' symbol, not the " symbol.

### 3.8.2 C++/Python interface

MathGL provides the interface to a set of languages via SWIG library. Some of these languages support classes. The typical example is Python – which is named in this chapter's title. Exactly the same classes are used for high-level C++ API. Its feature is using only inline member-functions what make high-level API to be independent on compiler even for binary build.

There are 3 main classes in:

- **mglGraph** – provide most plotting functions (see Chapter 4 [MathGL core], page 157).
- **mglData** – provide base data processing (see Chapter 6 [Data processing], page 251). It have an additional feature to access data values. You can use a construct like this: **dat[i]=sth**; or **sth=dat[i]** where flat representation of data is used (i.e., *i* can be in range 0...*nx\*nx\*nz-1*). You can also import NumPy arrays as input arguments in Python: **mgl\_dat = mglData(numpy\_dat);**.
- **mglParse** – provide functions for parsing MGL scripts (see Chapter 7 [MGL scripts], page 285).

To use Python classes just execute **'import mathgl'**. The simplest example will be:

```
import mathgl
a=mathgl.mglGraph()
a.Box()
a.WritePNG("test.png")
```

Alternatively you can import all classes from **mathgl** module and easily access MathGL classes like this:

```
from mathgl import *
a=mglGraph()
```

```
a.Box()  
a.WritePNG("test.png")
```

This becomes useful if you create many `mgldata` objects, for example.



## 4 MathGL core

The core of MathGL is **mglGraph** class defined in `#include <mgl2/mgl.h>`. It contains a lot of plotting functions for 1D, 2D and 3D data. It also encapsulates parameters for axes drawing. Moreover an arbitrary coordinate transformation can be used for each axis. All plotting functions use data encapsulated in **mglData** class (see Chapter 6 [Data processing], page 251) that allows to check sizes of used arrays easily. Also it have many functions for data handling: modify it by formulas, find momentums and distribution (histogram), apply operator (differentiate, integrate, transpose, Fourier and so on), change data sizes (interpolate, squeeze, crop and so on). Additional information about colors, fonts, formula parsing can be found in Chapter 3 [General concepts], page 146, and Chapter 9 [Other classes], page 303.

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

```
version 'ver' [MGL command]
bool CheckVersion (const char *ver) static [Method on mglGraph]
int mgl_check_version (const char *ver) [C function]
    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.
```

### 4.1 Create and delete objects

```
mglGraph (int kind=0, int width=600, int height=400) [Constructor on mglGraph]
mglGraph (const mglGraph &gr) [Constructor on mglGraph]
mglGraph (HMGL gr) [Constructor on mglGraph]
HMGL mgl_create_graph (int width, int height) [C function]
HMGL mgl_create_graph_gl () [C function]
    Creates the instance of class mglGraph with specified sizes width and height. Pa-
    rameter kind may have following values: '0' – use default plotter, '1' – use OpenGL
    plotter.

~mglGraph () [Destructor on mglGraph]
HMGL mgl_delete_graph (HMGL gr) [C function]
    Deletes the instance of class mglGraph.

HMGL Self () [Method on mglGraph]
    Returns the pointer to internal object of type HMGL.
```

### 4.2 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]
void DefaultPlotParam () [Method on mglGraph]
void mgl_set_def_param (HMGL gr) [C function]
    Restore initial values for all of parameters and clear the image.
```

### 4.2.1 Transparency

There are several functions and variables for setup transparency. The general function is `[alpha]`, page 158, which switch on/off the transparency for overall plot. It influence only for graphics which created after `[alpha]`, page 158, call (with one exception, OpenGL). Function `[alphadef]`, page 158, specify the default value of alpha-channel. Finally, function `[transptype]`, page 158, set the kind of transparency. See Section 2.9.2 [Transparency and lighting], page 112, for sample code and picture.

```
alpha [val=on] [MGL command]
void Alpha (bool enable) [Method on mglGraph]
void mgl_set_alpha (HMGL gr, int enable) [C function]
    Sets the transparency on/off and returns previous value of transparency. It is recom-
    mended to call this function before any plotting command. Default value is trans-
    parency off.
```

```
alphadef val [MGL command]
void SetAlphaDef (mreal val) [Method on mglGraph]
void mgl_set_alpha_default (HMGL gr, mreal alpha) [C function]
    Sets default value of alpha channel (transparency) for all plotting functions. Initial
    value is 0.5.
```

```
transptype val [MGL command]
void SetTranspType (int type) [Method on mglGraph]
void mgl_set_transp_type (HMGL gr, int type) [C function]
    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 2.9.3 [Types of transparency], page 113, for sample code and picture..

### 4.2.2 Lighting

There are several functions for setup lighting. The general function is `[light]`, page 158, which switch on/off the lighting for overall plot. It influence only for graphics which created after `[light]`, page 158, 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 2.9.6 [Lighting sample], page 117, for sample code and picture.

```
light [val=on] [MGL command]
bool Light (bool enable) [Method on mglGraph]
```

```

void mgl_set_light (HMGL gr, int enable) [C function]
    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]
void Light (int n, bool enable) [Method on mglGraph]
void mgl_set_light_n (HMGL gr, int n, int enable) [C function]
    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]
void AddLight (int n, mglPoint d, char c='w', mreal bright=0.5, mreal ap=0) [Method on mglGraph]
void AddLight (int n, mglPoint r, mglPoint d, char c='w', mreal bright=0.5, mreal ap=0) [Method on mglGraph]
void mgl_add_light (HMGL gr, int n, mreal dx, mreal dy, mreal dz) [C function]
void mgl_add_light_ext (HMGL gr, int n, mreal dx, mreal dy, mreal dz, char c, mreal bright, mreal ap) [C function]
void mgl_add_light_loc (HMGL gr, int n, mreal rx, mreal ry, mreal rz, mreal dx, mreal dy, mreal dz, char c, mreal bright, mreal ap) [C function]
    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]
void SetDiffuse (mreal bright) [Method on mglGraph]
void mgl_set_difbr (HMGL gr, mreal bright) [C function]
    Set brightness of diffusive light (only for local light sources).

ambient val [MGL command]
void SetAmbient (mreal bright=0.5) [Method on mglGraph]
void mgl_set_ambbr (HMGL gr, mreal bright) [C function]
    Sets the brightness of ambient light. The value should be in range [0,1].

attachlight val [MGL command]
void AttachLight (bool val) [Method on mglGraph]
void mgl_set_attach_light (HMGL gr, int val) [C function]
    Set to attach light settings to [inplot], page 173/[subplot], page 172. Note, OpenGL
    and some output formats don't support this feature.

```

### 4.2.3 Fog

```

fog val [dz=0.25] [MGL command]
void Fog (mreal d, mreal dz=0.25) [Method on mglGraph]
void mgl_set_fog (HMGL gr, mreal d, mreal dz) [C function]
    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  $\sim 1 - \exp(-d \cdot z)$ . 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 2.9.5 [Adding fog], page 116, for sample code and picture.

#### 4.2.4 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]
void SetBarWidth (mreal val) [Method on mglGraph]
void mgl_set_bar_width (HMGL gr, mreal val) [C function]
    Sets relative width of rectangles in [bars], page 198, [barh], page 198, [boxplot],
    page 199, [candle], page 200, [ohlc], page 201. Default value is 0.7.

marksize val [MGL command]
void SetMarkSize (mreal val) [Method on mglGraph]
void mgl_set_mark_size (HMGL gr, mreal val) [C function]
    Sets size of marks for Section 4.11 [1D plotting], page 194. Default value is 1.

arrowsize val [MGL command]
void SetArrowSize (mreal val) [Method on mglGraph]
void mgl_set_arrow_size (HMGL gr, mreal val) [C function]
    Sets size of arrows for Section 4.11 [1D plotting], page 194, lines and curves (see
    Section 4.7 [Primitives], page 184). Default value is 1.

meshnum val [MGL command]
void SetMeshNum (int val) [Method on mglGraph]
void mgl_set_meshnum (HMGL gr, int num) [C function]
    Sets approximate number of lines in [mesh], page 207, [fall], page 208, [grid2],
    page 213, and also the number of hachures in [vect], page 222, [dew], page 224,
    and the number of cells in [cloud], page 214. By default (=0) it draws all
    lines/hachures/cells.

facenum val [MGL command]
void SetFaceNum (int val) [Method on mglGraph]
void mgl_set_facenum (HMGL gr, int num) [C function]
    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]
void SetPlotId (const char *id) [Method on mglGraph]
void mgl_set_plotid (HMGL gr, const char *id) [C function]
    Sets default name id as filename for saving (in FLTK window for example).

const char * GetPlotId () [Method on mglGraph]
const char * mgl_get_plotid (HMGL gr) [C function only]
mgl_get_plotid (long gr, char *out, int len) [Fortran subroutine]
    Gets default name id as filename for saving (in FLTK window for example).
```

```
pendelta val [MGL command]
void SetPenDelta (double val) [Method on mglGraph]
void mgl_pen_delta (HMGL gr, double val) [C function]
    Changes the blur around lines and text (default is 1). For val>1 the text and lines
    are more sharpened. For val<1 the text and lines are more blurred.
```

### 4.2.5 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 2.2.9 [Cutting sample], page 34, for sample code and picture.

```
cut val [MGL command]
void SetCut (bool val) [Method on mglGraph]
void mgl_set_cut (HMGL gr, int val) [C function]
    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]
void SetCutBox (mglPoint p1, mglPoint p1) [Method on mglGraph]
void mgl_set_cut_box (HMGL gr, mreal x1, mreal y1, mreal z1, [C function]
    mreal x2, mreal y2, mreal z2)
    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]
void CutOff (const char *cond) [Method on mglGraph]
void mgl_set_cutoff (HMGL gr, const char *cond) [C function]
    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.
```

### 4.2.6 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 3.5 [Font styles], page 151.
```

```
rotatetext val [MGL command]
void SetRotatedText (bool val) [Method on mglGraph]
void mgl_set_rotated_text (HMGL gr, int val) [C function]
    Sets to use or not text rotation.
```

```
loadfont ['name'="] [MGL command]
void LoadFont (const char *name, const char [Method on mglGraph]
    *path="")
```

`void mgl_load_font (HMGL gr, const char *name, const char *path)` [C function]

Load font typeface from *path/name*. Empty name will load default font.

`void SetFontDef (const char *fnt)` [Method on `mglGraph`]

`void mgl_set_font_def (HMGL gr, const char * val)` [C function]  
Sets the font specification (see Section 4.8 [Text printing], page 188). Default is 'rC' – Roman font centering.

`void SetFontSize (mreal val)` [Method on `mglGraph`]

`void mgl_set_font_size (HMGL gr, mreal val)` [C function]  
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.

`void SetFontSizePT (mreal cm, int dpi=72)` [Method on `mglGraph`]

Set FontSize by size in pt and picture DPI (default is 16 pt for dpi=72).

`inline void SetFontSizeCM (mreal cm, int dpi=72)` [Method on `mglGraph`]

Set FontSize by size in centimeters and picture DPI (default is 0.56 cm = 16 pt).

`inline void SetFontSizeIN (mreal cm, int dpi=72)` [Method on `mglGraph`]

Set FontSize by size in inch and picture DPI (default is 0.22 in = 16 pt).

`void LoadFont (const char *name, const char *path= "")` [Method on `mglGraph`]

`void mgl_load_font (HMGL gr, const char *name, const char *path)` [C function]  
Load font typeface from *path/name*.

`void CopyFont (mglGraph * from)` [Method on `mglGraph`]

`void mgl_copy_font (HMGL gr, HMGL gr_from)` [C function]  
Copy font data from another `mglGraph` object.

`void RestoreFont ()` [Method on `mglGraph`]

`void mgl_restore_font (HMGL gr)` [C function]  
Restore font data to default typeface.

`void SetDefFont (const char *name, const char *path= "") static` [Method on `mglGraph`]

`void mgl_def_font (const char *name, const char *path)` [C function]  
Load default font typeface (for all newly created HMGL/`mglGraph` objects) from *path/name*.

#### 4.2.7 Palette and colors

`palette 'colors'` [MGL command]

`void SetPalette (const char *colors)` [Method on `mglGraph`]

`void mgl_set_palette (HMGL gr, const char *colors)` [C function]  
Sets the palette as selected colors. Default value is "Hbgrcmynlneup" 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 4.11 [1D plotting], page 194) 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 192, or [axis], page 190, functions).

```
void SetDefScheme (const char *sch) [Method on mglGraph]
void mgl_set_def_sch (HMGL gr, const char *sch) [C function]
    Sets the sch as default color scheme. Default value is "BbcyrR".

void SetColor (char id, mreal r, mreal g, mreal b) [Method on mglGraph]
    static
void mgl_set_color (char id, mreal r, mreal g, mreal b) [C function]
    Sets RGB values for color with given id. This is global setting which influence on any later usage of symbol id.
```

### 4.2.8 Masks

```
mask 'id' 'hex' [MGL command]
void SetMask (char id, const char *hex) [Method on mglGraph]
void SetMask (char id, uint64_t hex) [Method on mglGraph]
void mgl_set_mask (HMGL gr, const char *hex) [C function]
void mgl_set_mask_val (HMGL gr, uint64_t hex) [C function]
    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 3.4 [Color scheme], page 149): '-' 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]
void SetMaskAngle (int angle) [Method on mglGraph]
void mgl_set_mask_angle (HMGL gr, int angle) [C function]
    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.
```

### 4.2.9 Error handling

Normally user should set it to zero by `SetWarn(0)`; before plotting and check if `GetWarn()` or `Message()` return non zero after plotting. Only last warning will be saved. All warnings/errors produced by MathGL is not critical – the plot just will not be drawn. By default, all warnings are printed in stderr. You can disable it by using `mgl_suppress_warn(true)`;

```
void SetWarn (int code, const char *info="") [Method on mglGraph]
void mgl_set_warn (HMGL gr, int code, const char *info) [C function]
    Set warning code. Normally you should call this function only for clearing the warning state, i.e. call SetWarn(0); . Text info will be printed as is if code<0.
```

```

const char *Message ()                                [Method on mglGraph]
const char *mgl_get_mess (HMGL gr)                    [C function only]
mgl_get_mess (long gr, char *out, int len)             [Fortran subroutine]
    Return messages about matters why some plot are not drawn. If returned string is
    empty then there are no messages.

int GetWarn ()                                         [Method on mglGraph]
int mgl_get_warn (HMGL gr)                             [C function]
    Return the numerical ID of warning about the not drawn plot. Possible values are:

    mglWarnNone=0
        Everything OK

    mglWarnDim
        Data dimension(s) is incompatible

    mglWarnLow
        Data dimension(s) is too small

    mglWarnNeg
        Minimal data value is negative

    mglWarnFile
        No file or wrong data dimensions

    mglWarnMem
        Not enough memory

    mglWarnZero
        Data values are zero

    mglWarnLeg
        No legend entries

    mglWarnSlc
        Slice value is out of range

    mglWarnCnt
        Number of contours is zero or negative

    mglWarnOpen
        Couldn't open file

    mglWarnLId
        Light: ID is out of range

    mglWarnSize
        Setsize: size(s) is zero or negative

    mglWarnFmt
        Format is not supported for that build

    mglWarnTern
        Axis ranges are incompatible

    mglWarnNull
        Pointer is NULL

```



`mglWarnSpc`  
Not enough space for plot

`mglScrArg`  
Wrong argument(s) of a command in MGL script

`mglScrCmd`  
Wrong command in MGL script

`mglScrLong`  
Too long line in MGL script

`mglScrStr`  
Unbalanced ' in MGL script

`mglScrTemp`  
Change temporary data in MGL script

`void SuppressWarn (bool state) static` [Method on `mglGraph`]  
`void mgl_suppress_warn (int state)` [C function]  
 Disable printing warnings to `stderr` if *state* is nonzero.

`void SetGlobalWarn (const char *info) static` [Method on `mglGraph`]  
`void mgl_set_global_warn (const char *info)` [C function]  
 Set warning message *info* for global scope.

`const char * GlobalWarn () static` [Method on `mglGraph`]  
`const char * mgl_get_global_warn ()` [C function]  
 Get warning message(s) for global scope.

#### 4.2.10 Stop drawing

`void Stop (bool stop=true)` [Method on `mglGraph`]  
`void mgl_ask_stop (HMGL gr, int stop)` [C function only]  
 Ask to stop drawing if *stop* is non-zero, otherwise reset stop flag.

`bool NeedStop ()` [Method on `mglGraph`]  
`void mgl_need_stop (HMGL gr)` [C function only]  
 Return `true` if drawing should be terminated. Also it process all events in GUI.  
 User should call this function from time to time inside a long calculation to allow processing events for GUI.

`bool SetEventFunc (void (*func)(void *), void` [Method on `mglGraph`]  
`*par=NULL)`  
`void mgl_set_event_func (HMGL gr, void (*func)(void *),` [C function only]  
`void *par)`  
 Set callback function which will be called to process events of GUI library.

### 4.3 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 4.2.5 [Cutting], page 161), 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.

#### 4.3.1 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]
```

```
void SetRange (char dir, mreal v1, mreal v2) [Method on mglGraph]
void AddRange (char dir, mreal v1, mreal v2) [Method on mglGraph]
void mgl_set_range_val (HMGL gr, char dir, mreal v1, mreal v2) [C function]
void mgl_add_range_val (HMGL gr, char dir, mreal v1, mreal v2) [C function]
    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 166.
```

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

```
void SetRange (char dir, const mglDataA &dat, bool [Method on mglGraph]
    add=false)
void mgl_set_range_dat (HMGL gr, char dir, const HCDT a, int [C function]
    add)
    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]
void SetRanges (mglPoint p1, mglPoint p2) [Method on mglGraph]
void SetRanges (double x1, double x2, double y1, [Method on mglGraph]
    double y2, double z1=0, double z2=0)
void mgl_set_ranges (HMGL gr, double x1, double x2, double y1, [C function]
    double y2, double z1, double z2)
```

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].

```
void SetRanges (const mglDataA &xx, const mglDataA [Method on mglGraph]
    &yy)
void SetRanges (const mglDataA &xx, const mglDataA [Method on mglGraph]
    &yy, const mglDataA &zz)
```

```
void SetRanges (const mglDataA &xx, const mglDataA &yy, const mglDataA &zz, const mglDataA &cc) [Method on mglGraph]
```

Sets the ranges of 'x', 'y', 'z'-coordinates and coloring as minimal and maximal values of data xx, yy, zz, cc correspondingly.

```
void SetAutoRanges (mglPoint p1, mglPoint p2) [Method on mglGraph]
```

```
void SetAutoRanges (double x1, double x2, double y1, double y2, double z1=0, double z2=0, double c1=0, double c2=0) [Method on mglGraph]
```

```
void mgl_set_auto_ranges (HMGL gr, double x1, double x2, double y1, double y2, double z1, double z2, double z1, double z2) [C function]
```

Sets the ranges for automatic coordinates. If minimal and maximal values of the coordinate are the same then they are ignored.

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

```
void SetOrigin (mglPoint p0) [Method on mglGraph]
```

```
void SetOrigin (mreal x0, mreal y0, mreal z0=NAN) [Method on mglGraph]
```

```
void mgl_set_origin (HMGL gr, mreal x0, mreal y0, mreal z0) [C function]
```

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]
```

```
void ZoomAxis (mglPoint p1, mglPoint p2) [Method on mglGraph]
```

```
void mgl_zoom_axis (HMGL gr, mreal x1, mreal y1, mreal z1, mreal c1, mreal x2, mreal y2, mreal z2, mreal c2) [C function]
```

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()**.

### 4.3.2 Curved coordinates

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

```
void SetFunc (const char *EqX, const char *EqY, const char *EqZ="", const char *EqA="") [Method on mglGraph]
```

```
void mgl_set_func (HMGL gr, const char *EqX, const char *EqY, const char *EqZ, const char *EqA) [C function]
```

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 3.6 [Textual formulas], page 152.

**axis how** [MGL command]  
**void SetCoor (int how)** [Method on **mglGraph**]  
**void mgl\_set\_coor (HMGL gr, int how)** [C function]  
 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** – Spherical 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]  
**void Ternary (int tern)** [Method on **mglGraph**]  
**void mgl\_set\_ternary (HMGL gr, int tern)** [C function]  
 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 2.2.6 [Ternary axis], page 29, for sample code and picture. See Section 2.9.4 [Axis projection], page 115, for sample code and picture.

### 4.3.3 Ticks

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

```
void Adjust (const char *dir="xyzc") [Method on mglGraph]
void mgl_adjust_ticks (HMGL gr, const char *dir) [C function]
    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]
void SetTicks (char dir, mreal d=0, int ns=0, mreal [Method on mglGraph]
    org=NAN, const char *fact="")
void SetTicks (char dir, mreal d, int ns, mreal org, [Method on mglGraph]
    const wchar_t *fact)
void mgl_set_ticks (HMGL gr, char dir, mreal d, int ns, mreal [C function]
    org)
void mgl_set_ticks_fact (HMGL gr, char dir, mreal d, int ns, [C function]
    mreal org, const char *fact)
void mgl_set_ticks_factw (HMGL gr, char dir, mreal d, int ns, [C function]
    mreal org, const wchar_t *fact)
```

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]
void SetTicksVal (char dir, const char *lbl, bool [Method on mglGraph]
    add=false)
void SetTicksVal (char dir, const wchar_t *lbl, bool [Method on mglGraph]
    add=false)
void SetTicksVal (char dir, const mglDataA &val, [Method on mglGraph]
    const char *lbl, bool add=false)
void SetTicksVal (char dir, const mglDataA &val, [Method on mglGraph]
    const wchar_t *lbl, bool add=false)
void mgl_set_ticks_str (HMGL gr, char dir, const char *lbl, [C function]
    bool add)
void mgl_set_ticks_wcs (HMGL gr, char dir, const wchar_t *lbl, [C function]
    bool add)
void mgl_set_ticks_val (HMGL gr, char dir, HCDT val, const char [C function]
    *lbl, bool add)
```

```
void mgl_set_ticks_valw (HMGL gr, char dir, HCDT val, const [C function]
                        wchar_t *lbl, bool add)
```

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.

```
void AddTick (char dir, double val, const char *lbl) [Method on mglGraph]
```

```
void AddTick (char dir, double val, const wchar_t *lbl) [Method on mglGraph]
```

```
void mgl_add_tick (HMGL gr, char dir, double val, const char [C function]
                  *lbl)
```

```
void mgl_set_tickw (HMGL gr, char dir, double val, const [C function]
                   wchar_t *lbl)
```

The same as previous but add single tick label *lbl* at position *val* to the list of existed ones.

```
xtick 'templ' [MGL command]
```

```
ytick 'templ' [MGL command]
```

```
ztick 'templ' [MGL command]
```

```
ctick 'templ' [MGL command]
```

```
void SetTickTempl (char dir, const char *templ) [Method on mglGraph]
```

```
void SetTickTempl (char dir, const wchar_t *templ) [Method on mglGraph]
```

```
void mgl_set_tick_templ (HMGL gr, const char *templ) [C function]
```

```
void mgl_set_tick_templw (HMGL gr, const wchar_t *templ) [C function]
```

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 'templ'] [MGL command]
```

```
void SetTicksTime (char dir, mreal val, const char [Method on mglGraph]
                  *templ)
```

```
void mgl_set_ticks_time (HMGL gr, mreal val, const char [C function]
                        *templ)
```

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.

```
double mgl_get_time (const char*str, const char *templ) [C function]
```

Gets number of seconds from 1970 year to specified date/time *str*. The format of string is specified by *templ*, which 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. Note, that MS Visual Studio couldn’t handle date before 1970.

```

tuneticks val [pos=1.15] [MGL command]
void SetTuneTicks (int tune, mreal pos=1.15) [Method on mglGraph]
void mgl_tune_ticks (HMGL gr, int tune, mreal pos) [C function]
    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]
void SetTickShift (mglPoint d) [Method on mglGraph]
void mgl_set_tick_shift (HMGL gr, mreal dx, mreal dy, mreal dz, mreal dc) [C function]
    Set value of additional shift for ticks labels.

void SetTickRotate (bool val) [Method on mglGraph]
void mgl_set_tick_rotate (HMGL gr, bool val) [C function]
    Enable/disable ticks rotation if there are too many ticks or ticks labels are too long.

void SetTickSkip (bool val) [Method on mglGraph]
void mgl_set_tick_skip (HMGL gr, bool val) [C function]
    Enable/disable ticks skipping if there are too many ticks or ticks labels are too long.

void SetTimeUTC (bool val) [Method on mglGraph]
    Enable/disable using UTC time for ticks labels. In C/Fortran you can use mgl_set_
    flag(gr, val, MGL_USE_GMTIME);.

origintick val [MGL command]
void SetOriginTick (bool val=true) [Method on mglGraph]
    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]
void SetTickLen (mreal val, mreal stt=1) [Method on mglGraph]
void mgl_set_tick_len (HMGL gr, mreal val, mreal stt) [C function]
    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]
void SetAxisStl (const char *stl="k", const char *tck=0, const char *sub=0) [Method on mglGraph]
void mgl_set_axis_stl (HMGL gr, const char *stl, const char *tck, const char *sub) [C function]
    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).

```

## 4.4 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 172, [multiplot], page 172, or [inplot], page 173, for specifying the place. Second one can be [title], page 173, for adding title for the subplot. After it a [rotate], page 174, and [aspect], page 174. And finally any other plotting functions may be called. Alternatively you can use [columnplot], page 173, [gridplot], page 173, [stickplot], page 173, or relative [inplot], page 173, for positioning plots in the column (or grid, or stick) one by another without gap between plot axis (bounding boxes). See Section 2.2.1 [Subplots], page 19, for sample code and picture.

```
subplot nx ny m ['stl']='<>_~' dx=0 dy=0 [MGL command]
void SubPlot (int nx, int ny, int m, const char [Method on mglGraph]
              *stl="<>_~", mreal dx=0, mreal dy=0)
void mgl_subplot (HMGL gr, int nx, int ny, int m, const char [C function]
                  *stl)
void mgl_subplot_d (HMGL gr, int nx, int ny, int m, const char [C function]
                   *stl, mreal dx, mreal dy)
```

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]
void MultiPlot (int nx, int ny, int m, int dx, int dy, [Method on mglGraph]
                const char *stl="<>_~")
void mgl_multiplot (HMGL gr, int nx, int ny, int m, int dx, int [C function]
                   dy, const char *stl)
```

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]
void InPlot (mreal x1, mreal x2, mreal y1, mreal y2, [Method on mglGraph]
            bool rel=true)
void mgl_inplot (HMGL gr, mreal x1, mreal x2, mreal y1, mreal [C function]
                y2)
void mgl_relplot (HMGL gr, mreal x1, mreal x2, mreal y1, mreal [C function]
                 y2)

```

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 172, (or [inplot], page 173, 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]
void ColumnPlot (int num, int ind, mreal d=0) [Method on mglGraph]
void mgl_columnplot (HMGL gr, int num, int ind) [C function]
void mgl_columnplot_d (HMGL gr, int num, int ind, mreal d) [C function]

```

Puts further plotting in *ind*-th cell of column with *num* cells. The position is relative to previous [subplot], page 172, (or [inplot], page 173, with `rel=false`). Parameter *d* set extra gap between cells.

```

gridplot nx ny ind [d=0] [MGL command]
void GridPlot (int nx, int ny, int ind, mreal d=0) [Method on mglGraph]
void mgl_gridplot (HMGL gr, int nx, int ny, int ind) [C function]
void mgl_gridplot_d (HMGL gr, int nx, int ny, int ind, mreal d) [C function]

```

Puts further plotting in *ind*-th cell of *nx\*ny* grid. The position is relative to previous [subplot], page 172, (or [inplot], page 173, with `rel=false`). Parameter *d* set extra gap between cells.

```

stickplot num ind tet phi [MGL command]
void StickPlot (int num, int ind, mreal tet, mreal [Method on mglGraph]
               phi)
void mgl_stickplot (HMGL gr, int num, int ind, mreal tet, mreal [C function]
                   phi)

```

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 172, (or [inplot], page 173, with `rel=false`).

```

title 'title' ['stl']=" size=-2] [MGL command]
void Title (const char *txt, const char *stl="", [Method on mglGraph]
            mreal size=-2)
void Title (const wchar_t *txt, const char *stl="", [Method on mglGraph]
            mreal size=-2)
void mgl_title (HMGL gr, const char *txt, const char *stl, [C function]
               mreal size)
void mgl_titlew (HMGL gr, const wchar_t *txt, const char *stl, [C function]
                mreal size)

```

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

- font style (see, Section 3.5 [Font styles], page 151);
- ‘#’ 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]
void Rotate (mreal TetX, mreal TetZ, mreal TetY=0) [Method on mglGraph]
void mgl_rotate (HMGL gr, mreal TetX, mreal TetZ, mreal TetY) [C function]
    Rotates a further plotting relative to each axis {x, z, y} consecutively on angles TetX,
    TetZ, TetY.
```

```
rotate tet x y z [MGL command]
void RotateN (mreal Tet, mreal x, mreal y, mreal z) [Method on mglGraph]
void mgl_rotate_vector (HMGL gr, mreal Tet, mreal x, mreal y, [C function]
    mreal z)
    Rotates a further plotting around vector {x, y, z} on angle Tet.
```

```
aspect ax ay [az=1] [MGL command]
void Aspect (mreal Ax, mreal Ay, mreal Az=1) [Method on mglGraph]
void mgl_aspect (HMGL gr, mreal Ax, mreal Ay, mreal Az) [C function]
    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 174,
    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.
```

```
void Push () [Method on mglGraph]
void mgl_mat_push (HMGL gr) [C function]
    Push transformation matrix into stack. Later you can restore its current state by
    Pop() function.
```

```
void Pop () [Method on mglGraph]
void mgl_mat_pop (HMGL gr) [C function]
    Pop (restore last 'pushed') transformation matrix into stack.
```

```
void SetPlotFactor (mreal val) [Method on mglGraph]
void mgl_set_plotfactor (HMGL gr, mreal val) [C function]
    Sets the factor of plot size. It is not recommended to set it lower then 1.5. This
    is some analogue of function Zoom() but applied not to overall image but for each
    InPlot. Use negative value or zero to enable automatic selection.
```

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]
void Perspective (mreal a) [Method on mglGraph]
```

`void mgl_perspective (HMGL gr, mreal a)` [C function]  
 Add (switch on) the perspective to plot. The parameter  $a = \text{Depth}/(\text{Depth} + dz) \in [0, 1)$ . By default ( $a=0$ ) the perspective is off.

`view tetx tetz [tety=0]` [MGL command]  
`void View (mreal TetX, mreal TetZ, mreal TetY=0)` [Method on `mglGraph`]  
`void mgl_view (HMGL gr, mreal TetX, mreal TetZ, mreal TetY)` [C function]  
 Rotates a further plotting relative to each axis  $\{x, z, y\}$  consecutively on angles *TetX*, *TetZ*, *TetY*. Rotation is done independently on [rotate], page 174. 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]  
`void Zoom (mreal x1, mreal y1, mreal x2, mreal y2)` [Method on `mglGraph` (C++, Python)]  
`void mgl_set_zoom (HMGL gr, mreal x1, mreal y1, mreal x2, mreal y2)` [C function]

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  $[x1, x2] \times [y1, y2]$ . 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.

## 4.5 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]  
`void SetSize (int width, int height, bool clear=true)` [Method on `mglGraph`]  
`void mgl_set_size (HMGL gr, int width, int height)` [C function]  
`void mgl_scale_size (HMGL gr, int width, int height)` [C function]  
 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*.

`void SetSizeScl (double factor)` [Method on `mglGraph`]  
`void mgl_set_size_scl (HMGL gr, double factor)` [C function]  
 Set factor for width and height in all further calls of [setsize], page 175.

`quality [val=2]` [MGL command]  
`void SetQuality (int val=MGL_DRAW_NORM)` [Method on `mglGraph`]  
`void mgl_set_quality (HMGL gr, int val)` [C function]  
 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).

```

int GetQuality () [Method on mglGraph]
int mgl_get_quality (HMGL gr) [C function]
    Gets quality of the plot: 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).

void StartGroup (const char *name) [Method on mglGraph]
void mgl_start_group (HMGL gr, const char *name) [C function]
    Starts group definition. Groups contain objects and other groups, they are used to
    select a part of a model to zoom to or to make invisible or to make semitransparent
    and so on.

void EndGroup () [Method on mglGraph]
void mgl_end_group (HMGL gr) [C function]
    Ends group definition.

```

#### 4.5.1 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]
void WriteFrame (const char *fname="", const char [Method on mglGraph]
    *descr="")
void mgl_write_frame (HMGL gr, const char *fname, const char [C function]
    *descr)
    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 160, class property.

void WritePNG (const char *fname, const char [Method on mglGraph]
    *descr="", int compr="", bool alpha=true)
void mgl_write_png (HMGL gr, const char *fname, const char [C function]
    *descr)
void mgl_write_png_solid (HMGL gr, const char *fname, const [C function]
    char *descr)
    Exports current frame to PNG file. Parameter fname specifies the file name, descr
    adds description to file, alpha gives the transparency type. By default there are no
    description added and semitransparent image used. This function does nothing if
    HAVE_PNG isn't defined during compilation of MathGL library.

```

```
void WriteJPEG (const char *fname, const char          [Method on mglGraph]
                *descr="")
```

```
void mgl_write_jpg (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to JPEG file. Parameter *fname* specifies the file name, *descr* adds description to file. By default there is no description added. This function does nothing if HAVE\_JPEG isn't defined during compilation of MathGL library.

```
void WriteGIF (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_gif (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to GIF file. Parameter *fname* specifies the file name, *descr* adds description to file. By default there is no description added. This function does nothing if HAVE\_GIF isn't defined during compilation of MathGL library.

```
void WriteBMP (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_bmp (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to BMP file. Parameter *fname* specifies the file name, *descr* adds description to file. There is no compression used.

```
void WriteTGA (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_tga (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to TGA file. Parameter *fname* specifies the file name, *descr* adds description to file. There is no compression used.

```
void WriteEPS (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_eps (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to EPS file using vector representation. So it is not recommended for the export of large data plot. It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file. By default there is no description added. If file name is terminated by 'z' (for example, 'fname.eps.gz') then file will be compressed in gzip format. Note, that EPS format don't support color interpolation, and the resulting plot will look as you use [quality], page 175=1 for plotting.

```
void WriteBPS (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_eps (HMGL gr, const char *fname, const char      [C function]
                  *descr)
```

Exports current frame to EPS file using bitmap representation. Parameter *fname* specifies the file name, *descr* adds description to file. By default there is no description

added. If file name is terminated by ‘z’ (for example, ‘*fname.eps.gz*’) then file will be compressed in gzip format.

```
void WriteSVG (const char *fname, const char          [Method on mglGraph]
               *descr="")
void mgl_write_svg (HMGL gr, const char *fname, const char [C function]
                  *descr)
```

Exports current frame to SVG (Scalable Vector Graphics) file using vector representation. In difference of EPS format, SVG format support transparency that allows to correctly draw semitransparent plot (like [surfa], page 218, [surf3a], page 218, or [cloud], page 214). Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name). If file name is terminated by ‘z’ (for example, ‘*fname.svgz*’) then file will be compressed in gzip format. Note, that SVG format don’t support color interpolation, and the resulting plot will look as you use [quality], page 175=1 for plotting.

```
void WriteTEX (const char *fname, const char          [Method on mglGraph]
               *descr="")
void mgl_write_tex (HMGL gr, const char *fname, const char [C function]
                  *descr)
```

Exports current frame to LaTeX (package Tikz/PGF) file using vector representation. Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name). Note, there is no text scaling now (for example, in subplots), what may produce miss-aligned labels.

```
void WritePRC (const char *fname, const char          [Method on mglGraph]
               *descr="", bool make_pdf=true)
void mgl_write_prc (HMGL gr, const char *fname, const char [C function]
                  *descr, int make_pdf)
```

Exports current frame to PRC file using vector representation (see [http://en.wikipedia.org/wiki/PRC\\_%28file\\_format%29](http://en.wikipedia.org/wiki/PRC_%28file_format%29)). Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name). If parameter *make\_pdf=true* and PDF was enabled at MathGL configure then corresponding PDF file with 3D image will be created.

```
void WriteOBJ (const char *fname, const char          [Method on mglGraph]
               *descr="")
void mgl_write_obj (HMGL gr, const char *fname, const char [C function]
                  *descr)
```

Exports current frame to OBJ/MTL file using vector representation (see OBJ format ([http://en.wikipedia.org/wiki/Wavefront\\_.obj\\_file](http://en.wikipedia.org/wiki/Wavefront_.obj_file)) for details). Note, the output file may be too large for graphic of large data array (especially for surfaces).

It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name).

```
void WriteXYZ (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_xyz (HMGL gr, const char *fname, const char      [C function]
                   *descr)
```

Exports current frame to XYZ/XYZL/XYZF files using vector representation (see XYZ format (<http://people.sc.fsu.edu/~jburkardt/data/xyz/xyz.html>) for details). Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name).

```
void WriteSTL (const char *fname, const char          [Method on mglGraph]
               *descr="")
```

```
void mgl_write_stl (HMGL gr, const char *fname, const char      [C function]
                   *descr)
```

Exports current frame to STL file using vector representation (see STL format ([http://en.wikipedia.org/wiki/STL\\_\(file\\_format\)](http://en.wikipedia.org/wiki/STL_(file_format))) for details). Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name).

```
void WriteOFF (const char *fname, const char          [Method on mglGraph]
               *descr="", bool colored=false)
```

```
void mgl_write_off (HMGL gr, const char *fname, const char      [C function]
                   *descr, bool colored)
```

Exports current frame to OFF file using vector representation (see OFF format (<http://people.sc.fsu.edu/~jburkardt/data/off/off.html>) for details). Note, the output file may be too large for graphic of large data array (especially for surfaces). It is better to use bitmap format (for example PNG or JPEG). However, program has no internal limitations for size of output file. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name).

```
void ShowImage (const char *viewer, bool             [Method on mglGraph]
                nowait=false)
```

```
void mgl_show_image (const char *viewer, int nowait)      [C function]
```

Displays the current picture using external program *viewer* for viewing. The function save the picture to temporary file and call *viewer* to display it. If *nowait=true* then the function return immediately (it will not wait while window will be closed).

```
void WriteJSON (const char *fname, const char          [Method on mglGraph]
                *descr="")
```

```
void mgl_write_json (HMGL gr, const char *fname, const char [C function]
                    *descr)
```

Exports current frame to textual file using Section B.3 [JSON format], page 327. Later this file can be used for faster loading and viewing by JavaScript script. Parameter *fname* specifies the file name, *descr* adds description to file.

```
void ExportMGLD (const char *fname, const char          [Method on mglGraph]
                *descr="")
```

```
void mgl_export_mgld (HMGL gr, const char *fname, const char [C function]
                    *descr)
```

Exports points and primitives in file using Section B.2 [MGLD format], page 326. Later this file can be used for faster loading and viewing by `mglview` utility. Parameter *fname* specifies the file name, *descr* adds description to file (default is file name).

```
void ImportMGLD (const char *fname, bool add=false) [Method on mglGraph]
```

```
void mgl_import_mgld (HMGL gr, const char *fname, int add) [C function]
```

Imports points and primitives in file using Section B.2 [MGLD format], page 326. Later this file can be used for faster loading and viewing by `mglview` utility. Parameter *fname* specifies the file name, *add* sets to append or replace primitives to existed ones.

### 4.5.2 Frames/Animation

These functions provide ability to create several pictures simultaneously. For most of cases it is useless but for widget classes (see Chapter 5 [Widget classes], page 237) they can provide a way to show animation. Also you can write several frames into animated GIF file.

```
void NewFrame () [Method on mglGraph]
```

```
void mgl_new_frame (HMGL gr) [C function]
```

Creates new frame. Function returns current frame id. This is not thread safe function in OpenGL mode! Use direct list creation in multi-threading drawing. The function `EndFrame()` **must** be call after the finishing of the frame drawing for each call of this function.

```
void EndFrame () [Method on mglGraph]
```

```
void mgl_end_frame (HMGL gr) [C function]
```

Finishes the frame drawing.

```
int GetNumFrame () [Method on mglGraph]
```

```
int mgl_get_num_frame (HMGL gr) [C function]
```

Gets the number of created frames.

```
void SetFrame (int i) [Method on mglGraph]
```

```
void mgl_set_frame (HMGL gr, int i) [C function]
```

Finishes the frame drawing and sets drawing data to frame *i*, which should be in range `[0, GetNumFrame()-1]`. This function is similar to `EndFrame()` but don't add frame to the GIF image.



```

void GetFrame (int i) [Method on mglGraph]
void mgl_get_frame (HMGL gr, int i) [C function]
    Replaces drawing data by one from frame i. Function work if MGL_VECT_FRAME is set
    on (by default).

void ShowFrame (int i) [Method on mglGraph]
void mgl_show_frame (HMGL gr, int i) [C function]
    Appends drawing data from frame i to current one. Function work if MGL_VECT_FRAME
    is set on (by default).

void DelFrame (int i) [Method on mglGraph]
void mgl_del_frame (HMGL gr, int i) [C function]
    Deletes drawing data for frame i and shift all later frame indexes. Function work if
    MGL_VECT_FRAME is set on (by default). Do nothing in OpenGL mode.

void ResetFrames () [Method on mglGraph]
void mgl_reset_frames (HMGL gr) [C function]
    Reset frames counter (start it from zero).

void ClearFrame (int i) [Method on mglGraph]
void mgl_clear_frame (HMGL gr, int i) [C function]
    Clear list of primitives for current drawing.

void StartGIF (const char *fname, int ms=100) [Method on mglGraph]
void mgl_start_gif (HMGL gr, const char *fname, int ms) [C function]
    Start writing frames into animated GIF file fname. Parameter ms set the delay
    between frames in milliseconds. You should not change the picture size during writing
    the cinema. Use CloseGIF() to finalize writing. Note, that this function is disabled
    in OpenGL mode.

void CloseGIF () [Method on mglGraph]
void mgl_close_gif (HMGL gr) [C function]
    Finish writing animated GIF and close connected pointers.

```

### 4.5.3 Bitmap in memory

These functions return the created picture (bitmap), its width and height. You may display it by yourself in any graphical library (see also, Chapter 5 [Widget classes], page 237) or save in file (see also, Section 4.5.1 [Export to file], page 176).

```

const unsigned char * GetRGB () [Method on mglGraph]
void GetRGB (char *buf, int size) [Method on mglGraph]
void GetBGRN (char *buf, int size) [Method on mglGraph]
const unsigned char * mgl_get_rgb (HMGL gr) [C function]
    Gets RGB bitmap of the current state of the image. Format of each element of bits
    is: {red, green, blue}. Number of elements is Width*Height. Position of element {i,j}
    is [3*i + 3*Width*j] (or is [4*i + 4*Width*j] for GetBGRN()). You have to provide the
    proper size of the buffer, buf, i.e. the code for Python should look like

    from mathgl import *
    gr = mglGraph();

```

```

    bits='\t';
    bits=bits.expandtabs(4*gr.GetWidth()*gr.GetHeight());
    gr.GetBGRN(bits, len(bits));

const unsigned char * GetRGBA () [Method on mglGraph]
void GetRGBA (char *buf, int size) [Method on mglGraph]
const unsigned char * mgl_get_rgba (HMGL gr) [C function]
    Gets RGBA bitmap of the current state of the image. Format of each element of
    bits is: {red, green, blue, alpha}. Number of elements is Width*Height. Position of
    element {i,j} is [4*i + 4*Width*j].

int GetWidth () [Method on mglGraph]
int GetHeight () [Method on mglGraph]
int mgl_get_width (HMGL gr) [C function]
int mgl_get_height (HMGL gr) [C function]
    Gets width and height of the image.

mglPoint CalcXYZ (int xs, int ys) [Method on mglGraph]
void mgl_calc_xyz (HMGL gr, int xs, int ys, mreal *x, mreal *y, [C function]
    mreal *z)
    Calculate 3D coordinate {x,y,z} for screen point {xs,ys}. At this moment it ignore
    perspective and transformation formulas (curvilinear coordinates). The calculation
    are done for the last used InPlot (see Section 4.4 [Subplots and rotation], page 172).

mglPoint CalcScr (mglPoint p) [Method on mglGraph]
void mgl_calc_scr (HMGL gr, mreal x, mreal y, mreal z, int *xs, [C function]
    int *ys)
    Calculate screen point {xs,ys} for 3D coordinate {x,y,z}. The calculation are done
    for the last used InPlot (see Section 4.4 [Subplots and rotation], page 172).

void SetObjId (int id) [Method on mglGraph]
void mgl_set_obj_id (HMGL gr, int id) [C function]
    Set the numeric id for object or subplot/inplot.

int GetObjId (int xs, int ys) [Method on mglGraph]
int mgl_get_obj_id (HMGL gr, int xs, int ys) [C function]
    Get the numeric id for most upper object at pixel {xs, ys} of the picture.

int GetSplId (int xs, int ys) [Method on mglGraph]
int mgl_get_spl_id (HMGL gr, int xs, int ys) [C function]
    Get the numeric id for most subplot/inplot at pixel {xs, ys} of the picture.

void Highlight (int id) [Method on mglGraph]
void mgl_highlight (HMGL gr, int id) [C function]
    Highlight the object with given id.

long IsActive (int xs, int ys, int d=1) [Method on mglGraph]
long mgl_is_active (HMGL gr, int xs, int ys, int d) [C function]
    Checks if point {xs, ys} is close to one of active point (i.e. mglBase::Act) with
    accuracy d and return its index or -1 if not found. Active points are special points

```

which characterize primitives (like edges and so on). This function for advanced users only.

```
long SetDrawReg (int nx=1, int ny=1, int m=0) [Method on mglGraph]
long mgl_set_draw_reg (HMGL gr, int nx, int ny, int m) [C function]
```

Limits drawing region by rectangular area of  $m$ -th cell of matrix with sizes  $nx*ny$  (like in [subplot], page 172). This function can be used to update only small region of the image for purposes of higher speed. This function for advanced users only.

#### 4.5.4 Parallelization

Many of things MathGL do in parallel by default (if MathGL was built with pthread). However, there is function which set the number of threads to be used.

```
int mgl_set_num_thr (int n) [C function]
```

Set the number of threads to be used by MathGL. If  $n < 1$  then the number of threads is set as maximal number of processors (cores). If  $n = 1$  then single thread will be used (this is default if pthread was disabled).

Another option is combining bitmap image (taking into account Z-ordering) from different instances of `mglGraph`. This method is most appropriate for computer clusters when the data size is so large that it exceed the memory of single computer node.

```
int Combine (const mglGraph *g) [Method on mglGraph]
int mgl_combine_gr (HMGL gr, HMGL g) [C function]
```

Combine drawing from instance  $g$  with  $gr$  (or with this) taking into account Z-ordering of pixels. The width and height of both instances must be the same.

```
int MPI_Send (int id) [Method on mglGraph]
int mgl_mpi_send (HMGL gr, int id) [C function]
```

Send graphical information from node  $id$  using MPI. The width and height in both nodes must be the same.

```
int MPI_Recv (int id) [Method on mglGraph]
int mgl_mpi_send (HMGL gr, int id) [C function]
```

Receive graphical information from node  $id$  using MPI. The width and height in both nodes must be the same.

## 4.6 Background

These functions change background image.

```
clf ['col'] [MGL command]
void Clf () [Method on mglGraph]
void Clf (const char * col) [Method on mglGraph]
void Clf (char col) [Method on mglGraph]
void Clf (mreal r, mreal g, mreal b) [Method on mglGraph]
void mgl_clf (HMGL gr) [C function]
void mgl_clf_str (HMGL gr, const char * col) [C function]
void mgl_clf_chr (HMGL gr, char col) [C function]
void mgl_clf_rgb (HMGL gr, mreal r, mreal g, mreal b) [C function]
```

Clear the picture and fill background by specified color.

```

rasterize [MGL command]
void Rasterize () [Method on mglGraph]
void mgl_rasterize (HMGL gr) [C function]
    Force drawing the plot and use it as background. After it, function clear the list of
    primitives, like [clf], page 183. 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]
void LoadBackground (const char * fname, double [Method on mglGraph]
    alpha=1)
void mgl_load_background (HMGL gr, const char * fname, [C function]
    double alpha)
    Load PNG or JPEG file fname as background for the plot. Parameter alpha manually
    set transparency of the background.

```

## 4.7 Primitives

These functions draw some simple objects like line, point, sphere, drop, cone and so on. See Section 2.9.7 [Using primitives], page 119, for sample code and picture.

```

ball x y ['col']='r.' [MGL command]
ball x y z ['col']='r.' [MGL command]
void Ball (mglPoint p, char col='r') [Method on mglGraph]
void Mark (mglPoint p, const char *mark) [Method on mglGraph]
void mgl_mark (HMGL gr, mreal x, mreal y, mreal z, const char [C function]
    *mark)
    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]
void Error (mglPoint p, mglPoint e, char *stl="") [Method on mglGraph]
void mgl_error_box (HMGL gr, mreal x, mreal y, mreal z, mreal [C function]
    ex, mreal ey, mreal ez, char *stl)
    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]
void Line (mglPoint p1, mglPoint p2, char *stl="B", [Method on mglGraph]
    int num=2)
void mgl_line (HMGL gr, mreal x1, mreal y1, mreal z1, mreal x2, [C function]
    mreal y2, mreal z2, char *stl, int num)
    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 4.3.2 [Curved coordinates], page 167). 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]
void Curve (mglPoint p1, mglPoint d1, mglPoint p2, [Method on mglGraph]
            mglPoint d2, const char *stl="B", int num=100)
void mgl_curve (HMGL gr, mreal x1, mreal y1, mreal z1, mreal [C function]
                dx1, mreal dy1, mreal dz1, mreal x2, mreal y2, mreal z2, mreal dx2,
                mreal dy2, mreal dz2, const char *stl, int num)
```

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 4.3.2 [Curved coordinates], page 167). 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]
void Face (mglPoint p1, mglPoint p2, mglPoint p3, [Method on mglGraph]
            mglPoint p4, const char *stl="w")
void mgl_face (HMGL gr, mreal x1, mreal y1, mreal z1, mreal x2, [C function]
                mreal y2, mreal z2, mreal x3, mreal y3, mreal z3, mreal x4, mreal y4,
                mreal z4, const char *stl)
```

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]
void FaceX (mreal x0, mreal y0, mreal z0, mreal wy, [Method on mglGraph]
            mreal wz, const char *stl="w", mreal d1=0, mreal d2=0)
void FaceY (mreal x0, mreal y0, mreal z0, mreal wx, [Method on mglGraph]
            mreal wz, const char *stl="w", mreal d1=0, mreal d2=0)
void FaceZ (mreal x0, mreal y0, mreal z0, mreal wx, [Method on mglGraph]
            mreal wy, const char *stl="w", mreal d1=0, mreal d2=0)
void mgl_facex (HMGL gr, mreal x0, mreal y0, mreal z0, mreal [C function]
                wy, mreal wz, const char *stl, mreal d1, mreal d2)
void mgl_facey (HMGL gr, mreal x0, mreal y0, mreal z0, mreal [C function]
                wx, mreal wz, const char *stl, mreal d1, mreal d2)
```

```
void mgl_facez (HMGL gr, mreal x0, mreal y0, mreal z0, mreal          [C function]
                wx, mreal wy, const char *stl, mreal d1, mreal d2)
```

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]
```

```
void Sphere (mglPoint p, mreal r, const char [Method on mglGraph]
             *stl="r")
```

```
void mgl_sphere (HMGL gr, mreal x0, mreal y0, mreal z0, mreal r, [C function]
                 const char *stl)
```

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]
```

```
void Drop (mglPoint p, mglPoint d, mreal r, const [Method on mglGraph]
           char *col="r", mreal shift=1, mreal ap=1)
```

```
void mgl_drop (HMGL gr, mreal x0, mreal y0, mreal z0, mreal dx, [C function]
               mreal dy, mreal dz, mreal r, const char *col, mreal shift, mreal ap)
```

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]
```

```
void Cone (mglPoint p1, mglPoint p2, mreal r1, mreal [Method on mglGraph]
           r2=-1, const char *stl="B")
```

```
void mgl_cone (HMGL gr, mreal x1, mreal y1, mreal z1, mreal x2, [C function]
               mreal y2, mreal z2, mreal r1, mreal r2, const char *stl)
```

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]
```

```
void Circle (mglPoint p, mreal r, const char [Method on mglGraph]
             *stl="r")
```

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]

void Ellipse (mglPoint p1, mglPoint p2, mreal r, [Method on mglGraph]  
const char \*col="r")

void mgl\_ellipse (HMGL gr, mreal x1, mreal y1, mreal z1, mreal [C function]  
x2, mreal y2, mreal z2, mreal r, const char \*col)

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]

void Rhomb (mglPoint p1, mglPoint p2, mreal r, const [Method on mglGraph]  
char \*col="r")

void mgl\_rhomb (HMGL gr, mreal x1, mreal y1, mreal z1, mreal [C function]  
x2, mreal y2, mreal z2, mreal r, const char \*col)

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]

void Arc (mglPoint p0, mglPoint p1, mreal a, const [Method on mglGraph]  
char \*col="r")

void Arc (mglPoint p0, mglPoint pa, mglPoint p1, [Method on mglGraph]  
mreal a, const char \*col="r")

void mgl\_arc (HMGL gr, mreal x0, mreal y0, mreal x1, mreal y1, [C function]  
mreal a, const char \*col)

void mgl\_arc\_ext (HMGL gr, mreal x0, mreal y0, mreal z0, mreal [C function]  
xa, mreal ya, mreal za, mreal x1, mreal y1, mreal z1, mreal a, const  
char \*col)

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]

```
void Polygon (mglPoint p0, mglPoint p1, int num,          [Method on mglGraph]
              const char *col="r")
```

```
void mgl_polygon (HMGL gr, mreal x0, mreal y0, mreal z0, mreal    [C function]
                 x1, mreal y1, mreal z1, int num, const char *col)
```

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]
```

```
void Logo (const char *fname, bool smooth=false,          [Method on mglGraph]
           const char *opt="")
```

```
void Logo (long w, long h, const unsigned char *rgba,     [Method on mglGraph]
           bool smooth=false, const char *opt="")
```

```
void mgl_logo (HMGL gr, long w, long h, const unsigned    [C function only]
              char *rgba, bool smooth, const char *opt)
```

```
void mgl_logo_file (HMGL gr, const char *fname, bool smooth, [C function]
                   const char *opt)
```

Draw bitmap (logo) along whole axis range, which can be changed by Section 3.7 [Command options], page 153. Bitmap can be loaded from file or specified as RGBA values for pixels. Parameter *smooth* set to draw bitmap without or with color interpolation.

## 4.8 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 3.5 [Font styles], page 151). 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 4.2.6 [Font settings], page 161.

The font parameters are described by string. This string may set the text color '`wkrgbcymhRGCYMHw`' (see Section 3.2 [Color styles], page 147). Starting from MathGL v.2.3, you can set color gradient for text (see Section 3.4 [Color scheme], page 149). 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 2.2.7 [Text features], page 30, for sample code and picture.

```
text x y 'text' ['fnt']=" size=-1] [MGL command]
text x y z 'text' ['fnt']=" size=-1] [MGL command]
void Puts (mglPoint p, const char *text, const char [Method on mglGraph]
          *fnt=":C", mreal size=-1)
void Putsw (mglPoint p, const wchar_t *text, const [Method on mglGraph]
            char *fnt=":C", mreal size=-1)
void Puts (mreal x, mreal y, const char *text, const [Method on mglGraph]
            char *fnt=":AC", mreal size=-1)
void Putsw (mreal x, mreal y, const wchar_t *text, [Method on mglGraph]
            const char *fnt=":AC", mreal size=-1)
void mgl_puts (HMGL gr, mreal x, mreal y, mreal z, const char [C function]
               *text, const char *fnt, mreal size)
void mgl_putsw (HMGL gr, mreal x, mreal y, mreal z, const [C function]
                wchar_t *text, const char *fnt, mreal size)
```

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]
void Puts (mglPoint p, mglPoint d, const char *text, [Method on mglGraph]
            const char *fnt=":L", mreal size=-1)
void Putsw (mglPoint p, mglPoint d, const wchar_t [Method on mglGraph]
            *text, const char *fnt=":L", mreal size=-1)
void mgl_puts_dir (HMGL gr, mreal x, mreal y, mreal z, mreal [C function]
                  dx, mreal dy, mreal dz, const char *text, const char *fnt, mreal size)
void mgl_putsw_dir (HMGL gr, mreal x, mreal y, mreal z, mreal [C function]
                   dx, mreal dy, mreal dz, const wchar_t *text, const char *fnt, mreal
                   size)
```

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 161, command are used.

```
text ydat 'text' ['fnt']=" [MGL command]
text xdat ydat 'text' ['fnt']=" [MGL command]
text xdat ydat zdat 'text' ['fnt']=" [MGL command]
void Text (const mglDataA &y, const char *text, const [Method on mglGraph]
            char *fnt="", const char *opt="")
void Text (const mglDataA &y, const wchar_t *text, [Method on mglGraph]
            const char *fnt="", const char *opt="")
```

```

void Text (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const char *text, const char *fnt="", const char *opt="")
void Text (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const wchar_t *text, const char *fnt="", const char *opt="")
void Text (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const char *text, const char *fnt="", const char
           *opt="")
void Text (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const wchar_t *text, const char *fnt="", const
           char *opt="")
void mgl_text_y (HMGL gr, HCDT y, const char *text, const char      [C function]
                *fnt, const char *opt)
void mgl_textw_y (HMGL gr, HCDT y, const wchar_t *text, const      [C function]
                 char *fnt, const char *opt)
void mgl_text_xy (HCDT x, HCDT y, const char *text, const char      [C function]
                 *fnt, const char *opt)
void mgl_textw_xy (HCDT x, HCDT y, const wchar_t *text, const      [C function]
                  char *fnt, const char *opt)
void mgl_text_xyz (HCDT x, HCDT y, HCDT z, const char *text,      [C function]
                  const char *fnt, const char *opt)
void mgl_textw_xyz (HCDT x, HCDT y, HCDT z, const wchar_t         [C function]
                   *text, const char *fnt, const char *opt)

```

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 above 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 4.3.1 [Ranges (bounding box)], page 166). If array *z* is not specified then  $z[i]$  equal to minimal z-axis value is used. String *opt* contain command options (see Section 3.7 [Command options], page 153).

## 4.9 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 4.3 [Axis settings], page 166.

```

axis ['dir'='xyz' 'stl'=""]                                     [MGL command]
void Axis (const char *dir="xyz", const char                    [Method on mglGraph]
           *stl="", const char *opt="")
void mgl_axis (HMGL gr, const char *dir, const char *stl, const  [C function]
              char *opt)

```

Draws axes with ticks (see Section 4.3 [Axis settings], page 166). 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 171);
- ‘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 overridden by using *stl* string. Option **value** set the manual rotation angle for the ticks. See Section 2.2.2 [Axis and ticks], page 21, for sample code and picture.

```
colorbar ['sch']="] [MGL command]
void Colorbar (const char *sch="") [Method on mglGraph]
void mgl_colorbar (HMGL gr, const char *sch) [C function]
```

Draws colorbar. Parameter *sch* may contain:

- color scheme (see Section 3.4 [Color scheme], page 149);
- ‘<>^\_’ 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 171);
- ‘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 2.2.4 [Colorbars], page 27, for sample code and picture.

```
colorbar vdat ['sch']="] [MGL command]
void Colorbar (const mglDataA &v, const char [Method on mglGraph]
               *sch="")
void mgl_colorbar_val (HMGL gr, HCDT v, const char *sch) [C function]
```

The same as previous but with sharp colors *sch* (current palette if *sch*="") for values *v*. See Section 2.6.14 [ContD sample], page 83, for sample code and picture.

```
colorbar 'sch' x y [w=1 h=1] [MGL command]
void Colorbar (const char *sch, mreal x, mreal y, [Method on mglGraph]
               mreal w=1, mreal h=1)
```

```
void mgl_colorbar_ext (HMGL gr, const char *sch, mreal x, [C function]
                      mreal y, mreal w, mreal h)
```

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]
void Colorbar (const mglDataA &v, const char *sch, [Method on mglGraph]
               mreal x, mreal y, mreal w=1, mreal h=1)
```

```
void mgl_colorbar_val_ext (HMGL gr, HCDT v, const char *sch, [C function]
                          mreal x, mreal y, mreal w, mreal h)
```

The same as previous but with sharp colors  $sch$  (current palette if  $sch=""$ ) for values  $v$ . See Section 2.6.14 [ContD sample], page 83, for sample code and picture.

```
grid ['dir'='xyz' 'pen'='B'] [MGL command]
void Grid (const char *dir="xyz", const char [Method on mglGraph]
           *pen="B", const char *opt="")
```

```
void mgl_axis_grid (HMGL gr, const char *dir, const char *pen, [C function]
                   const char *opt)
```

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 190. The style of lines is determined by  $pen$  parameter (default value is dark blue solid line  $'B-'$ ).

```
box ['stl'='k' ticks=on] [MGL command]
void Box (const char *col="", bool ticks=true) [Method on mglGraph]
```

```
void mgl_box (HMGL gr, int ticks) [C function]
```

```
void mgl_box_str (HMGL gr, const char *col, int ticks) [C function]
```

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 2.2.5 [Bounding box], page 28, 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]
```

```
void Label (char dir, const char *text, mreal pos=1, [Method on mglGraph]
            const char *opt="")
```

```
void Label (char dir, const wchar_t *text, mreal [Method on mglGraph]
            pos=1, const char *opt="")
```

```
void mgl_label (HMGL gr, char dir, const char *text, mreal pos, [C function]
               const char *opt)
```

```
void mgl_labelw (HMGL gr, char dir, const wchar_t *text, mreal [C function]
                pos, const char *opt)
```

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 4.8 [Text printing], page 188.

## 4.10 Legend

These functions draw legend to the graph (useful for Section 4.11 [1D plotting], page 194). 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 193) 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 4.2.6 [Font settings], page 161). 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 2.2.8 [Legend sample], page 33, for sample code and picture.

```
legend [pos=3 'fnt'='#'] [MGL command]
void Legend (int pos=0x3, const char *fnt="#", const char *opt="") [Method on mglGraph]
void mgl_legend (HMGL gr, int pos, const char *fnt, const char *opt) [C function]
```

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]
void Legend (mreal x, mreal y, const char *fnt="#", const char *opt="") [Method on mglGraph]
void mgl_legend_pos (HMGL gr, mreal x, mreal y, const char *fnt, const char *opt) [C function]
```

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]
void AddLegend (const char *text, const char *style) [Method on mglGraph]
```

```

void AddLegend (const wchar_t *text, const char          [Method on mglGraph]
                 *style)
void mgl_add_legend (HMGL gr, const char *text, const char [C function]
                 *style)
void mgl_add_legendw (HMGL gr, const wchar_t *text, const [C function]
                 char *style)
    Adds string text to internal legend accumulator. The style of described line and mark
    is specified in string style (see Section 3.3 [Line styles], page 147).

clearlegend                                             [MGL command]
void ClearLegend ()                                    [Method on mglGraph]
void mgl_clear_legend (HMGL gr)                        [C function]
    Clears saved legend strings.

legendmarks val                                       [MGL command]
void SetLegendMarks (int num)                          [Method on mglGraph]
void mgl_set_legend_marks (HMGL gr, int num)           [C function]
    Set the number of marks in the legend. By default 1 mark is used.

```

## 4.11 1D plotting

These functions perform plotting of 1D data. 1D means that data depended from only 1 parameter like parametric curve  $\{x[i], y[i], z[i]\}$ ,  $i=1\dots n$ . 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 3.3 [Line styles], page 147). By default (*pen=""*) solid line with color from palette is used (see Section 4.2.7 [Palette and colors], page 162). Symbol *!* set to use new color from palette for each point (not for each curve, as default). String *opt* contain command options (see Section 3.7 [Command options], page 153). See Section 2.5 [1D samples], page 44, for sample code and picture.

```

plot ydat ['stl']="]                                   [MGL command]
plot xdat ydat ['stl']="]                             [MGL command]
plot xdat ydat zdat ['stl']="]                       [MGL command]
void Plot (const mglDataA &y, const char *pen="",      [Method on mglGraph]
          const char *opt="")
void Plot (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
          const char *pen="", const char *opt="")
void Plot (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
          const mglDataA &z, const char *pen="", const char *opt="")
void mgl_plot (HMGL gr, HCDT y, const char *pen, const char [C function]
              *opt)
void mgl_plot_xy (HMGL gr, HCDT x, HCDT y, const char *pen, [C function]
                 const char *opt)

```

```
void mgl_plot_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *pen, const char *opt)
```

These functions draw continuous lines between points  $\{x[i], y[i], z[i]\}$ . See also [area], page 196, [step], page 195, [stem], page 197, [tube], page 205, [mark], page 201, [error], page 201, [belt], page 208, [tens], page 195, [tape], page 196. See Section 2.5.1 [Plot sample], page 45, for sample code and picture.

```
radar adat ['stl']="] [MGL command]
```

```
void Radar (const mglDataA &a, const char *pen="", [Method on mglGraph]
            const char *opt="")
```

```
void mgl_radar (HMGL gr, HCDT a, const char *pen, const char [C function]
               *opt)
```

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 194. See Section 2.5.2 [Radar sample], page 46, for sample code and picture.

```
step ydat ['stl']="] [MGL command]
```

```
step xdat ydat ['stl']="] [MGL command]
```

```
step xdat ydat zdat ['stl']="] [MGL command]
```

```
void Step (const mglDataA &y, const char *pen="", [Method on mglGraph]
           const char *opt="")
```

```
void Step (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const char *pen="", const char *opt="")
```

```
void Step (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *pen="", const char *opt="")
```

```
void mgl_step (HMGL gr, HCDT y, const char *pen, const char [C function]
               *opt)
```

```
void mgl_step_xy (HMGL gr, HCDT x, HCDT y, const char *pen, [C function]
                  const char *opt)
```

```
void mgl_step_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                   *pen, const char *opt)
```

These functions draw continuous stairs for points to axis plane. See also [plot], page 194, [stem], page 197, [tile], page 209, [boxs], page 208. See Section 2.5.3 [Step sample], page 47, 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]
```

```
void Tens (const mglDataA &y, const mglDataA &c, [Method on mglGraph]
           const char *pen="", const char *opt="")
```

```
void Tens (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &c, const char *pen="", const char *opt="")
```

```
void Tens (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const mglDataA &c, const char *pen="", const
           char *opt="")
```

```

void mgl_tens (HMGL gr, HCDT y, HCDT c, const char *pen, const char *opt) [C function]
void mgl_tens_xy (HMGL gr, HCDT x, HCDT y, HCDT c, const char *pen, const char *opt) [C function]
void mgl_tens_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c, const char *pen, const char *opt) [C function]

```

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 3.4 [Color scheme], page 149) and style and/or width of line (see Section 3.3 [Line styles], page 147). See also [plot], page 194, [mesh], page 207, [fall], page 208. See Section 2.5.4 [Tens sample], page 48, for sample code and picture.

```

tape ydat ['stl']=" [MGL command]
tape xdat ydat ['stl']=" [MGL command]
tape xdat ydat zdat ['stl']=" [MGL command]
void Tape (const mglDataA &y, const char *pen="", const char *opt="") [Method on mglGraph]
void Tape (const mglDataA &x, const mglDataA &y, const char *pen="", const char *opt="") [Method on mglGraph]
void Tape (const mglDataA &x, const mglDataA &y, const mglDataA &z, const char *pen="", const char *opt="") [Method on mglGraph]
void mgl_tape (HMGL gr, HCDT y, const char *pen, const char *opt) [C function]
void mgl_tape_xy (HMGL gr, HCDT x, HCDT y, const char *pen, const char *opt) [C function]
void mgl_tape_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char *pen, const char *opt) [C function]

```

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 160, and can be changed by option value. See also [plot], page 194, [flow], page 224, [barwidth], page 160. See Section 2.5.21 [Tape sample], page 65, for sample code and picture.

```

area ydat ['stl']=" [MGL command]
area xdat ydat ['stl']=" [MGL command]
area xdat ydat zdat ['stl']=" [MGL command]
void Area (const mglDataA &y, const char *pen="", const char *opt="") [Method on mglGraph]
void Area (const mglDataA &x, const mglDataA &y, const char *pen="", const char *opt="") [Method on mglGraph]
void Area (const mglDataA &x, const mglDataA &y, const mglDataA &z, const char *pen="", const char *opt="") [Method on mglGraph]
void mgl_area (HMGL gr, HCDT y, const char *pen, const char *opt) [C function]
void mgl_area_xy (HMGL gr, HCDT x, HCDT y, const char *pen, const char *opt) [C function]

```



```
void mgl_area_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *pen, const char *opt)
```

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 194, [bars], page 198, [stem], page 197, [region], page 197. See Section 2.5.5 [Area sample], page 49, 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]
```

```
void Region (const mglDataA &y1, const mglDataA &y2, [Method on mglGraph]
             const char *pen="", const char *opt="")
```

```
void Region (const mglDataA &x, const mglDataA &y1, [Method on mglGraph]
             const mglDataA &y2, const char *pen="", const char *opt="")
```

```
void Region (const mglDataA &x1, const mglDataA &y1, [Method on mglGraph]
             const mglDataA &x2, const mglDataA &y2, const char *pen="", const
             char *opt="")
```

```
void Region (const mglDataA &x1, const mglDataA &y1, [Method on mglGraph]
             const mglDataA &z1, const mglDataA &x2, const mglDataA &y2, const
             mglDataA &z2, const char *pen="", const char *opt="")
```

```
void mgl_region (HMGL gr, HCDT y1, HCDT y2, const char *pen, [C function]
                const char *opt)
```

```
void mgl_region_xy (HMGL gr, HCDT x, HCDT y1, HCDT y2, const [C function]
                   char *pen, const char *opt)
```

```
void mgl_region_3d (HMGL gr, HCDT x1, HCDT y1, HCDT z1, HCDT [C function]
                   x2, HCDT y2, HCDT z2, const char *pen, const char *opt)
```

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 196, [bars], page 198, [stem], page 197. See Section 2.5.6 [Region sample], page 50, for sample code and picture.

```
stem ydat ['stl']="] [MGL command]
```

```
stem xdat ydat ['stl']="] [MGL command]
```

```
stem xdat ydat zdat ['stl']="] [MGL command]
```

```
void Stem (const mglDataA &y, const char *pen="", [Method on mglGraph]
           const char *opt="")
```

```
void Stem (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const char *pen="", const char *opt="")
```

```
void Stem (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *pen="", const char *opt="")
```

```
void mgl_stem (HMGL gr, HCDT y, const char *pen, const char [C function]
              *opt)
```

```
void mgl_stem_xy (HMGL gr, HCDT x, HCDT y, const char *pen, [C function]
                 const char *opt)
```

```
void mgl_stem_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *pen, const char *opt)
```

These functions draw vertical lines from points to axis plane. See also [area], page 196, [bars], page 198, [plot], page 194, [mark], page 201. See Section 2.5.7 [Stem sample], page 51, for sample code and picture.

```
bars ydat ['stl']="] [MGL command]
```

```
bars xdat ydat ['stl']="] [MGL command]
```

```
bars xdat ydat zdat ['stl']="] [MGL command]
```

```
void Bars (const mglDataA &y, const char *pen="", [Method on mglGraph]
           const char *opt="")
```

```
void Bars (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const char *pen="", const char *opt="")
```

```
void Bars (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *pen="", const char *opt="")
```

```
void mgl_bars (HMGL gr, HCDT y, const char *pen, const char [C function]
              *opt)
```

```
void mgl_bars_xy (HMGL gr, HCDT x, HCDT y, const char *pen, [C function]
                 const char *opt)
```

```
void mgl_bars_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *pen, const char *opt)
```

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 198, [cones], page 199, [area], page 196, [stem], page 197, [chart], page 199, [barwidth], page 160. See Section 2.5.8 [Bars sample], page 52, for sample code and picture.

```
barh vdat ['stl']="] [MGL command]
```

```
barh ydat vdat ['stl']="] [MGL command]
```

```
void Barh (const mglDataA &v, const char *pen="", [Method on mglGraph]
           const char *opt="")
```

```
void Barh (const mglDataA &y, const mglDataA &v, [Method on mglGraph]
           const char *pen="", const char *opt="")
```

```
void mgl_barh (HMGL gr, HCDT v, const char *pen, const char [C function]
              *opt)
```

```
void mgl_barh_xy (HMGL gr, HCDT y, HCDT v, const char *pen, [C function]
                 const char *opt)
```

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 198, [barwidth], page 160. See Section 2.5.9 [Barh sample], page 53, for sample code and picture.

```

cones ydat ['stl']="] [MGL command]
cones xdat ydat ['stl']="] [MGL command]
cones xdat ydat zdat ['stl']="] [MGL command]
void Cones (const mglDataA &y, const char *pen="", [Method on mglGraph]
            const char *opt="")
void Cones (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const char *pen="", const char *opt="")
void Cones (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const char *pen="", const char *opt="")
void mgl_cones (HMGL gr, HCDT y, const char *pen, const char [C function]
               *opt)
void mgl_cones_xy (HMGL gr, HCDT x, HCDT y, const char *pen, [C function]
                  const char *opt)
void mgl_cones_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                   *pen, const char *opt)

```

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 198, [cone], page 186, [barwidth], page 160. See Section 2.5.10 [Cones sample], page 54, for sample code and picture.

```

chart adat ['col']="] [MGL command]
void Chart (const mglDataA &a, const char *col="", [Method on mglGraph]
            const char *opt="")
void mgl_chart (HMGL gr, HCDT a, const char *col, const char [C function]
               *opt)

```

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 4.2.7 [Palette and colors], page 162). 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 2.5.11 [Chart sample], page 55, for sample code and picture.

```

boxplot adat ['stl']="] [MGL command]
boxplot xdat adat ['stl']="] [MGL command]

```

```

void BoxPlot (const mglDataA &a, const char [Method on mglGraph]
               *pen="", const char *opt="")
void BoxPlot (const mglDataA &x, const mglDataA &a, [Method on mglGraph]
               const char *pen="", const char *opt="")
void mgl_boxplot (HMGL gr, HCDT a, const char *pen, const char [C function]
                  *opt)
void mgl_boxplot_xy (HMGL gr, HCDT x, HCDT a, const char *pen, [C function]
                     const char *opt)

```

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 194, [error], page 201, [bars], page 198, [barwidth], page 160. See Section 2.5.12 [BoxPlot sample], page 56, 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]
void Candle (const mglDataA &v1, const char [Method on mglGraph]
              *pen="", const char *opt="")
void Candle (const mglDataA &v1, const mglDataA &v2, [Method on mglGraph]
              const char *pen="", const char *opt="")
void Candle (const mglDataA &v1, const mglDataA &y1, [Method on mglGraph]
              const mglDataA &y2, const char *pen="", const char *opt="")
void Candle (const mglDataA &v1, const mglDataA &v2, [Method on mglGraph]
              const mglDataA &y1, const mglDataA &y2, const char *pen="", const
              char *opt="")
void Candle (const mglDataA &x, const mglDataA &v1, [Method on mglGraph]
              const mglDataA &v2, const mglDataA &y1, const mglDataA &y2, const
              char *pen="", const char *opt="")
void mgl_candle (HMGL gr, HCDT v1, HCDT y1, HCDT y2, const char [C function]
                  *pen, const char *opt)
void mgl_candle_yv (HMGL gr, HCDT v1, HCDT v2, HCDT y1, HCDT [C function]
                    y2, const char *pen, const char *opt)
void mgl_candle_xyv (HMGL gr, HCDT x, HCDT v1, HCDT v2, HCDT [C function]
                     y1, HCDT y2, const char *pen, const char *opt)

```

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 194, [bars], page 198, [ohlc], page 201, [barwidth], page 160. See Section 2.5.13 [Candle sample], page 57, for sample code and picture.

```

ohlc odat hdat ldat cdat ['stl']="] [MGL command]
ohlc xdat odat hdat ldat cdat ['stl']="] [MGL command]
void OHLC (const mglDataA &o, const mglDataA &h, [Method on mglGraph]
           const mglDataA &l, const mglDataA &c, const char *pen="", const
           char *opt="")
void OHLC (const mglDataA &x, const mglDataA &o, [Method on mglGraph]
           const mglDataA &h, const mglDataA &l, const mglDataA &c, const
           char *pen="", const char *opt="")
void mgl_ohlc (HMGL gr, HCDT o, HCDT h, HCDT l, HCDT c, const [C function]
               char *pen, const char *opt)
void mgl_ohlc_x (HMGL gr, HCDT x, HCDT o, HCDT h, HCDT l, HCDT c, [C function]
                 const char *pen, const char *opt)

```

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 200, [plot], page 194, [barwidth], page 160. See Section 2.5.14 [OHLC sample], page 57, 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]
void Error (const mglDataA &y, const mglDataA &ey, [Method on mglGraph]
            const char *pen="", const char *opt="")
void Error (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &ey, const char *pen="", const char *opt="")
void Error (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &ex, const mglDataA &ey, const char *pen="", const
            char *opt="")
void mgl_error (HMGL gr, HCDT y, HCDT ey, const char *pen, [C function]
                const char *opt)
void mgl_error_xy (HMGL gr, HCDT x, HCDT y, HCDT ey, const char [C function]
                  *pen, const char *opt)
void mgl_error_exy (HMGL gr, HCDT x, HCDT y, HCDT ex, HCDT ey, [C function]
                   const char *pen, const char *opt)

```

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 194, [mark], page 201. See Section 2.5.15 [Error sample], page 58, 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]
void Mark (const mglDataA &y, const mglDataA &r, [Method on mglGraph]
           const char *pen="", const char *opt="")

```

```

void Mark (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &r, const char *pen="", const char *opt="")
void Mark (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const mglDataA &r, const char *pen="", const
           char *opt="")
void mgl_mark_y (HMGL gr, HCDT y, HCDT r, const char *pen,      [C function]
                const char *opt)
void mgl_mark_xy (HMGL gr, HCDT x, HCDT y, HCDT r, const char   [C function]
                 *pen, const char *opt)
void mgl_mark_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT r,    [C function]
                  const char *pen, const char *opt)

```

These functions draw marks with size  $r[i]*[\text{marksize}]$ , page 160, at points  $\{x[i], y[i], z[i]\}$ . If you need to draw markers of the same size then you can use `[plot]`, page 194, function with empty line style `' '`. For markers with size in axis range use `[error]`, page 201, with style `'@'`. See also `[plot]`, page 194, `[textmark]`, page 202, `[error]`, page 201, `[stem]`, page 197. See Section 2.5.16 `[Mark sample]`, page 60, 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]
void TextMark (const mglDataA &y, const char *txt,          [Method on mglGraph]
               const char *fnt="", const char *opt="")
void TextMark (const mglDataA &y, const wchar_t           [Method on mglGraph]
               *txt, const char *fnt="", const char *opt="")
void TextMark (const mglDataA &y, const mglDataA &r,        [Method on mglGraph]
               const char *txt, const char *fnt="", const char *opt="")
void TextMark (const mglDataA &y, const mglDataA &r,        [Method on mglGraph]
               const wchar_t *txt, const char *fnt="", const char *opt="")
void TextMark (const mglDataA &x, const mglDataA &y,        [Method on mglGraph]
               const mglDataA &r, const char *txt, const char *fnt="", const char
               *opt="")
void TextMark (const mglDataA &x, const mglDataA &y,        [Method on mglGraph]
               const mglDataA &r, const wchar_t *txt, const char *fnt="", const
               char *opt="")
void TextMark (const mglDataA &x, const mglDataA &y,        [Method on mglGraph]
               const mglDataA &z, const mglDataA &r, const char *txt, const char
               *fnt="", const char *opt="")
void TextMark (const mglDataA &x, const mglDataA &y,        [Method on mglGraph]
               const mglDataA &z, const mglDataA &r, const wchar_t *txt, const
               char *fnt="", const char *opt="")
void mgl_textmark (HMGL gr, HCDT y, const char *txt, const   [C function]
                  char *fnt, const char *opt)
void mgl_textmarkw (HMGL gr, HCDT y, const wchar_t *txt, const [C function]
                   char *fnt, const char *opt)

```

```

void mgl_textmark_yr (HMGL gr, HCDT y, HCDT r, const char *txt,      [C function]
                     const char *fnt, const char *opt)
void mgl_textmarkw_yr (HMGL gr, HCDT y, HCDT r, const wchar_t      [C function]
                     *txt, const char *fnt, const char *opt)
void mgl_textmark_xyr (HMGL gr, HCDT x, HCDT y, HCDT r, const      [C function]
                     char *txt, const char *fnt, const char *opt)
void mgl_textmarkw_xyr (HMGL gr, HCDT x, HCDT y, HCDT r, const      [C function]
                     wchar_t *txt, const char *fnt, const char *opt)
void mgl_textmark_xyzr (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT r,   [C function]
                     const char *txt, const char *fnt, const char *opt)
void mgl_textmarkw_xyzr (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT     [C function]
                     r, const wchar_t *txt, const char *fnt, const char *opt)

```

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 194, [mark], page 201, [stem], page 197. See Section 2.5.17 [TextMark sample], page 61, 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]
void Label (const mglDataA &y, const char *txt, const             [Method on mglGraph]
            char *fnt="", const char *opt="")
void Label (const mglDataA &y, const wchar_t *txt,              [Method on mglGraph]
            const char *fnt="", const char *opt="")
void Label (const mglDataA &x, const mglDataA &y,               [Method on mglGraph]
            const char *txt, const char *fnt="", const char *opt="")
void Label (const mglDataA &x, const mglDataA &y,               [Method on mglGraph]
            const wchar_t *txt, const char *fnt="", const char *opt="")
void Label (const mglDataA &x, const mglDataA &y,               [Method on mglGraph]
            const mglDataA &z, const char *txt, const char *fnt="", const char
            *opt="")
void Label (const mglDataA &x, const mglDataA &y,               [Method on mglGraph]
            const mglDataA &z, const wchar_t *txt, const char *fnt="", const
            char *opt="")
void mgl_label (HMGL gr, HCDT y, const char *txt, const char      [C function]
               *fnt, const char *opt)
void mgl_labelw (HMGL gr, HCDT y, const wchar_t *txt, const      [C function]
                char *fnt, const char *opt)
void mgl_label_xy (HMGL gr, HCDT x, HCDT y, const char *txt,     [C function]
                  const char *fnt, const char *opt)
void mgl_labelw_xy (HMGL gr, HCDT x, HCDT y, const wchar_t      [C function]
                   *txt, const char *fnt, const char *opt)
void mgl_label_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char  [C function]
                   *txt, const char *fnt, const char *opt)

```

```
void mgl_labelw_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const      [C function]
                    wchar_t *txt, const char *fnt, const char *opt)
```

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 3.5 [Font styles], page 151;
- ‘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 194, [mark], page 201, [textmark], page 202, [table], page 204.  
See Section 2.5.18 [Label sample], page 62, for sample code and picture.

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

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

```
void Table (const mglDataA &val, const char *txt,               [Method on mglGraph]
            const char *fnt="", const char *opt="")
```

```
void Table (const mglDataA &val, const wchar_t *txt,           [Method on mglGraph]
            const char *fnt="", const char *opt="")
```

```
void Table (mreal x, mreal y, const mglDataA &val,             [Method on mglGraph]
            const char *txt, const char *fnt="", const char *opt="")
```

```
void Table (mreal x, mreal y, const mglDataA &val,             [Method on mglGraph]
            const wchar_t *txt, const char *fnt="", const char *opt="")
```

```
void mgl_table (HMGL gr, mreal x, mreal y, HCDT val, const char [C function]
                *txt, const char *fnt, const char *opt)
```

```
void mgl_tablew (HMGL gr, mreal x, mreal y, HCDT val, const    [C function]
                 wchar_t *txt, const char *fnt, const char *opt)
```

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 3.5 [Font styles], page 151;
- ‘#’ for drawing cell borders;
- ‘|’ for limiting table width 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 194, `[label]`, page 203. See Section 2.5.19 `[Table sample]`, page 63, 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]
void Tube (const mglDataA &y, const mglDataA &r, [Method on mglGraph]
           const char *pen="", const char *opt="")
void Tube (const mglDataA &y, mreal r, const char [Method on mglGraph]
           *pen="", const char *opt="")
void Tube (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &r, const char *pen="", const char *opt="")
void Tube (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           mreal r, const char *pen="", const char *opt="")
void Tube (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const mglDataA &r, const char *pen="", const
           char *opt="")
void Tube (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, mreal r, const char *pen="", const char
           *opt="")
void mgl_tube_r (HMGL gr, HCDT y, HCDT r, const char *pen, [C function]
                const char *opt)
void mgl_tube (HMGL gr, HCDT y, mreal r, const char *pen, const [C function]
               char *opt)
void mgl_tube_xyr (HMGL gr, HCDT x, HCDT y, HCDT r, const char [C function]
                  *pen, const char *opt)
void mgl_tube_xy (HMGL gr, HCDT x, HCDT y, mreal r, const char [C function]
                  *pen, const char *opt)
void mgl_tube_xyzr (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT r, [C function]
                   const char *pen, const char *opt)
void mgl_tube_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, mreal r, [C function]
                   const char *pen, const char *opt)

```

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 194. See Section 2.5.20 `[Tube sample]`, page 64, for sample code and picture.

```

torus rdat zdat ['stl']="] [MGL command]
void Torus (const mglDataA &r, const mglDataA &z, [Method on mglGraph]
            const char *pen="", const char *opt="")
void mgl_torus (HMGL gr, HCDT r, HCDT z, const char *pen, const [C function]
                char *opt)

```

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 194, [axial], page 212. See Section 2.5.22 [Torus sample], page 66, for sample code and picture.

```
lamerey x0 ydat ['stl']="] [MGL command]
lamerey x0 'y(x)' ['stl']="] [MGL command]
void Lamerey (double x0, const mglDataA &y, const [Method on mglGraph]
    char *stl="", const char *opt="")
void Lamerey (double x0, const char *y, const char [Method on mglGraph]
    *stl="", const char *opt="")
void mgl_lamerey_dat (HMGL gr, double x0, HCDT y, const char [C function]
    *stl, const char *opt)
void mgl_lamerey_str (HMGL gr, double x0, const char *y, const [C function]
    char *stl, const char *opt)
```

These functions draw Lamerey diagram for mapping  $x_{\text{new}} = y(x_{\text{old}})$  starting from point  $x_0$ . 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 194, [fplot], page 229, [bifurcation], page 206, [pmap], page 206. See Section 2.5.23 [Lamerey sample], page 67, for sample code and picture.

```
bifurcation dx ydat ['stl']="] [MGL command]
bifurcation dx 'y(x)' ['stl']="] [MGL command]
void Bifurcation (double dx, const mglDataA &y, [Method on mglGraph]
    const char *stl="", const char *opt="")
void Bifurcation (double dx, const char *y, const [Method on mglGraph]
    char *stl="", const char *opt="")
void mgl_bifurcation_dat (HMGL gr, double dx, HCDT y, const [C function]
    char *stl, const char *opt)
void mgl_bifurcation_str (HMGL gr, double dx, const char *y, [C function]
    const char *stl, const char *opt)
```

These functions draw bifurcation diagram for mapping  $x_{\text{new}} = y(x_{\text{old}})$ . 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 194, [fplot], page 229, [lamerey], page 206. See Section 2.5.24 [Bifurcation sample], page 68, 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]
void Pmap (const mglDataA &y, const mglDataA &s, [Method on mglGraph]
    const char *stl="", const char *opt="")
void Pmap (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
    const mglDataA &s, const char *stl="", const char *opt="")
void Pmap (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
    const mglDataA &z, const mglDataA &s, const char *stl="", const
    char *opt="")
void mgl_pmap (HMGL gr, HMDT y, HCDT s, const char *stl, const [C function]
    char *opt)
```

```
void mgl_pmap_xy (HMGL gr, HCDT x, HMDT y, HCDT s, const char [C function]
                 *stl, const char *opt)
```

```
void mgl_pmap_xyz (HMGL gr, HCDT x, HMDT y, HCDT z, HCDT s, [C function]
                  const char *stl, const char *opt)
```

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 194, [mark], page 201, [lamerey], page 206. See Section 2.5.25 [Pmap sample], page 69, for sample code and picture.

## 4.12 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 3.4 [Color scheme], page 149) for plot. String *opt* contain command options (see Section 3.7 [Command options], page 153). See Section 2.6 [2D samples], page 70, for sample code and picture.

```
surf zdat ['sch']="] [MGL command]
```

```
surf xdat ydat zdat ['sch']="] [MGL command]
```

```
void Surf (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Surf (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_surf (HMGL gr, HCDT z, const char *sch, const char [C function]
              *opt)
```

```
void mgl_surf_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                 *sch, const char *opt)
```

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 207, [dens], page 209, [belt], page 208, [tile], page 209, [boxs], page 208, [surfc], page 217, [surfa], page 218. See Section 2.6.1 [Surf sample], page 71, for sample code and picture.

```
mesh zdat ['sch']="] [MGL command]
```

```
mesh xdat ydat zdat ['sch']="] [MGL command]
```

```
void Mesh (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Mesh (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_mesh (HMGL gr, HCDT z, const char *sch, const char [C function]
              *opt)
```

```
void mgl_mesh_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

The function draws mesh lines for surface specified parametrically  $\{x[i,j], y[i,j], z[i,j]\}$ . See also [surf], page 207, [fall], page 208, [meshnum], page 160, [cont], page 209, [tens], page 195. See Section 2.6.5 [Mesh sample], page 75, for sample code and picture.

```
fall zdat ['sch']="] [MGL command]
```

```
fall xdat ydat zdat ['sch']="] [MGL command]
```

```
void Fall (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Fall (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_fall (HMGL gr, HCDT z, const char *sch, const char [C function]
              *opt)
```

```
void mgl_fall_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 208, [mesh], page 207, [tens], page 195, [meshnum], page 160. See Section 2.6.6 [Fall sample], page 76, for sample code and picture.

```
belt zdat ['sch']="] [MGL command]
```

```
belt xdat ydat zdat ['sch']="] [MGL command]
```

```
void Belt (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Belt (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_belt (HMGL gr, HCDT z, const char *sch, const char [C function]
              *opt)
```

```
void mgl_belt_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 194). If *sch* contain 'x' then belts are drawn along x-direction else (by default) belts are drawn along y-direction. See also [fall], page 208, [surf], page 207, [plot], page 194, [meshnum], page 160. See Section 2.6.7 [Belt sample], page 77, for sample code and picture.

```
boxs zdat ['sch']="] [MGL command]
```

```
boxs xdat ydat zdat ['sch']="] [MGL command]
```

```
void Boxs (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Boxs (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_boxs (HMGL gr, HCDT z, const char *sch, const char [C function]
              *opt)
```

```
void mgl_boxs_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 207, [dens], page 209, [tile], page 209, [step], page 195. See Section 2.6.8 [Boxs sample], page 78, for sample code and picture.

```
tile zdat ['sch']="] [MGL command]
```

```
tile xdat ydat zdat ['sch']="] [MGL command]
```

```
void Tile (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Tile (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_tile (HMGL gr, HCDT z, const char *sch, const char [C function]
               *opt)
```

```
void mgl_tile_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 195. See also [surf], page 207, [boxs], page 208, [step], page 195, [tiles], page 220. See Section 2.6.9 [Tile sample], page 79, for sample code and picture.

```
dens zdat ['sch']="] [MGL command]
```

```
dens xdat ydat zdat ['sch']="] [MGL command]
```

```
void Dens (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="", mreal zVal=NaN)
```

```
void Dens (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="", mreal
           zVal=NaN)
```

```
void mgl_dens (HMGL gr, HCDT z, const char *sch, const char [C function]
               *opt)
```

```
void mgl_dens_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 207, [cont], page 209, [contf], page 210, [boxs], page 208, [tile], page 209, [dens[xyz]]. See Section 2.6.11 [Dens sample], page 80, for sample code and picture.

```
cont vdat zdat ['sch']="] [MGL command]
```

```
cont vdat xdat ydat zdat ['sch']="] [MGL command]
```

```
void Cont (const mglDataA &v, const mglDataA &z, [Method on mglGraph]
           const char *sch="", const char *opt="")
```

```
void Cont (const mglDataA &v, const mglDataA &x, [Method on mglGraph]
           const mglDataA &y, const mglDataA &z, const char *sch="", const
           char *opt="")
```

```
void mgl_cont__val (HMGL gr, HCDT v, HCDT z, const char *sch, [C function]
                   const char *opt)
```

```
void mgl_cont_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z,      [C function]
                    const char *sch, const char *opt)
```

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 209, [contf], page 210, [contd], page 211, [axial], page 212, cont[xyz]. See Section 2.6.12 [Cont sample], page 81, for sample code and picture.

```
cont zdat ['sch']="]                                     [MGL command]
```

```
cont xdat ydat zdat ['sch']="]                           [MGL command]
```

```
void Cont (const mglDataA &z, const char *sch="",             [Method on mglGraph]
          const char *opt="")
```

```
void Cont (const mglDataA &x, const mglDataA &y,             [Method on mglGraph]
          const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_cont (HMGL gr, HCDT z, const char *sch, const char  [C function]
              *opt)
```

```
void mgl_cont_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                 *sch, const char *opt)
```

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]
```

```
void ContF (const mglDataA &v, const mglDataA &z,           [Method on mglGraph]
            const char *sch="", const char *opt="")
```

```
void ContF (const mglDataA &v, const mglDataA &x,           [Method on mglGraph]
            const mglDataA &y, const mglDataA &z, const char *sch="", const
            char *opt="")
```

```
void mgl_contf_val (HMGL gr, HCDT v, HCDT z, const char *sch, [C function]
                  const char *opt)
```

```
void mgl_contf_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z, [C function]
                     const char *sch, const char *opt)
```

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 209, [cont], page 209, [contd], page 211, contf[xyz]. See Section 2.6.13 [ContF sample], page 82, for sample code and picture.

```
contf zdat ['sch']="]                                     [MGL command]
```

```
contf xdat ydat zdat ['sch']="]                           [MGL command]
```

```
void ContF (const mglDataA &z, const char *sch="",           [Method on mglGraph]
          const char *opt="")
```

```
void ContF (const mglDataA &x, const mglDataA &y,           [Method on mglGraph]
          const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_contf (HMGL gr, HCDT z, const char *sch, const char  [C function]
              *opt)
```

```
void mgl_contf_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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]
```

```
void ContD (const mglDataA &v, const mglDataA &z, [Method on mglGraph]
            const char *sch="", const char *opt="")
```

```
void ContD (const mglDataA &v, const mglDataA &x, [Method on mglGraph]
            const mglDataA &y, const mglDataA &z, const char *sch="", const
            char *opt="")
```

```
void mgl_contd_val (HMGL gr, HCDT v, HCDT z, const char *sch, [C function]
                  const char *opt)
```

```
void mgl_contd_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z, [C function]
                      const char *sch, const char *opt)
```

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 209, [cont], page 209, [contf], page 210. See Section 2.6.14 [ContD sample], page 83, for sample code and picture.

```
contd zdat ['sch']="] [MGL command]
```

```
contd xdat ydat zdat ['sch']="] [MGL command]
```

```
void ContD (const mglDataA &z, const char *sch="", [Method on mglGraph]
            const char *opt="")
```

```
void ContD (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_contd (HMGL gr, HCDT z, const char *sch, const char [C function]
               *opt)
```

```
void mgl_contd_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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]
```

```
void ContV (const mglDataA &v, const mglDataA &z, [Method on mglGraph]
            const char *sch="", const char *opt="")
```

```
void ContV (const mglDataA &v, const mglDataA &x, [Method on mglGraph]
            const mglDataA &y, const mglDataA &z, const char *sch="", const
            char *opt="")
```

```
void mgl_contv_val (HMGL gr, HCDT v, HCDT z, const char *sch, [C function]
                  const char *opt)
```

```
void mgl_contv_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z,      [C function]
                     const char *sch, const char *opt)
```

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 209, [contf], page 210. See Section 2.6.15 [ContV sample], page 84, for sample code and picture.

```
contv zdat ['sch']="]                                     [MGL command]
```

```
contv xdat ydat zdat ['sch']="]                           [MGL command]
```

```
void ContV (const mglDataA &z, const char *sch="",           [Method on mglGraph]
            const char *opt="")
```

```
void ContV (const mglDataA &x, const mglDataA &y,           [Method on mglGraph]
            const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_contv (HMGL gr, HCDT z, const char *sch, const char [C function]
               *opt)
```

```
void mgl_contv_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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]
```

```
void Axial (const mglDataA &v, const mglDataA &z,          [Method on mglGraph]
            const char *sch="", const char *opt="")
```

```
void Axial (const mglDataA &v, const mglDataA &x,          [Method on mglGraph]
            const mglDataA &y, const mglDataA &z, const char *sch="", const
            char *opt="")
```

```
void mgl_axial_val (HMGL gr, HCDT v, HCDT z, const char *sch,    [C function]
                   const char *opt)
```

```
void mgl_axial_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z, [C function]
                      const char *sch, const char *opt)
```

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 209, [contf], page 210, [torus], page 205, [surf3], page 213. See Section 2.6.16 [Axial sample], page 85, for sample code and picture.

```
axial zdat ['sch']="]                                     [MGL command]
```

```
axial xdat ydat zdat ['sch']="]                           [MGL command]
```

```
void Axial (const mglDataA &z, const char *sch="",         [Method on mglGraph]
            const char *opt="", int num=3)
```

```
void Axial (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
            const mglDataA &z, const char *sch="", const char *opt="", int
            num=3)
```



```
void mgl_axial (HMGL gr, HCDT z, const char *sch, const char [C function]
                *opt)
```

```
void mgl_axial_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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]
```

```
void Grid (const mglDataA &z, const char *sch="", [Method on mglGraph]
           const char *opt="")
```

```
void Grid (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void mgl_grid (HMGL gr, HCDT z, const char *sch, const char [C function]
               *opt)
```

```
void mgl_grid_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char [C function]
                  *sch, const char *opt)
```

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 209, [cont], page 209, [contf], page 210, [grid3], page 216, [meshnum], page 160.

### 4.13 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 3.4 [Color scheme], page 149) for plot. String *opt* contain command options (see Section 3.7 [Command options], page 153). See Section 2.7 [3D samples], page 87, for sample code and picture.

```
surf3 adat val ['sch']="] [MGL command]
```

```
surf3 xdat ydat zdat adat val ['sch']="] [MGL command]
```

```
void Surf3 (mreal val, const mglDataA &a, const char [Method on mglGraph]
            *sch="", const char *opt="")
```

```
void Surf3 (mreal val, const mglDataA &x, const [Method on mglGraph]
            mglDataA &y, const mglDataA &z, const mglDataA &a, const char
            *sch="", const char *opt="")
```

```
void mgl_surf3_val (HMGL gr, mreal val, HCDT a, const char [C function]
                   *sch, const char *opt)
```

```
void mgl_surf3_xyz_val (HMGL gr, mreal val, HCDT x, HCDT y, [C function]
                       HCDT z, HCDT a, const char *sch, const char *opt)
```

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 214, [dens3], page 214, [surf3c], page 217, [surf3a], page 218, [axial], page 212. See Section 2.7.1 [Surf3 sample], page 87, for sample code and picture.

```
surf3 adat ['sch']="] [MGL command]
surf3 xdat ydat zdat adat ['sch']="] [MGL command]
void Surf3 (const mglDataA &a, const char *sch="", [Method on mglGraph]
            const char *opt="")
void Surf3 (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &a, const char *sch="", const
            char *opt="")
void mgl_surf3 (HMGL gr, HCDT a, const char *sch, const char [C function]
               *opt)
void mgl_surf3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
                   const char *sch, const char *opt)
```

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]
void Cloud (const mglDataA &a, const char *sch="", [Method on mglGraph]
            const char *opt="")
void Cloud (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &a, const char *sch="", const
            char *opt="")
void mgl_cloud (HMGL gr, HCDT a, const char *sch, const char [C function]
               *opt)
void mgl_cloud_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
                   const char *sch, const char *opt)
```

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 160. 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 213, [meshnum], page 160. See Section 2.7.5 [Cloud sample], page 91, for sample code and picture.

```
dens3 adat ['sch']=" sval=-1] [MGL command]
dens3 xdat ydat zdat adat ['sch']=" sval=-1] [MGL command]
void Dens3 (const mglDataA &a, const char *sch="", [Method on mglGraph]
            mreal sVal=-1, const char *opt="")
void Dens3 (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &a, const char *sch="", mreal
            sVal=-1, const char *opt="")
void mgl_dens3 (HMGL gr, HCDT a, const char *sch, mreal sVal, [C function]
               const char *opt)
```

```
void mgl_dens3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,          [C function]
                  const char *sch, mreal sVal, const char *opt)
```

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 215, [contf3], page 215, [dens], page 209, [grid3], page 216. See Section 2.7.6 [Dens3 sample], page 92, for sample code and picture.

```
cont3 vdat adat ['sch'=" sval=-1]                                     [MGL command]
```

```
cont3 vdat xdat ydat zdat adat ['sch'=" sval=-1]                     [MGL command]
```

```
void Cont3 (const mglDataA &v, const mglDataA &a,                    [Method on mglGraph]
            const char *sch="", mreal sVal=-1, const char *opt="")
```

```
void Cont3 (const mglDataA &v, const mglDataA &x,                    [Method on mglGraph]
            const mglDataA &y, const mglDataA &z, const mglDataA &a, const
            char *sch="", mreal sVal=-1, const char *opt="")
```

```
void mgl_cont3_val (HMGL gr, HCDT v, HCDT a, const char *sch,       [C function]
                  mreal sVal, const char *opt)
```

```
void mgl_cont3_xyz_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT      [C function]
                      z, HCDT a, const char *sch, mreal sVal, const char *opt)
```

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 214, [contf3], page 215, [cont], page 209, [grid3], page 216. See Section 2.7.7 [Cont3 sample], page 93, for sample code and picture.

```
cont3 adat ['sch'=" sval=-1]                                         [MGL command]
```

```
cont3 xdat ydat zdat adat ['sch'=" sval=-1]                         [MGL command]
```

```
void Cont3 (const mglDataA &a, const char *sch="",                  [Method on mglGraph]
            mreal sVal=-1, const char *opt="", const char *opt="")
```

```
void Cont3 (const mglDataA &x, const mglDataA &y,                  [Method on mglGraph]
            const mglDataA &z, const mglDataA &a, const char *sch="", mreal
            sVal=-1, const char *opt="")
```

```
void mgl_cont3 (HMGL gr, HCDT a, const char *sch, mreal sVal,      [C function]
               const char *opt)
```

```
void mgl_cont3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,      [C function]
                  const char *sch, mreal sVal, const char *opt)
```

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]
```

```
void Contf3 (const mglDataA &v, const mglDataA &a,                [Method on mglGraph]
            const char *sch="", mreal sVal=-1, const char *opt="")
```

```

void Contf3 (const mglDataA &v, const mglDataA &x,          [Method on mglGraph]
             const mglDataA &y, const mglDataA &z, const mglDataA &a, const
             char *sch="", mreal sVal=-1, const char *opt="")
void mgl_contf3_val (HMGL gr, HCDT v, HCDT a, const char *sch,      [C function]
                    mreal sVal, const char *opt)
void mgl_contf3_xyz_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT      [C function]
                        z, HCDT a, const char *sch, mreal sVal, const char *opt)

```

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 214, [cont3], page 215, [contf], page 210, [grid3], page 216. See Section 2.7.8 [ContF3 sample], page 94, for sample code and picture.

```

contf3 adat ['sch']=" sval=-1]                                [MGL command]
contf3 xdat ydat zdat adat ['sch']=" sval=-1]                [MGL command]
void Contf3 (const mglDataA &a, const char *sch="",          [Method on mglGraph]
             mreal sVal=-1, const char *opt="", const char *opt="")
void Contf3 (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
             const mglDataA &z, const mglDataA &a, const char *sch="", mreal
             sVal=-1, const char *opt="")
void mgl_contf3 (HMGL gr, HCDT a, const char *sch, mreal sVal,    [C function]
                const char *opt)
void mgl_contf3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,    [C function]
                    const char *sch, mreal sVal, const char *opt)

```

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]
void Grid3 (const mglDataA &a, const char *sch="",          [Method on mglGraph]
            mreal sVal=-1, const char *opt="")
void Grid3 (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
            const mglDataA &z, const mglDataA &a, const char *sch="", mreal
            sVal=-1, const char *opt="")
void mgl_grid3 (HMGL gr, HCDT a, const char *sch, mreal sVal,    [C function]
                const char *opt)
void mgl_grid3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,    [C function]
                    const char *sch, mreal sVal, const char *opt)

```

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 215, [contf3], page 215, [dens3], page 214, [grid2], page 213, [meshnum], page 160.

```

beam tr g1 g2 adat rval ['sch']=" flag=0 num=3]            [MGL command]
void Beam (const mglDataA &tr, const mglDataA &g1,        [Method on mglGraph]
           const mglDataA &g2, const mglDataA &a, mreal r, const char *stl="",
           int flag=0, int num=3)

```

```

void Beam (mreal val, const mglDataA &tr, const [Method on mglGraph]
           mglDataA &g1, const mglDataA &g2, const mglDataA &a, mreal r, const
           char *stl="", int flag=0)
void mgl_beam (HMGL gr, HCDT tr, HCDT g1, HCDT g2, HCDT a, mreal [C function]
              r, const char *stl, int flag, int num)
void mgl_beam_val (HMGL gr, mreal val, HCDT tr, HCDT g1, HCDT [C function]
                  g2, HCDT a, mreal r, const char *stl, int flag)

```

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  $n_x > 2$ . The y-size of data arrays *tr*, *g1*, *g2* and z-size of the data array *a* must be equal. See also [surf3], page 213.

## 4.14 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 3.4 [Color scheme], page 149) for plot. String *opt* contain command options (see Section 3.7 [Command options], page 153).

```

surfC zdat cdat ['sch']="] [MGL command]
surfC xdat ydat zdat cdat ['sch']="] [MGL command]
void SurfC (const mglDataA &z, const mglDataA &c, [Method on mglGraph]
            const char *sch="", const char *opt="")
void SurfC (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &c, const char *sch="", const
            char *opt="")
void mgl_surfC (HMGL gr, HCDT z, HCDT c, const char *sch, const [C function]
               char *opt)
void mgl_surfC_xy (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c, [C function]
                  const char *sch, const char *opt)

```

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 207, [surfa], page 218, [surfca], page 219, [surf3c], page 217. See Section 2.6.2 [SurfC sample], page 72, for sample code and picture.

```

surf3C adat cdat val ['sch']="] [MGL command]
surf3C xdat ydat zdat adat cdat val ['sch']="] [MGL command]
void Surf3C (mreal val, const mglDataA &a, const [Method on mglGraph]
             mglDataA &c, const char *sch="", const char *opt="")

```

```

void Surf3C (mreal val, const mglDataA &x, const          [Method on mglGraph]
             mglDataA &y, const mglDataA &z, const mglDataA &a, const mglDataA
             &c, const char *sch="", const char *opt="")
void mgl_surf3c_val (HMGL gr, mreal val, HCDT a, HCDT c, const      [C function]
                    char *sch, const char *opt)
void mgl_surf3c_xyz_val (HMGL gr, mreal val, HCDT x, HCDT y,        [C function]
                        HCDT z, HCDT a, HCDT c, const char *sch, const char *opt)

```

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 213, 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 213, [surfc], page 217, [surf3a], page 218, [surf3ca], page 220. See Section 2.7.2 [Surf3C sample], page 88, for sample code and picture.

```

surf3c adat cdat ['sch']="]                                [MGL command]
surf3c xdat ydat zdat adat cdat ['sch']="]                [MGL command]
void Surf3C (const mglDataA &a, const mglDataA &c,          [Method on mglGraph]
             const char *sch="", const char *opt="")
void Surf3C (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
             const mglDataA &z, const mglDataA &a, const mglDataA &c, const
             char *sch="", const char *opt="")
void mgl_surf3c (HMGL gr, HCDT a, HCDT c, const char *sch,    [C function]
                 const char *opt)
void mgl_surf3c_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
                    HCDT c, const char *sch, const char *opt)

```

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]
void Surfa (const mglDataA &z, const mglDataA &c,          [Method on mglGraph]
            const char *sch="", const char *opt="")
void Surfa (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
            const mglDataA &z, const mglDataA &c, const char *sch="", const
            char *opt="")
void mgl_surfa (HMGL gr, HCDT z, HCDT c, const char *sch, const [C function]
                char *opt)
void mgl_surfa_xy (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c,    [C function]
                  const char *sch, const char *opt)

```

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 207, [surfc], page 217, [surfca], page 219, [surf3a], page 218. See Section 2.6.3 [SurfA sample], page 73, for sample code and picture.

```

surf3a adat cdat val ['sch']="]                            [MGL command]
surf3a xdat ydat zdat adat cdat val ['sch']="]            [MGL command]

```

```

void Surf3A (mreal val, const mglDataA &a, const           [Method on mglGraph]
             mglDataA &c, const char *sch="", const char *opt="")
void Surf3A (mreal val, const mglDataA &x, const           [Method on mglGraph]
             mglDataA &y, const mglDataA &z, const mglDataA &a, const mglDataA
             &c, const char *sch="", const char *opt="")
void mgl_surf3a_val (HMGL gr, mreal val, HCDT a, HCDT c, const [C function]
                    char *sch, const char *opt)
void mgl_surf3a_xyz_val (HMGL gr, mreal val, HCDT x, HCDT y, [C function]
                        HCDT z, HCDT a, HCDT c, const char *sch, const char *opt)

```

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 213, 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 213, [surf3c], page 217, [surf3a], page 218, [surf3ca], page 220. See Section 2.7.3 [Surf3A sample], page 89, for sample code and picture.

```

surf3a adat cdat ['sch']="] [MGL command]
surf3a xdat ydat zdat adat cdat ['sch']="] [MGL command]
void Surf3A (const mglDataA &a, const mglDataA &c, [Method on mglGraph]
             const char *sch="", const char *opt="")
void Surf3A (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
             const mglDataA &z, const mglDataA &a, const mglDataA &c, const
             char *sch="", const char *opt="")
void mgl_surf3a (HMGL gr, HCDT a, HCDT c, const char *sch, [C function]
                 const char *opt)
void mgl_surf3a_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
                    HCDT c, const char *sch, const char *opt)

```

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]
void SurfCA (const mglDataA &z, const mglDataA &c, [Method on mglGraph]
             const mglDataA &a, const char *sch="", const char *opt="")
void SurfCA (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
             const mglDataA &z, const mglDataA &c, const mglDataA &a, const
             char *sch="", const char *opt="")
void mgl_surfca (HMGL gr, HCDT z, HCDT c, HCDT a, const char [C function]
                 *sch, const char *opt)
void mgl_surfca_xy (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c, [C function]
                    HCDT a, const char *sch, const char *opt)

```

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 221-like coloring if use '%' in color scheme.

See also [surf], page 207, [surfc], page 217, [surfa], page 218, [surf3ca], page 220. See Section 2.6.4 [SurfCA sample], page 74, for sample code and picture.

```
surf3ca adat cdat bdat val ['sch']=" [MGL command]
surf3ca xdat ydat zdat adat cdat bdat val ['sch']=" [MGL command]
void Surf3CA (mreal val, const mglDataA &a, const [Method on mglGraph]
              mglDataA &c, const mglDataA &b, const char *sch="", const char
              *opt="")
void Surf3CA (mreal val, const mglDataA &x, const [Method on mglGraph]
              mglDataA &y, const mglDataA &z, const mglDataA &a, const mglDataA
              &c, const mglDataA &b, const char *sch="", const char *opt="")
void mgl_surf3ca_val (HMGL gr, mreal val, HCDT a, HCDT c, HCDT [C function]
                     b, const char *sch, const char *opt)
void mgl_surf3ca_xyz_val (HMGL gr, mreal val, HCDT x, HCDT y, [C function]
                          HCDT z, HCDT a, HCDT c, HCDT b, const char *sch, const char *opt)
```

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 213, 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 221-like coloring if use '%' in color scheme. See also [surf3], page 213, [surfca], page 219, [surf3c], page 217, [surf3a], page 218. See Section 2.7.4 [Surf3CA sample], page 90, for sample code and picture.

```
surf3ca adat cdat bdat ['sch']=" [MGL command]
surf3ca xdat ydat zdat adat cdat bdat ['sch']=" [MGL command]
void Surf3CA (const mglDataA &a, const mglDataA &c, [Method on mglGraph]
              const mglDataA &b, const char *sch="", const char *opt="")
void Surf3CA (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
              const mglDataA &z, const mglDataA &a, const mglDataA &c, const
              mglDataA &b, const char *sch="", const char *opt="")
void mgl_surf3ca (HMGL gr, HCDT a, HCDT c, HCDT b, const char [C function]
                  *sch, const char *opt)
void mgl_surf3ca_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
                      HCDT c, HCDT b, const char *sch, const char *opt)
```

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]
void TileS (const mglDataA &z, const mglDataA &c, [Method on mglGraph]
            const char *sch="", const char *opt="")
void TileS (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &r, const char *sch="", const
            char *opt="")
void mgl_tiles (HMGL gr, HCDT z, HCDT c, const char *sch, const [C function]
                char *opt)
```



```
void mgl_tiles_xy (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT r,          [C function]
                  const char *sch, const char *opt)
```

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 209, 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 218, [tile], page 209. See Section 2.6.10 [TileS sample], page 80, for sample code and picture.

```
map udat vdat ['sch']="]                                     [MGL command]
```

```
map xdat ydat udat vdat ['sch']="]                           [MGL command]
```

```
void Map (const mglDataA &ax, const mglDataA &ay,              [Method on mglGraph]
          const char *sch="", const char *opt="")
```

```
void Map (const mglDataA &x, const mglDataA &y,                [Method on mglGraph]
          const mglDataA &ax, const mglDataA &ay, const char *sch="", const
          char *opt="")
```

```
void mgl_map (HMGL gr, HCDT ax, HCDT ay, const char *sch, const [C function]
              char *opt)
```

```
void mgl_map_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay,    [C function]
                 const char *sch, const char *opt)
```

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 2.9.9 [Mapping visualization], page 123, for sample code and picture.

```
stfa re im dn ['sch']="]                                     [MGL command]
```

```
stfa xdat ydat re im dn ['sch']="]                           [MGL command]
```

```
void STFA (const mglDataA &re, const mglDataA &im,            [Method on mglGraph]
           int dn, const char *sch="", const char *opt="")
```

```
void STFA (const mglDataA &x, const mglDataA &y,              [Method on mglGraph]
           const mglDataA &re, const mglDataA &im, int dn, const char
           *sch="", const char *opt="")
```

```
void mgl_stfa (HMGL gr, HCDT re, HCDT im, int dn, const char   [C function]
               *sch, const char *opt)
```

```
void mgl_stfa_xy (HMGL gr, HCDT x, HCDT y, HCDT re, HCDT im, int [C function]
                  dn, const char *sch, const char *opt)
```

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  $res[i, j] = |\sum_d^n \exp(I * j * d) * (re[i * dn + d] + I * im[i * dn + d])| / dn$  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 2.9.8 [STFA sample], page 122, for sample code and picture.

## 4.15 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 3.4 [Color scheme], page 149) for plot. String  $opt$  contain command options (see Section 3.7 [Command options], page 153).

```

traj xdat ydat udat vdat ['sch']=" [MGL command]
traj xdat ydat zdat udat vdat wdat ['sch']=" [MGL command]
void Traj (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &ax, const mglDataA &ay, const char *sch="", const
           char *opt="")
void Traj (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const mglDataA &ax, const mglDataA &ay, const
           mglDataA &az, const char *sch="", const char *opt="")
void mgl_traj_xyz (HMGL gr, HCDTx, HCDTy, HCDTz, HCDTax, [C function]
                  HCDTay, HCDTaz, const char *sch, const char *opt)
void mgl_traj_xy (HMGL gr, HCDTx, HCDTy, HCDTax, HCDTay, const [C function]
                  char *sch, const char *opt)

```

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 3.3 [Line styles], page 147). By default ( $pen=""$ ) color from palette is used (see Section 4.2.7 [Palette and colors], page 162). 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 222. See Section 2.8.3 [Traj sample], page 106, for sample code and picture.

```

vect udat vdat ['sch']=" [MGL command]
vect xdat ydat udat vdat ['sch']=" [MGL command]
void Vect (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
           const char *sch="", const char *opt="")
void Vect (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &ax, const mglDataA &ay, const char *sch="", const
           char *opt="")
void mgl_vect_2d (HMGL gr, HCDT ax, HCDT ay, const char *sch, [C function]
                  const char *opt)
void mgl_vect_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay, [C function]
                  const char *sch, const char *opt)

```

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 160. 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 224, [dew], page 224. See Section 2.8.1 [Vect sample], page 104, for sample code and picture.

```

vect udat vdat wdat ['sch']=" [MGL command]
vect xdat ydat zdat udat vdat wdat ['sch']=" [MGL command]
void Vect (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
           const mglDataA &az, const char *sch="", const char *opt="")
void Vect (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &z, const mglDataA &ax, const mglDataA &ay, const
           mglDataA &az, const char *sch="", const char *opt="")
void mgl_vect_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const [C function]
                 char *sch, const char *opt)
void mgl_vect_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, [C function]
                  HCDT ay, HCDT az, const char *sch, const char *opt)

```

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]
void Vect3 (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
            const mglDataA &az, const char *sch="", mreal sVal=-1, const char
            *opt="")
void Vect3 (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
            const mglDataA &z, const mglDataA &ax, const mglDataA &ay, const
            mglDataA &az, const char *sch="", mreal sVal=-1, const char
            *opt="")
void mgl_vect3 (HMGL gr, HCDT ax, HCDT ay, HCDT az, const char [C function]
               *sch, mreal sVal, const char *opt)
void mgl_vect3_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, [C function]
                   HCDT ay, HCDT az, const char *sch, mreal sVal, const char *opt)

```

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 160. 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 222, [flow], page 224, [dew], page 224. See Section 2.8.2 [Vect3 sample], page 105, for sample code and picture.

```
dew udat vdat ['sch']=" [MGL command]
dew xdat ydat udat vdat ['sch']=" [MGL command]
void Dew (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
          const char *sch="", const char *opt="")
void Dew (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
          const mglDataA &ax, const mglDataA &ay, const char *sch="", const
          char *opt="")
void mgl_dew (HMGL gr, HCDT ax, HCDT ay, const char *sch, const [C function]
             char *opt)
void mgl_dew_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay, [C function]
                const char *sch, const char *opt)
```

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 160. See also [vect], page 222. See Section 2.8.6 [Dew sample], page 109, for sample code and picture.

```
flow udat vdat ['sch']=" [MGL command]
flow xdat ydat udat vdat ['sch']=" [MGL command]
void Flow (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
           const char *sch="", const char *opt="")
void Flow (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &ax, const mglDataA &ay, const char *sch="", const
           char *opt="")
void mgl_flow_2d (HMGL gr, HCDT ax, HCDT ay, const char *sch, [C function]
                 const char *opt)
void mgl_flow_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay, [C function]
                 const char *sch, const char *opt)
```

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;
- ‘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 226, [vect], page 222, [tape], page 196, [barwidth], page 160. See Section 2.8.4 [Flow sample], page 107, for sample code and picture.

```
flow udat vdat wdat ['sch']=" [MGL command]
flow xdat ydat zdat udat vdat wdat ['sch']=" [MGL command]
void Flow (const mglDataA &ax, const mglDataA &ay, [Method on mglGraph]
           const mglDataA &az, const char *sch="", const char *opt="")
```

```

void Flow (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const mglDataA &ax, const mglDataA &ay, const
           mglDataA &az, const char *sch="", const char *opt="")
void mgl_flow_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const [C function]
                 char *sch, const char *opt)
void mgl_flow_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, [C function]
                  HCDT ay, HCDT az, const char *sch, const char *opt)

```

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]
void FlowP (mglPoint p0, const mglDataA &ax, const [Method on mglGraph]
            mglDataA &ay, const char *sch="", const char *opt="")
void FlowP (mglPoint p0, const mglDataA &x, const [Method on mglGraph]
            mglDataA &y, const mglDataA &ax, const mglDataA &ay, const char
            *sch="", const char *opt="")
void mgl_flowp_2d (HMGL gr, mreal x0, mreal y0, mreal z0, HCDT [C function]
                  ax, HCDT ay, const char *sch, const char *opt)
void mgl_flowp_xy (HMGL gr, mreal x0, mreal y0, mreal z0, HCDT [C function]
                  x, HCDT y, HCDT ax, HCDT ay, const char *sch, const char *opt)

```

The same as first one ([flow], page 224) 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]
void FlowP (mglPoint p0, const mglDataA &ax, const [Method on mglGraph]
            mglDataA &ay, const mglDataA &az, const char *sch="", const char
            *opt="")
void FlowP (mglPoint p0, const mglDataA &x, const [Method on mglGraph]
            mglDataA &y, const mglDataA &z, const mglDataA &ax, const mglDataA
            &ay, const mglDataA &az, const char *sch="", const char *opt="")
void mgl_flowp_3d (HMGL gr, mreal x0, mreal y0, mreal z0, HCDT [C function]
                  ax, HCDT ay, HCDT az, const char *sch, const char *opt)
void mgl_flowp_xyz (HMGL gr, mreal x0, mreal y0, mreal z0, HCDT [C function]
                   x, HCDT y, HCDT z, HCDT ax, HCDT ay, HCDT az, const char *sch, const
                   char *opt)

```

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]
void Grad (const mglDataA &phi, const char *sch="", [Method on mglGraph]
           const char *opt="")
void Grad (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
           const mglDataA &phi, const char *sch="", const char *opt="")

```

```

void Grad (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const mglDataA &phi, const char *sch="", const
           char *opt="")
void mgl_grad (HMGL gr, HCDT phi, const char *sch, const char      [C function]
               *opt)
void mgl_grad_xy (HMGL gr, HCDT x, HCDT y, HCDT phi, const char    [C function]
                  *sch, const char *opt)
void mgl_grad_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT phi,      [C function]
                  const char *sch, const char *opt)

```

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 209, [cont], page 209, [flow], page 224.

```

pipe udat vdat ['sch'=" r0=0.05]                                [MGL command]
pipe xdat ydat udat vdat ['sch'=" r0=0.05]                      [MGL command]
void Pipe (const mglDataA &ax, const mglDataA &ay,          [Method on mglGraph]
           const char *sch="", mreal r0=0.05, const char *opt="")
void Pipe (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &ax, const mglDataA &ay, const char *sch="", mreal
           r0=0.05, const char *opt="")
void mgl_pipe_2d (HMGL gr, HCDT ax, HCDT ay, const char *sch,      [C function]
                  mreal r0, const char *opt)
void mgl_pipe_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay,      [C function]
                  const char *sch, mreal r0, const char *opt)

```

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 224, [vect], page 222. See Section 2.8.5 [Pipe sample], page 108, 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]
void Pipe (const mglDataA &ax, const mglDataA &ay,          [Method on mglGraph]
           const mglDataA &az, const char *sch="", mreal r0=0.05, const char
           *opt="")
void Pipe (const mglDataA &x, const mglDataA &y,          [Method on mglGraph]
           const mglDataA &z, const mglDataA &ax, const mglDataA &ay, const
           mglDataA &az, const char *sch="", mreal r0=0.05, const char
           *opt="")
void mgl_pipe_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const      [C function]
                  char *sch, mreal r0, const char *opt)

```

```
void mgl_pipe_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax,      [C function]
                  HCDT ay, HCDT az, const char *sch, mreal r0, const char *opt)
```

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  $\sqrt{ax^2 + ay^2 + az^2}$ .

## 4.16 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 3.4 [Color scheme], page 149.

```
densx dat ['sch']=" sval=nan] [MGL command]
densy dat ['sch']=" sval=nan] [MGL command]
densz dat ['sch']=" sval=nan] [MGL command]
void DensX (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void DensY (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void DensZ (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void mgl_dens_x (HMGL gr, HCDT a, const char *stl, mreal sVal, [C function]
                const char *opt)
void mgl_dens_y (HMGL gr, HCDT a, const char *stl, mreal sVal, [C function]
                const char *opt)
void mgl_dens_z (HMGL gr, HCDT a, const char *stl, mreal sVal, [C function]
                const char *opt)
```

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 227, [ContFXYZ], page 228, [dens], page 209, Section 4.18 [Data manipulation], page 235. See Section 2.7.9 [Dens projection sample], page 95, 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]
void ContX (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void ContY (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void ContZ (const mglDataA &a, const char *stl="", [Method on mglGraph]
           mreal sVal=NAN, const char *opt="")
void mgl_cont_x (HMGL gr, HCDT a, const char *stl, mreal sVal, [C function]
                const char *opt)
```

```
void mgl_cont_y (HMGL gr, HCDT a, const char *stl, mreal sVal,      [C function]
                const char *opt)
```

```
void mgl_cont_z (HMGL gr, HCDT a, const char *stl, mreal sVal,      [C function]
                const char *opt)
```

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 228, [DensXYZ], page 227, [cont], page 209, Section 4.18 [Data manipulation], page 235. See Section 2.7.10 [Cont projection sample], page 96, for sample code and picture.

```
void ContX (const mglDataA &v, const mglDataA &a,                  [Method on mglGraph]
            const char *stl="", mreal sVal=NAN, const char *opt="")
```

```
void ContY (const mglDataA &v, const mglDataA &a,                  [Method on mglGraph]
            const char *stl="", mreal sVal=NAN, const char *opt="")
```

```
void ContZ (const mglDataA &v, const mglDataA &a,                  [Method on mglGraph]
            const char *stl="", mreal sVal=NAN, const char *opt="")
```

```
void mgl_cont_x_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                    mreal sVal, const char *opt)
```

```
void mgl_cont_y_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                    mreal sVal, const char *opt)
```

```
void mgl_cont_z_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                    mreal sVal, const char *opt)
```

The same as previous with manual contour levels.

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

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

```
contfz dat ['sch'=" sval=nan]                                     [MGL command]
```

```
void ContFX (const mglDataA &a, const char *stl="",                [Method on mglGraph]
             mreal sVal=NAN, const char *opt="")
```

```
void ContFY (const mglDataA &a, const char *stl="",                [Method on mglGraph]
             mreal sVal=NAN, const char *opt="")
```

```
void ContFZ (const mglDataA &a, const char *stl="",                [Method on mglGraph]
             mreal sVal=NAN, const char *opt="")
```

```
void mgl_contf_x (HMGL gr, HCDT a, const char *stl, mreal sVal,    [C function]
                  const char *opt)
```

```
void mgl_contf_y (HMGL gr, HCDT a, const char *stl, mreal sVal,    [C function]
                  const char *opt)
```

```
void mgl_contf_z (HMGL gr, HCDT a, const char *stl, mreal sVal,    [C function]
                  const char *opt)
```

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 228, [DensXYZ], page 227, [cont], page 209, Section 4.18 [Data manipulation], page 235. See Section 2.7.11 [ContF projection sample], page 97, for sample code and picture.



```

void ContFX (const mglDataA &v, const mglDataA &a,      [Method on mglGraph]
             const char *stl="", mreal sVal=NAN, const char *opt="")
void ContFY (const mglDataA &v, const mglDataA &a,      [Method on mglGraph]
             const char *stl="", mreal sVal=NAN, const char *opt="")
void ContFZ (const mglDataA &v, const mglDataA &a,      [Method on mglGraph]
             const char *stl="", mreal sVal=NAN, const char *opt="")
void mgl_contf_x_val (HMGL gr, HCDT v, HCDT a, const char *stl, [C function]
                    mreal sVal, const char *opt)
void mgl_contf_y_val (HMGL gr, HCDT v, HCDT a, const char *stl, [C function]
                    mreal sVal, const char *opt)
void mgl_contf_z_val (HMGL gr, HCDT v, HCDT a, const char *stl, [C function]
                    mreal sVal, const char *opt)

```

The same as previous with manual contour levels.

```

fplot 'y(x)' ['pen']="]                                [MGL command]
void FPlot (const char *eqY, const char *pen="",        [Method on mglGraph]
            const char *opt="")
void mgl_fplot (HMGL gr, const char *eqY, const char *pen, [C function]
               const char *opt)

```

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 194.

```

fplot 'x(t)' 'y(t)' 'z(t)' ['pen']="]                [MGL command]
void FPlot (const char *eqX, const char *eqY, const      [Method on mglGraph]
            char *eqZ, const char *pen, const char *opt="")
void mgl_fplot_xyz (HMGL gr, const char *eqX, const char [C function]
                   *eqY, const char *eqZ, const char *pen, const char *opt)

```

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 194.

```

fsurf 'z(x,y)' ['sch']="]                              [MGL command]
void FSurf (const char *eqZ, const char *sch="",        [Method on mglGraph]
            const char *opt="");
void mgl_fsurf (HMGL gr, const char *eqZ, const char *sch, [C function]
               const char *opt);

```

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 207.

```

fsurf 'x(u,v)' 'y(u,v)' 'z(u,v)' ['sch']="]          [MGL command]
void FSurf (const char *eqX, const char *eqY, const      [Method on mglGraph]
            char *eqZ, const char *sch="", const char *opt="")
void mgl_fsurf_xyz (HMGL gr, const char *eqX, const char [C function]
                   *eqY, const char *eqZ, const char *sch, const char *opt)

```

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 207.

```

triplot idat xdat ydat ['sch']="] [MGL command]
triplot idat xdat ydat zdat ['sch']="] [MGL command]
triplot idat xdat ydat zdat cdat ['sch']="] [MGL command]
void TriPlot (const mglDataA &id, const mglDataA &x, [Method on mglGraph]
               const mglDataA &y, const char *sch="", const char *opt="")
void TriPlot (const mglDataA &id, const mglDataA &x, [Method on mglGraph]
               const mglDataA &y, const mglDataA &z, const mglDataA &c, const
               char *sch="", const char *opt="")
void TriPlot (const mglDataA &id, const mglDataA &x, [Method on mglGraph]
               const mglDataA &y, const mglDataA &z, const char *sch="", const
               char *opt="")
void mgl_triplot_xy (HMGL gr, HCDT id, HCDT x, HCDT y, const [C function]
                    char *sch, const char *opt)
void mgl_triplot_xyz (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT z, [C function]
                     const char *sch, const char *opt)
void mgl_triplot_xyzc (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT z, [C function]
                      HCDT c, const char *sch, const char *opt)

```

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 231, [crust], page 232, [quadplot], page 231, [triangulation], page 280. See Section 2.7.12 [TriPlot and QuadPlot], page 98, 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]
void TriCont (const mglDataA &id, const mglDataA &x, [Method on mglGraph]
               const mglDataA &y, const mglDataA &z, const mglDataA &c, const
               char *sch="", const char *opt="")
void TriCont (const mglDataA &id, const mglDataA &x, [Method on mglGraph]
               const mglDataA &y, const mglDataA &z, const char *sch="", const
               char *opt="")
void TriContV (const mglDataA &v, const mglDataA [Method on mglGraph]
               &id, const mglDataA &x, const mglDataA &y, const mglDataA &z, const
               mglDataA &c, const char *sch="", const char *opt="")
void TriContV (const mglDataA &v, const mglDataA [Method on mglGraph]
               &id, const mglDataA &x, const mglDataA &y, const mglDataA &z, const
               char *sch="", const char *opt="")
void mgl_tricont_xyzc (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT z, [C function]
                      HCDT c, const char *sch, const char *opt)
void mgl_tricont_xyz (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT z, [C function]
                     const char *sch, const char *opt)
void mgl_tricont_xyzcv (HMGL gr, HCDT v, HCDT id, HCDT x, HCDT [C function]
                       y, HCDT z, HCDT c, const char *sch, const char *opt)

```

```
void mgl_tricont_xyzv (HMGL gr, HCDT v, HCDT id, HCDT x, HCDT y,    [C function]
                      HCDT z, const char *sch, const char *opt)
```

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 230, [cont], page 209, [triangulation], page 280.

```
quadplot idat xdat ydat ['sch']="]                                [MGL command]
```

```
quadplot idat xdat ydat zdat ['sch']="]                          [MGL command]
```

```
quadplot idat xdat ydat zdat cdat ['sch']="]                     [MGL command]
```

```
void QuadPlot (const mglDataA &id, const mglDataA                [Method on mglGraph]
               &x, const mglDataA &y, const char *sch="", const char *opt="")
```

```
void QuadPlot (const mglDataA &id, const mglDataA                [Method on mglGraph]
               &x, const mglDataA &y, const mglDataA &z, const mglDataA &c, const
               char *sch="", const char *opt="")
```

```
void QuadPlot (const mglDataA &id, const mglDataA                [Method on mglGraph]
               &x, const mglDataA &y, const mglDataA &z, const char *sch="",
               const char *opt="")
```

```
void mgl_quadplot_xy (HMGL gr, HCDT id, HCDT x, HCDT y, const    [C function]
                     char *sch, const char *opt)
```

```
void mgl_quadplot_xyz (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT z, [C function]
                      const char *sch, const char *opt)
```

```
void mgl_quadplot_xyzc (HMGL gr, HCDT id, HCDT x, HCDT y, HCDT   [C function]
                       z, HCDT c, const char *sch, const char *opt)
```

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 230. See Section 2.7.12 [TriPlot and QuadPlot], page 98, for sample code and picture.

```
dots xdat ydat zdat ['sch']="]                                    [MGL command]
```

```
dots xdat ydat zdat adat ['sch']="]                              [MGL command]
```

```
void Dots (const mglDataA &x, const mglDataA &y,                 [Method on mglGraph]
           const mglDataA &z, const char *sch="", const char *opt="")
```

```
void Dots (const mglDataA &x, const mglDataA &y,                 [Method on mglGraph]
           const mglDataA &z, const mglDataA &a, const char *sch="", const
           char *opt="")
```

```
void Dots (const mglDataA &x, const mglDataA &y,                 [Method on mglGraph]
           const mglDataA &z, const mglDataA &c, const mglDataA &a, const
           char *sch="", const char *opt="")
```

```
void mgl_dots (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, [C function]
               const char *opt)
```

```
void mgl_dots_a (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, const char *sch, const char *opt) [C function]
```

```
void mgl_dots_ca (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c, HCDT a, const char *sch, const char *opt) [C function]
```

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 195, plot with style ‘.’ to draw non-transparent dots with specified colors. Arrays *x*, *y*, *z*, *a* must have equal sizes. See also [crust], page 232, [tens], page 195, [mark], page 201, [plot], page 194. See Section 2.7.13 [Dots sample], page 99, for sample code and picture.

```
crust xdat ydat zdat ['sch']=" [MGL command]
```

```
void Crust (const mglDataA &x, const mglDataA &y, const mglDataA &z, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void mgl_crust (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]
```

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 231, [triplot], page 230.

## 4.17 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 4.2.9 [Error handling], page 163).

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 2.9.13 [Nonlinear fitting hints], page 128, 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]
```

```
mglData FitS (const mglDataA &a, const mglDataA &s, const char *func, const char *var, const char *opt="") [Method on mglGraph]
```

```
mglData FitS (const mglDataA &a, const mglDataA &s, const char *func, const char *var, mglData &ini, const char *opt="") [Method on mglGraph]
```

```

mglData FitS (const mglDataA &x, const mglDataA &a,      [Method on mglGraph]
               const mglDataA &s, const char *func, const char *var, const char
               *opt="")
mglData FitS (const mglDataA &x, const mglDataA &a,      [Method on mglGraph]
               const mglDataA &s, const char *func, const char *var, mglData &ini,
               const char *opt="")
mglData FitS (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
               const mglDataA &a, const mglDataA &s, const char *func, const char
               *var, const char *opt="")
mglData FitS (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
               const mglDataA &a, const mglDataA &s, const char *func, const char
               *var, mglData &ini, const char *opt="")
mglData FitS (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
               const mglDataA &z, const mglDataA &a, const mglDataA &s, const
               char *func, const char *var, const char *opt="")
mglData FitS (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
               const mglDataA &z, const mglDataA &a, const mglDataA &s, const
               char *func, const char *var, mglData &ini, const char *opt="")
HMDT mgl_fit_ys (HMGL gr, HCDT a, HCDT s, const char *func,      [C function]
                const char *var, HMDT ini, const char *opt)
HMDT mgl_fit_xys (HMGL gr, HCDT x, HCDT a, HCDT s, const char      [C function]
                 *func, const char *var, HMDT ini, const char *opt)
HMDT mgl_fit_xyzs (HMGL gr, HCDT x, HCDT y, HCDT a, HCDT s,      [C function]
                  const char *func, const char *var, HMDT ini, const char *opt)
HMDT mgl_fit_xyzas (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,      [C function]
                   HCDT s, const char *func, const char *var, HMDT ini, const char *opt)
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]
mglData Fit (const mglDataA &a, const char *func,      [Method on mglGraph]
             const char *var, const char *opt="")
mglData Fit (const mglDataA &a, const char *func,      [Method on mglGraph]
             const char *var, mglData &ini, const char *opt="")
mglData Fit (const mglDataA &x, const mglDataA &a,      [Method on mglGraph]
             const char *func, const char *var, const char *opt="")
mglData Fit (const mglDataA &x, const mglDataA &a,      [Method on mglGraph]
             const char *func, const char *var, mglData &ini, const char
             *opt="")
mglData Fit (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
             const mglDataA &a, const char *func, const char *var, const char
             *opt="")

```

```

mglData Fit (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
             const mglDataA &a, const char *func, const char *var, mglData &ini,
             const char *opt="")
mglData Fit (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
             const mglDataA &z, const mglDataA &a, const char *func, const char
             *var, const char *opt="")
mglData Fit (const mglDataA &x, const mglDataA &y,      [Method on mglGraph]
             const mglDataA &z, const mglDataA &a, const char *func, const char
             *var, mglData &ini, const char *opt="")
HMDT mgl_fit_y (HMGL gr, HCDT a, const char *func, const char      [C function]
               *var, HMDT ini, const char *opt)
HMDT mgl_fit_xy (HMGL gr, HCDT x, HCDT a, const char *func,      [C function]
                const char *var, HMDT ini, const char *opt)
HMDT mgl_fit_xyz (HMGL gr, HCDT x, HCDT y, HCDT a, const char      [C function]
                 *func, const char *var, HMDT ini, const char *opt)
HMDT mgl_fit_xyza (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a,      [C function]
                  const char *func, const char *var, HMDT ini, const char *opt)
    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.

mglData Fit2 (const mglDataA &a, const char *func,      [Method on mglGraph]
              const char *var, const char *opt="")
mglData Fit2 (mglData &fit, const mglDataA &a, const      [Method on mglGraph]
              char *func, const char *var, mglData &ini, const char *opt="")
mglData Fit3 (mglData &fit, const mglDataA &a, const      [Method on mglGraph]
              char *func, const char *var, const char *opt="")
mglData Fit3 (mglData &fit, const mglDataA &a, const      [Method on mglGraph]
              char *func, const char *var, mglData &ini, const char *opt="")
HMDT mgl_fit_2 (HMGL gr, HCDT a, const char *func, const char      [C function]
               *var, HMDT ini, const char *opt)
HMDT mgl_fit_3 (HMGL gr, HCDT a, const char *func, const char      [C function]
               *var, HMDT ini, const char *opt)
    Fit data along all directions for 2d or 3d arrays a with s=1 and x, y, z equidistantly
    distributed in axis range.

putsfit x y ['pre'=" 'fnt'=" size=-1]                  [MGL command]
void PutsFit (mglPoint p, const char *prefix="",      [Method on mglGraph]
             const char *font="", mreal size=-1)
void mgl_puts_fit (HMGL gr, mreal x, mreal y, mreal z, const      [C function]
                  char *prefix, const char *font, mreal size)
    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 4.8 [Text printing], page 188.

const char *GetFit ()                                  [Method on mglGraph]
const char * mgl_get_fit (HMGL gr)                    [C function only]
mgl_get_fit (long gr, char *out, int len)              [Fortran subroutine]
    Get last fitted formula with found coefficients (as numbers).

```

```

mreal GetFitChi () [Method on mglGraph]
mreal mgl_get_fit_chi () [C function]
    Get \chi for last fitted formula.

mreal GetFitCovar () [Method on mglGraph]
mreal mgl_get_fit_covar () [C function]
    Get covariance matrix for last fitted formula.

```

## 4.18 Data manipulation

```

hist RES xdat adat [MGL command]
hist RES xdat ydat adat [MGL command]
hist RES xdat ydat zdat adat [MGL command]
mglData Hist (const mglDataA &x, const mglDataA &a, [Method on mglGraph]
    const char *opt="")
mglData Hist (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
    const mglDataA &a, const char *opt="")
mglData Hist (const mglDataA &x, const mglDataA &y, [Method on mglGraph]
    const mglDataA &z, const mglDataA &a, const char *opt="")
HMMDT mgl_hist_x (HMGL gr, HCDT x, HCDT a, const char *opt) [C function]
HMMDT mgl_hist_xy (HMGL gr, HCDT x, HCDT y, HCDT a, const char [C function]
    *opt)
HMMDT mgl_hist_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, [C function]
    const char *opt)

```

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]
void Fill (mglData &u, const char *eq, const char [Method on mglGraph]
    *opt="")
void Fill (mglData &u, const char *eq, const [Method on mglGraph]
    mglDataA &v, const char *opt="")
void Fill (mglData &u, const char *eq, const [Method on mglGraph]
    mglDataA &v, const mglDataA &w, const char *opt="")
void mgl_data_fill_eq (HMGL gr, HMMDT u, const char *eq, HCDTv, [C function]
    HCDTw, const char *opt)

```

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]
void DataGrid (mglData &u, const mglDataA &x, const [Method on mglGraph]
               mglDataA &y, const mglDataA &z, const char *opt="")
void mgl_data_grid (HMGL gr, HMDT u, HCDT x, HCDT y, HCDT z, [C function]
                   const char *opt)

```

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 2.9.11 [Making regular data], page 127, 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]
void Refill (mglDataA &dat, const mglDataA &x, const [Method on mglData]
            mglDataA &y, const mglDataA &z, const char *opt="")
void Refill (mglDataA &dat, const mglDataA &x, const [Method on mglData]
            mglDataA &y, const mglDataA &z, const char *opt="")
void Refill (mglDataA &dat, const mglDataA &x, const [Method on mglData]
            mglDataA &y, const mglDataA &z, const mglDataA &v, const char
            *opt="")
void mgl_data_refill_gr (HMGL gr, HMDT a, HCDT x, HCDT y, HCDT [C function]
                       z, HCDT v, long sl, const char *opt)

```

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]
mglData PDE (const char *ham, const mglDataA [Method on mglGraph]
            &ini_re, const mglDataA &ini_im, mreal dz=0.1, mreal k0=100, const
            char *opt="")
HMDT mgl_pde_solve (HMGL gr, const char *ham, HCDT ini_re, [C function]
                   HCDT ini_im, mreal dz, mreal k0, const char *opt)

```

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 2.9.14 [PDE solving hints], page 130, for sample code and picture.



## 5 Widget classes

There are set of “window” classes for making a window with MathGL graphics: `mglWindow`, `mglFLTK`, `mglQT` and `mglGLUT` for whole window, `Fl_MathGL` and `QMathGL` as widgets. All these classes allow user to show, rotate, export, and change view of the plot using keyboard. Most of them (except `mglGLUT`) also have toolbar and menu for simplifying plot manipulation. All window classes have mostly the same set of functions derived from Section 5.1 [`mglWnd` class], page 238.

For drawing you can use: `NULL` pointer if you’ll update plot manually, global callback function of type `int draw(HMGL gr, void *p)` or `int draw(mglGraph *gr)`, or instance of class derived from Section 5.2 [`mglDraw` class], page 240. Basically, this class have 2 main virtual methods:

```
class mglDraw
{
public:
    virtual int Draw(mglGraph *) { return 0; };
    virtual void Reload() {};
};
```

You should inherit yours class from `mglDraw` and re-implement one or both functions for drawing.

The window can be constructed using one of following classes (see Section 2.1.1 [Using MathGL window], page 7, for examples).

```
mglFLTK (const char *title="MathGL") [Constructor on mglFLTK]
mglFLTK (int (*draw)(HMGL gr, void *p), const char [Constructor on mglFLTK]
        *title="MathGL", void *par=NULL, void (*reload)(HMGL gr, void *p)=0)
mglFLTK (int (*draw)(mglGraph *gr), const char [Constructor on mglFLTK]
        *title="MathGL")
mglFLTK (mglDraw *draw, const char [Constructor on mglFLTK]
        *title="MathGL")
HMGL mgl_create_graph_fltk (int (*draw)(HMGL gr, void *p), [C function]
        const char *title, void *par, void (*reload)(HMGL gr, void *p))
```

Creates a FLTK-based window for plotting. Parameter *draw* sets a pointer to drawing function (this is the name of function) or instance of Section 5.2 [`mglDraw` class], page 240. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Note, that *draw* can be `NULL` for displaying static bitmaps only (no animation or slides). Parameter *title* sets the title of the window. Parameter *par* contains pointer to data for the plotting function *draw*. FLTK-based windows is a bit faster than Qt ones, and provide better support of multi-threading.

```
int RunThr () [Method on mglFLTK]
int mgl_fltk_thr () [C function]
    Run main loop for event handling in separate thread. Note, right now it work for
    FLTK windows only.
```

```

mglQT (const char *title="MathGL") [Constructor on mglQT]
mglQT (int (*draw)(HMGL gr, void *p), const char [Constructor on mglQT]
      *title="MathGL", void *par=NULL, void (*reload)(HMGL gr, void *p)=0)
mglQT (int (*draw)(mglGraph *gr), const char [Constructor on mglQT]
      *title="MathGL")
mglQT (mglDraw *draw, const char *title="MathGL") [Constructor on mglQT]
HMGL mgl_create_graph_qt (int (*draw)(HMGL gr, void *p), [C function]
      const char *title, void *par, void (*reload)(HMGL gr, void *p))

```

Creates a FLTK-based window for plotting. Parameter *draw* sets a pointer to drawing function (this is the name of function) or instance of Section 5.2 [mglDraw class], page 240. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Note, that *draw* can be NULL for displaying static bitmaps only (no animation or slides). Parameter *title* sets the title of the window. Parameter *par* contains pointer to data for the plotting function *draw*.

```

mglGLUT (const char *title="MathGL") [Constructor on mglGLUT]
mglGLUT (int (*draw)(HMGL gr, void *p), const char [Constructor on mglGLUT]
      *title="MathGL", void *par=NULL, void (*reload)(HMGL gr, void *p)=0)
mglGLUT (int (*draw)(mglGraph *gr), const char [Constructor on mglGLUT]
      *title="MathGL")
mglGLUT (mglDraw *draw, const char [Constructor on mglGLUT]
      *title="MathGL")
HMGL mgl_create_graph_glut (int (*draw)(HMGL gr, void *p), [C function]
      const char *title, void *par, void (*reload)(HMGL gr, void *p))

```

Creates a GLUT-based window for plotting. Parameter *draw* sets a pointer to drawing function (this is the name of function) or instance of mglDraw class. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Note, that *draw* can be NULL for displaying static bitmaps only (no animation or slides). Parameter *title* sets the title of the window. Parameter *par* contains pointer to data for the plotting function *draw*. GLUT-based windows are fastest one but there is no toolbar, and plot have some issues due to OpenGL limitations.

There are some keys handles for manipulating by the plot: 'a', 'd', 'w', 's' for the rotating; ',', '.' for viewing of the previous or next frames in the list; 'r' for the switching of transparency; 'f' for the switching of lightning; 'x' for hiding (closing) the window.

## 5.1 mglWnd class

This class is abstract class derived from mglGraph class (see Chapter 4 [MathGL core], page 157). It is defined in `#include <mgl2/wnd.h>` and provide base methods for handling window with MathGL graphics. Inherited classes are exist for QT and FLTK widget libraries: mglQT in `#include <mgl2/qt.h>`, mglFLTK in `#include <mgl2/fltk.h>`.

```

int Run () [Method on mglWnd]
int mgl_qt_run () [C function]
int mgl_fltk_run () [C function]
    Run main loop for event handling. Usually it should be called in a separate thread
    or as last function call in main().

void SetDrawFunc (int (*draw)(HMGL gr, void *p), void [Method on mglWnd]
    *par=NULL, void (*reload)(void *p)=NULL)
void SetDrawFunc (int (*draw)(mglGraph *gr)) [Method on mglWnd]
void SetDrawFunc (mglDraw *obj) [Method on mglWnd]
void mgl_wnd_set_func (HMGL gr, int (*draw)(HMGL gr, void *p), [C function]
    void *par, void (*reload)(void *p))
    Set callback functions for drawing (draw) and data reloading (reload), or instance
    obj of a class derived from mglDraw.

void SetClickFunc (void (*func)(HMGL gr, void *p)) [Method on mglWnd]
void mgl_set_click_func (void (*func)(HMGL gr, void *p)) [C function]
    Set callback function func which will be called on mouse click.

void SetMutex(pthread_mutex_t *mutex) [Method on mglWnd]
void mgl_wnd_set_mutex(HMGL gr, pthread_mutex_t *mutex) [C function]
    Set external mutex for lock/unlock external calculations by widget. This functions is
    called automatically at using Section 5.2 [mglDraw class], page 240.

void ToggleAlpha () [Method on mglWnd]
void mgl_wnd_toggle_alpha (HMGL gr) [C function]
    Switch on/off transparency but do not overwrite switches in user drawing function.

void ToggleLight () [Method on mglWnd]
void mgl_wnd_toggle_light (HMGL gr) [C function]
    Switch on/off lighting but do not overwrite switches in user drawing function.

void ToggleRotate () [Method on mglWnd]
void mgl_wnd_toggle_rotate (HMGL gr) [C function]
    Switch on/off rotation by mouse. Usually, left button is used for rotation, middle
    button for shift, right button for zoom/perspective.

void ToggleZoom () [Method on mglWnd]
void mgl_wnd_toggle_zoom (HMGL gr) [C function]
    Switch on/off zooming by mouse. Just select rectangular region by mouse and it will
    be zoomed in.

void ToggleNo () [Method on mglWnd]
void mgl_wnd_toggle_no (HMGL gr) [C function]
    Switch off all zooming and rotation and restore initial state.

void Update () [Method on mglWnd]
void mgl_wnd_update (HMGL gr) [C function]
    Update window contents. This is very useful function for manual updating the plot
    while long calculation was running in parallel thread.

```

```

void ReLoad () [Method on mglWnd]
void mgl_wnd_reload (HMGL gr) [C function]
    Reload user data and update picture. This function also update number of frames
    which drawing function can create.

void Adjust () [Method on mglWnd]
void mgl_wnd_adjust (HMGL gr) [C function]
    Adjust size of bitmap to window size.

void NextFrame () [Method on mglWnd]
void mgl_wnd_next_frame (HMGL gr) [C function]
    Show next frame if one.

void PrevFrame () [Method on mglWnd]
void mgl_wnd_prev_frame (HMGL gr) [C function]
    Show previous frame if one.

void Animation () [Method on mglWnd]
void mgl_wnd_animation (HMGL gr) [C function]
    Run/stop slideshow (animation) of frames.

void SetDelay (double dt) [Method on mglWnd]
void mgl_wnd_set_delay (HMGL gr, double dt) [C function]
    Sets delay for animation in seconds. Default value is 1 sec.

double GetDelay () [Method on mglWnd]
double mgl_wnd_get_delay (HMGL gr) [C function]
    Gets delay for animation in seconds.

void Setup (bool clfupd=true, bool showpos=false) [Method on mglWnd]
void mgl_setup_window (HMGL gr, bool clfupd, bool showpos) [C function]
    Enable/disable flags for:
    • clearing plot before Update();
    • showing the last mouse click position in the widget.

mglPoint LastMousePos () [Method on mglWnd]
void mgl_get_last_mouse_pos (HMGL gr, mreal *x, mreal *y, [C function]
    mreal *z)
    Gets last position of mouse click.

```

## 5.2 mglDraw class

This class provide base functionality for callback drawing and running calculation in separate thread. It is defined in `#include <mgl2/wnd.h>`. You should make inherited class and implement virtual functions if you need it.

```

int Draw (mglGraph *gr) [Virtual method on mglDraw]
    This is callback drawing function, which will be called when any redrawing is required
    for the window. There is support of a list of plots (frames). So as one can prepare a
    set of frames at first and redraw it fast later (but it requires more memory). Function
    should return positive number of frames for the list or zero if it will plot directly.

```

**void Reload ()** [Virtual method on `mglDraw`]  
This is callback function, which will be called if user press menu or toolbar to reload data.

**void Click ()** [Virtual method on `mglDraw`]  
This is callback function, which will be called if user click mouse.

**void Calc ()** [Virtual method on `mglDraw`]  
This is callback function, which will be called if user start calculations in separate thread by calling `mglDraw::Run()` function. It should periodically call `mglDraw::Check()` function to check if calculations should be paused.

**void Run ()** [Method on `mglDraw`]  
Runs `mglDraw::Calc()` function in separate thread. It also initialize `mglDraw::thr` variable and unlock `mglDraw::mutex`. Function is present only if FLTK support for widgets was enabled.

**void Cancel ()** [Method on `mglDraw`]  
Cancels thread with calculations. Function is present only if FLTK support for widgets was enabled.

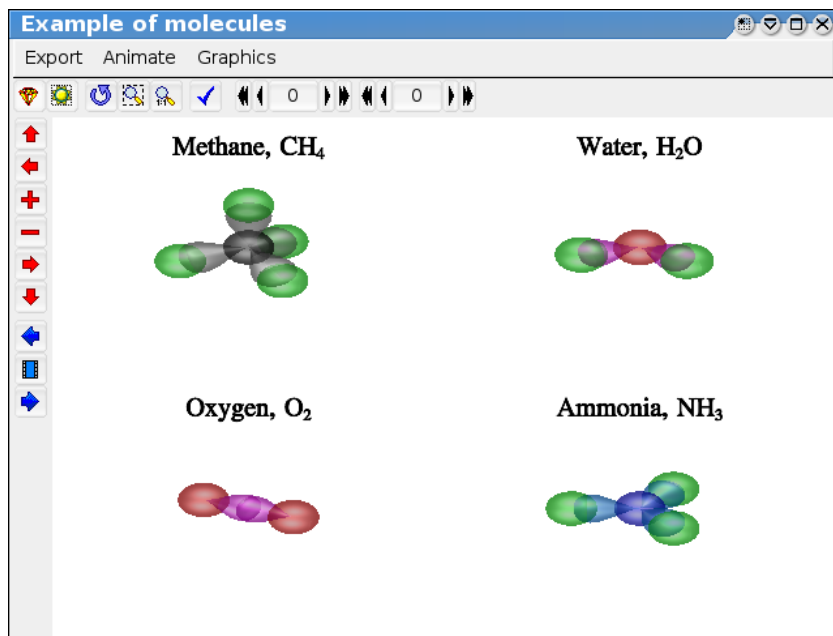
**void Pause ()** [Method on `mglDraw`]  
Pauses thread with calculations by locking `mglDraw::mutex`. You should call `mglDraw::Continue()` to continue calculations. Function is present only if FLTK support for widgets was enabled.

**void Continue ()** [Method on `mglDraw`]  
Continues calculations by unlocking `mglDraw::mutex`. Function is present only if FLTK support for widgets was enabled.

**void Continue ()** [Method on `mglDraw`]  
Checks if calculations should be paused and pause it. Function is present only if FLTK support for widgets was enabled.

### 5.3 Fl\_MathGL class

Class is FLTK widget which display MathGL graphics. It is defined in `#include <mgl2/fltk.h>`.



```
void set_draw (int (*draw)(HMGL gr, void *p)) [Method on Fl_MathGL]
void set_draw (int (*draw)(mglGraph *gr)) [Method on Fl_MathGL]
void set_draw (mglDraw *draw) [Method on Fl_MathGL]
```

Sets drawing function as global function or as one from a class `mglDraw`. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Parameter *par* contains pointer to data for the plotting function *draw*.

```
void update () [Method on Fl_MathGL]
    Update (redraw) plot.
```

```
void set_angle (mreal t, mreal p) [Method on Fl_MathGL]
    Set angles for additional plot rotation
```

```
void set_flag (int f) [Method on Fl_MathGL]
    Set bitwise flags for general state (1-Alpha, 2-Light)
```

```
void set_state (bool r, bool z) [Method on Fl_MathGL]
    Set flags for handling mouse: z=true allow zooming, r=true allow rotation/shifting/perspective and so on.
```

```
void set_zoom (mreal X1, mreal Y1, mreal X2, mreal Y2) [Method on Fl_MathGL]
    Set zoom in/out region
```

```
void get_zoom (mreal *X1, mreal *Y1, mreal *X2, mreal *Y2) [Method on Fl_MathGL]
    Get zoom in/out region
```

`void set_popup (const Fl_Menu_Item *pmenu,` [Method on Fl\_MathGL]  
                   `Fl_Widget *w, void *v)`  
 Set popup menu pointer

`void set_graph (HMGL gr)` [Method on Fl\_MathGL]  
`void set_graph (mg1Graph *gr)` [Method on Fl\_MathGL]  
 Set new grapher instead of built-in one. Note that Fl\_MathGL will automatically delete this object at destruction or at new `set_graph()` call.

`HMGL get_graph ()` [Method on Fl\_MathGL]  
 Get pointer to grapher.

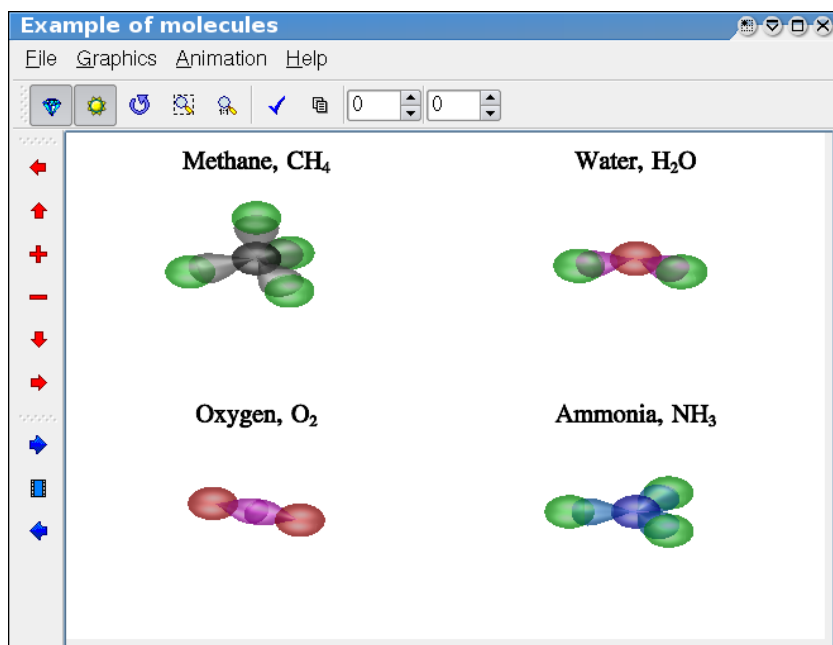
`void adjust ()` [Method on Fl\_MathGL]  
 Adjust image size to fit whole widget.

`Fl_Valuator * tet_val` [Fl\_MathGL option of Fl\_MathGL]  
 Pointer to external tet-angle validator.

`Fl_Valuator * phi_val` [Fl\_MathGL option of Fl\_MathGL]  
 Pointer to external phi-angle validator.

## 5.4 QMathGL class

Class is Qt widget which display MathGL graphics. It is defined in `#include <mgl2/qt.h>`.



`void setDraw (mg1Draw *dr)` [Method on QMathGL]  
 Sets drawing functions from a class inherited from `mg1Draw`.

**void setDraw** (int (\*draw)(mglBase \*gr, void \*p), void \*par=NULL) [Method on QMathGL]  
**void setDraw** (int (\*draw)(mglGraph \*gr)) [Method on QMathGL]  
 Sets the drawing function *draw*. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Parameter *par* contains pointer to data for the plotting function *draw*.

**void setGraph** (HMGL gr) [Method on QMathGL]  
**void setGraph** (mglGraph \*gr) [Method on QMathGL]  
 Set pointer to external grapher (instead of built-in one). Note that QMathGL will automatically delete this object at destruction or at new **setGraph()** call.

**HMGL getGraph** () [Method on QMathGL]  
 Get pointer to grapher.

**void setPopup** (QMenu \*p) [Method on QMathGL]  
 Set popup menu pointer.

**void setSize** (int w, int h) [Method on QMathGL]  
 Set widget/picture sizes

**double getRatio** () [Method on QMathGL]  
 Return aspect ratio of the picture.

**int getPer** () [Method on QMathGL]  
 Get perspective value in percents.

**int getPhi** () [Method on QMathGL]  
 Get Phi-angle value in degrees.

**int getTet** () [Method on QMathGL]  
 Get Theta-angle value in degrees.

**bool getAlpha** () [Method on QMathGL]  
 Get transparency state.

**bool getLight** () [Method on QMathGL]  
 Get lightning state.

**bool getZoom** () [Method on QMathGL]  
 Get mouse zooming state.

**bool getRotate** () [Method on QMathGL]  
 Get mouse rotation state.

**void refresh** () [Slot on QMathGL]  
 Redraw saved bitmap without executing drawing function.

**void update** () [Slot on QMathGL]  
 Update picture by executing drawing function.



<code>void copy ()</code> Copy graphics to clipboard.	[Slot on QMathGL]
<code>void copyClickCoor ()</code> Copy coordinates of click (as text).	[Slot on QMathGL]
<code>void print ()</code> Print current picture.	[Slot on QMathGL]
<code>void stop ()</code> Send signal to stop drawing.	[Slot on QMathGL]
<code>void adjust ()</code> Adjust image size to fit whole widget.	[Slot on QMathGL]
<code>void nextSlide ()</code> Show next slide.	[Slot on QMathGL]
<code>void prevSlide ()</code> Show previous slide.	[Slot on QMathGL]
<code>void animation (bool st=true)</code> Start/stop animation.	[Slot on QMathGL]
<code>void setPer (int val)</code> Set perspective value.	[Slot on QMathGL]
<code>void setPhi (int val)</code> Set Phi-angle value.	[Slot on QMathGL]
<code>void setTet (int val)</code> Set Theta-angle value.	[Slot on QMathGL]
<code>void setAlpha (bool val)</code> Switch on/off transparency.	[Slot on QMathGL]
<code>void setLight (bool val)</code> Switch on/off lightning.	[Slot on QMathGL]
<code>void setGrid (bool val)</code> Switch on/off drawing of grid for absolute coordinates.	[Slot on QMathGL]
<code>void setZoom (bool val)</code> Switch on/off mouse zooming.	[Slot on QMathGL]
<code>void setRotate (bool val)</code> Switch on/off mouse rotation.	[Slot on QMathGL]
<code>void zoomIn ()</code> Zoom in graphics.	[Slot on QMathGL]
<code>void zoomOut ()</code> Zoom out graphics.	[Slot on QMathGL]

<code>void shiftLeft ()</code> Shift graphics to left direction.	[Slot on QMathGL]
<code>void shiftRight ()</code> Shift graphics to right direction.	[Slot on QMathGL]
<code>void shiftUp ()</code> Shift graphics to up direction.	[Slot on QMathGL]
<code>void shiftDown ()</code> Shift graphics to down direction.	[Slot on QMathGL]
<code>void restore ()</code> Restore zoom and rotation to default values.	[Slot on QMathGL]
<code>void exportPNG (QString fname="")</code> Export current picture to PNG file.	[Slot on QMathGL]
<code>void exportPNGs (QString fname="")</code> Export current picture to PNG file (no transparency).	[Slot on QMathGL]
<code>void exportJPG (QString fname="")</code> Export current picture to JPEG file.	[Slot on QMathGL]
<code>void exportBPS (QString fname="")</code> Export current picture to bitmap EPS file.	[Slot on QMathGL]
<code>void exportEPS (QString fname="")</code> Export current picture to vector EPS file.	[Slot on QMathGL]
<code>void exportSVG (QString fname="")</code> Export current picture to SVG file.	[Slot on QMathGL]
<code>void exportGIF (QString fname="")</code> Export current picture to GIF file.	[Slot on QMathGL]
<code>void exportTEX (QString fname="")</code> Export current picture to LaTeX/Tikz file.	[Slot on QMathGL]
<code>void exportTGA (QString fname="")</code> Export current picture to TGA file.	[Slot on QMathGL]
<code>void exportXYZ (QString fname="")</code> Export current picture to XYZ/XYZL/XYZF file.	[Slot on QMathGL]
<code>void exportOBJ (QString fname="")</code> Export current picture to OBJ/MTL file.	[Slot on QMathGL]
<code>void exportSTL (QString fname="")</code> Export current picture to STL file.	[Slot on QMathGL]
<code>void exportOFF (QString fname="")</code> Export current picture to OFF file.	[Slot on QMathGL]

<b>void</b> <b>setUsePrimitives</b> ( <b>bool</b> <i>use</i> )	[Slot on QMathGL]
Enable using list of primitives for frames. This allows frames transformation/zoom but requires much more memory. Default value is <b>true</b> .	
<b>void</b> <b>setMGLFont</b> ( <b>QString</b> <i>path</i> )	[Slot on QMathGL]
Restore ( <i>path=""</i> ) or load font for graphics.	
<b>void</b> <b>about</b> ()	[Slot on QMathGL]
Show about information.	
<b>void</b> <b>aboutQt</b> ()	[Slot on QMathGL]
Show information about Qt version.	
<b>void</b> <b>phiChanged</b> ( <b>int</b> <i>val</i> )	[Signal on QMathGL]
Phi angle changed (by mouse or by toolbar).	
<b>void</b> <b>tetChanged</b> ( <b>int</b> <i>val</i> )	[Signal on QMathGL]
Tet angle changed (by mouse or by toolbar).	
<b>void</b> <b>perChanged</b> ( <b>int</b> <i>val</i> )	[Signal on QMathGL]
Perspective changed (by mouse or by toolbar).	
<b>void</b> <b>alphaChanged</b> ( <b>bool</b> <i>val</i> )	[Signal on QMathGL]
Transparency changed (by toolbar).	
<b>void</b> <b>lightChanged</b> ( <b>bool</b> <i>val</i> )	[Signal on QMathGL]
Lighting changed (by toolbar).	
<b>void</b> <b>gridChanged</b> ( <b>bool</b> <i>val</i> )	[Signal on QMathGL]
Grid drawing changed (by toolbar).	
<b>void</b> <b>zoomChanged</b> ( <b>bool</b> <i>val</i> )	[Signal on QMathGL]
Zooming changed (by toolbar).	
<b>void</b> <b>rotateChanged</b> ( <b>bool</b> <i>val</i> )	[Signal on QMathGL]
Rotation changed (by toolbar).	
<b>void</b> <b>mouseClick</b> ( <b>mreal</b> <i>x</i> , <b>mreal</b> <i>y</i> , <b>mreal</b> <i>z</i> )	[Signal on QMathGL]
Mouse click take place at position { <i>x,y,z</i> }.	
<b>void</b> <b>frameChanged</b> ( <b>int</b> <i>val</i> )	[Signal on QMathGL]
Need another frame to show.	
<b>void</b> <b>showWarn</b> ( <b>QString</b> <i>warn</i> )	[Signal on QMathGL]
Need to show warning.	
<b>void</b> <b>posChanged</b> ( <b>QString</b> <i>pos</i> )	[Signal on QMathGL]
Position of mouse click is changed.	
<b>void</b> <b>objChanged</b> ( <b>int</b> <i>id</i> )	[Signal on QMathGL]
Object id is changed (due to mouse click).	

**void refreshData ()** [Signal on QMathGL]  
 Data can be changed (drawing is finished).

**QString appName** [QMathGL option of QMathGL]  
 Application name for message boxes.

**bool autoResize** [QMathGL option of QMathGL]  
 Allow auto resizing (default is false).

## 5.5 wxMathGL class

Class is WX widget which display MathGL graphics. It is defined in `#include <mgl2/wx.h>`.

**void SetDraw (mglDraw \*dr)** [Method on wxMathGL]  
 Sets drawing functions from a class inherited from mglDraw.

**void SetDraw (int (\*draw)(mglBase \*gr, void \*p), void \*par=NULL)** [Method on wxMathGL]

**void SetDraw (int (\*draw)(mglGraph \*gr))** [Method on wxMathGL]  
 Sets the drawing function *draw*. There is support of a list of plots (frames). So as one can prepare a set of frames at first and redraw it fast later (but it requires more memory). Function should return positive number of frames for the list or zero if it will plot directly. Parameter *par* contains pointer to data for the plotting function *draw*.

**void SetGraph (HMGL gr)** [Method on wxMathGL]

**void SetGraph (mglGraph \*gr)** [Method on wxMathGL]  
 Set pointer to external grapher (instead of built-in one). Note that wxMathGL will automatically delete this object at destruction or at new `setGraph()` call.

**HMGL GetGraph ()** [Method on wxMathGL]  
 Get pointer to grapher.

**void SetPopup (wxMenu \*p)** [Method on wxMathGL]  
 Set popup menu pointer.

**void SetSize (int w, int h)** [Method on wxMathGL]  
 Set widget/picture sizes

**double GetRatio ()** [Method on wxMathGL]  
 Return aspect ratio of the picture.

**int GetPer ()** [Method on wxMathGL]  
 Get perspective value in percents.

**int GetPhi ()** [Method on wxMathGL]  
 Get Phi-angle value in degrees.

**int GetTet ()** [Method on wxMathGL]  
 Get Theta-angle value in degrees.

<b>bool GetAlpha ()</b> Get transparency state.	[Method on wxMathGL]
<b>bool GetLight ()</b> Get lightning state.	[Method on wxMathGL]
<b>bool GetZoom ()</b> Get mouse zooming state.	[Method on wxMathGL]
<b>bool GetRotate ()</b> Get mouse rotation state.	[Method on wxMathGL]
<b>void Repaint ()</b> Redraw saved bitmap without executing drawing function.	[Method on wxMathGL]
<b>void Update ()</b> Update picture by executing drawing function.	[Method on wxMathGL]
<b>void Copy ()</b> Copy graphics to clipboard.	[Method on wxMathGL]
<b>void Print ()</b> Print current picture.	[Method on wxMathGL]
<b>void Adjust ()</b> Adjust image size to fit whole widget.	[Method on wxMathGL]
<b>void NextSlide ()</b> Show next slide.	[Method on wxMathGL]
<b>void PrevSlide ()</b> Show previous slide.	[Method on wxMathGL]
<b>void Animation (bool st=true)</b> Start/stop animation.	[Method on wxMathGL]
<b>void SetPer (int val)</b> Set perspective value.	[Method on wxMathGL]
<b>void SetPhi (int val)</b> Set Phi-angle value.	[Method on wxMathGL]
<b>void SetTet (int val)</b> Set Theta-angle value.	[Method on wxMathGL]
<b>void SetAlpha (bool val)</b> Switch on/off transparency.	[Method on wxMathGL]
<b>void SetLight (bool val)</b> Switch on/off lightning.	[Method on wxMathGL]
<b>void SetZoom (bool val)</b> Switch on/off mouse zooming.	[Method on wxMathGL]

<code>void SetRotate (bool val)</code> Switch on/off mouse rotation.	[Method on wxMathGL]
<code>void ZoomIn ()</code> Zoom in graphics.	[Method on wxMathGL]
<code>void ZoomOut ()</code> Zoom out graphics.	[Method on wxMathGL]
<code>void ShiftLeft ()</code> Shift graphics to left direction.	[Method on wxMathGL]
<code>void ShiftRight ()</code> Shift graphics to right direction.	[Method on wxMathGL]
<code>void ShiftUp ()</code> Shift graphics to up direction.	[Method on wxMathGL]
<code>void ShiftDown ()</code> Shift graphics to down direction.	[Method on wxMathGL]
<code>void Restore ()</code> Restore zoom and rotation to default values.	[Method on wxMathGL]
<code>void About ()</code> Show about information.	[Method on wxMathGL]
<code>void ExportPNG (QString fname="")</code> Export current picture to PNG file.	[Method on wxMathGL]
<code>void ExportPNGs (QString fname="")</code> Export current picture to PNG file (no transparency).	[Method on wxMathGL]
<code>void ExportJPG (QString fname="")</code> Export current picture to JPEG file.	[Method on wxMathGL]
<code>void ExportBPS (QString fname="")</code> Export current picture to bitmap EPS file.	[Method on wxMathGL]
<code>void ExportEPS (QString fname="")</code> Export current picture to vector EPS file.	[Method on wxMathGL]
<code>void ExportSVG (QString fname="")</code> Export current picture to SVG file.	[Method on wxMathGL]

## 6 Data processing

This chapter describe classes `mglData` and `mglDataC` for working with data arrays of real and complex numbers. Both classes are derived from abstract class `mglDataA`, and can be used as arguments of any plotting functions (see Chapter 4 [MathGL core], page 157). These classes are defined in `#include <mgl2/data.h>` and `#include <mgl2/datac.h>` correspondingly. The classes have mostly the same set of functions for easy and safe allocation, resizing, loading, saving, modifying of data arrays. Also it can numerically differentiate and integrate data, interpolate, fill data by formula and so on. Classes support data with dimensions up to 3 (like function of 3 variables – x,y,z). The internal representation of numbers is `mreal` (or `dual=std::complex<mreal>` for `mglDataC`), which can be configured as float or double by selecting option `--enable-double` at the MathGL configuring (see Section 1.3 [Installation], page 2). Float type have smaller size in memory and usually it has enough precision in plotting purposes. However, double type provide high accuracy what can be important for time-axis, for example. Data arrays are denoted by Small Caps (like `DAT`) if it can be (re-)created by MGL commands.

### 6.1 Public variables

<code>mreal * a</code>	[Variable of <code>mglData</code> ]
<code>dual * a</code>	[Variable of <code>mglDataC</code> ]
Data array itself. The flat data representation is used. For example, matrix [nx x ny] is presented as flat (1d-) array with length nx*ny. The element with indexes {i, j, k} is a[i+nx*j+nx*ny*k] (indexes are zero based).	
<code>long nx</code>	[Variable of <code>mglData</code> ]
<code>long nx</code>	[Variable of <code>mglDataC</code> ]
Number of points in 1st dimensions ('x' dimension).	
<code>long ny</code>	[Variable of <code>mglData</code> ]
<code>long ny</code>	[Variable of <code>mglDataC</code> ]
Number of points in 2nd dimensions ('y' dimension).	
<code>long nz</code>	[Variable of <code>mglData</code> ]
<code>long nz</code>	[Variable of <code>mglDataC</code> ]
Number of points in 3d dimensions ('z' dimension).	
<code>std::string id</code>	[Variable of <code>mglData</code> ]
<code>std::string id</code>	[Variable of <code>mglDataC</code> ]
Names of column (or slice if nz>1) – one character per column.	
<code>bool link</code>	[Variable of <code>mglData</code> ]
<code>bool link</code>	[Variable of <code>mglDataC</code> ]
Flag to use external data, i.e. don't delete it.	
<code>std::wstring s</code>	[Variable of <code>mglDataA</code> ]
Name of data. It is used in parsing of MGL scripts.	
<code>bool temp</code>	[Variable of <code>mglDataA</code> ]
Flag of temporary variable, which should be deleted.	

`void (*)(void *) func` [Variable of `mglDataA`]  
 Pointer to callback function which will be called at destroying.

`void * o` [Variable of `mglDataA`]  
 Pointer to object for callback function.

`mreal GetVal (long i)` [Method on `mglData`]  
`mreal GetVal (long i)` [Method on `mglDataC`]  
`void SetVal (mreal val, long i)` [Method on `mglData`]  
`void SetVal (mreal val, long i)` [Method on `mglDataC`]  
 Gets or sets the value in by "flat" index *i* without border checking. Index *i* should be in range  $[0, nx*ny*nz-1]$ .

`long GetNx ()` [Method on `mglDataA`]  
`long GetNy ()` [Method on `mglDataA`]  
`long GetNz ()` [Method on `mglDataA`]  
`long mgl_data_get_nx (HCDT dat)` [C function]  
`long mgl_data_get_ny (HCDT dat)` [C function]  
`long mgl_data_get_nz (HCDT dat)` [C function]  
 Gets the x-, y-, z-size of the data.

`mreal mgl_data_get_value (HCDT dat, int i, int j, int k)` [C function]  
`dual mgl_datac_get_value (HCDT dat, int i, int j, int k)` [C function]  
`mreal * mgl_data_value (HMDT dat, int i, int j, int k)` [C function]  
`dual * mgl_datac_value (HADT dat, int i, int j, int k)` [C function]  
`void mgl_data_set_value (HMDT dat, mreal v, int i, int j, int k)` [C function]  
`void mgl_datac_set_value (HADT dat, dual v, int i, int j, int k)` [C function]  
 Gets or sets the value in specified cell of the data with border checking.

`const mreal * mgl_data_data (HCDT dat)` [C function]  
`const dual * mgl_datac_data (HCDT dat)` [C function]  
 Returns pointer to internal data array.

`void mgl_data_set_func (mglDataA *dat, void (*func)(void *), void *par)` [C function only]  
 Set pointer to callback function which will be called at destroying.

`void mgl_data_set_name (mglDataA *dat, const char *name)` [C function]  
`void mgl_data_set_name_w (mglDataA *dat, const wchar_t *name)` [C function]  
 Set name of data, which used in parsing of MGL scripts.

## 6.2 Data constructor

`new DAT [nx=1 'eq']` [MGL command]  
`new DAT nx ny ['eq']` [MGL command]  
`new DAT nx ny nz ['eq']` [MGL command]  
`mglData (int mx=1, int my=1, int mz=1)` [Constructor on `mglData`]  
`mglDataC (int mx=1, int my=1, int mz=1)` [Constructor on `mglDataC`]



```

HMDT mgl_create_data () [C function]
HMDT mgl_create_data_size (int mx, int my, int mz) [C function]
HADT mgl_create_datac () [C function]
HADT mgl_create_datac_size (int mx, int my, int mz) [C function]

```

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 258.

```

copy DAT dat2 ['eq']=" [MGL command]
copy DAT val [MGL command]
mglData (const mglDataA &dat2) [Constructor on mglData]
mglData (const mglDataA *dat2) [Constructor on mglData]
mglData (int size, const float *dat2) [Constructor on mglData]
mglData (int size, int cols, const float *dat2) [Constructor on mglData]
mglData (int size, const double *dat2) [Constructor on mglData]
mglData (int size, int cols, const double *dat2) [Constructor on mglData]
mglData (const double *dat2, int size) [Constructor on mglData]
mglData (const double *dat2, int size, int cols) [Constructor on mglData]
mglDataC (const mglDataA &dat2) [Constructor on mglDataC]
mglDataC (const mglDataA *dat2) [Constructor on mglDataC]
mglDataC (int size, const float *dat2) [Constructor on mglDataC]
mglDataC (int size, int cols, const float *dat2) [Constructor on mglDataC]
mglDataC (int size, const double *dat2) [Constructor on mglDataC]
mglDataC (int size, int cols, const double *dat2) [Constructor on mglDataC]
mglDataC (int size, const dual *dat2) [Constructor on mglDataC]
mglDataC (int size, int cols, const dual *dat2) [Constructor on mglDataC]

```

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 258.

```

read DAT 'fname' [MGL command]
mglData (const char *fname) [Constructor on mglData]
mglDataC (const char *fname) [Constructor on mglDataC]
HMDT mgl_create_data_file (const char *fname) [C function]
HADT mgl_create_datac_file (const char *fname) [C function]

```

Reads data from tab-separated text file with auto determining sizes of the data.

```

delete dat [MGL command]
~mglData () [Destructor on mglData]
void mgl_delete_data (HMDT dat) [C function]
~mglDataC () [Destructor on mglDataC]
void mgl_delete_datac (HADT dat) [C function]

```

Deletes the instance of class *mglData*.

### 6.3 Data resizing

```

new DAT [nx=1 ny=1 nz=1] [MGL command]
void Create (int mx, int my=1, int mz=1) [Method on mglData]

```

```
void Create (int mx, int my=1, int mz=1) [Method on mglDataC]
void mgl_data_create (HMDT dat, int mx, int my, int mz) [C function]
void mgl_datac_create (HADT dat, int mx, int my, int mz) [C function]
```

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]
void Rearrange (int mx, int my=0, int mz=0) [Method on mglData]
void Rearrange (int mx, int my=0, int mz=0) [Method on mglDataC]
void mgl_data_rearrange (HMDT dat, int mx, int my, int mz) [C function]
void mgl_datac_rearrange (HADT dat, int mx, int my, int mz) [C function]
```

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]
void Transpose (const char *dim="yx") [Method on mglData]
void Transpose (const char *dim="yx") [Method on mglDataC]
void mgl_data_transpose (HMDT dat, const char *dim) [C function]
void mgl_datac_transpose (HADT dat, const char *dim) [C function]
```

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]
void Extend (int n1, int n2=0) [Method on mglData]
void Extend (int n1, int n2=0) [Method on mglDataC]
void mgl_data_extend (HMDT dat, int n1, int n2) [C function]
void mgl_datac_extend (HADT dat, int n1, int n2) [C function]
```

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  $j=0\dots n1$ . Correspondingly, for  $n1<0$  new array will be  $a_{ij}^{new} = a_j^{old}$  where  $i=0\dots |n1|$ .

```
squeeze dat rx [ry=1 rz=1 sm=off] [MGL command]
void Squeeze (int rx, int ry=1, int rz=1, bool [Method on mglData]
    smooth=false)
void Squeeze (int rx, int ry=1, int rz=1, bool [Method on mglDataC]
    smooth=false)
void mgl_data_squeeze (HMDT dat, int rx, int ry, int rz, int [C function]
    smooth)
void mgl_datac_squeeze (HADT dat, int rx, int ry, int rz, int [C function]
    smooth)
```

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]
void Crop (int n1, int n2, char dir='x') [Method on mglData]
void Crop (int n1, int n2, char dir='x') [Method on mglDataC]
void mgl_data_crop (HMDT dat, int n1, int n2, char dir) [C function]
void mgl_datac_crop (HADT dat, int n1, int n2, char dir) [C function]
    Cuts off edges of the data  $i < n1$  and  $i > n2$  if  $n2 > 0$  or  $i > n[xyz] - n2$  if  $n2 \leq 0$  along
    direction dir.
```

```
insert dat 'dir' [pos=off num=0] [MGL command]
void Insert (char dir, int pos=0, int num=1) [Method on mglData]
void Insert (char dir, int pos=0, int num=1) [Method on mglDataC]
void mgl_data_insert (HMDT dat, char dir, int pos, char num) [C function]
void mgl_datac_insert (HADT dat, char dir, int pos, char num) [C function]
    Insert num slices along dir-direction at position pos and fill it by zeros.
```

```
delete dat 'dir' [pos=off num=0] [MGL command]
void Delete (char dir, int pos=0, int num=1) [Method on mglData]
void Delete (char dir, int pos=0, int num=1) [Method on mglDataC]
void mgl_data_delete (HMDT dat, char dir, int pos, char num) [C function]
void mgl_datac_delete (HADT dat, char dir, int pos, char num) [C function]
    Delete num slices along dir-direction at position pos.
```

```
sort dat idx [idy=-1] [MGL command]
void Sort (lond idx, long idy=-1) [Method on mglData]
void mgl_data_sort (HMDT dat, lond idx, long idy) [C function]
    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]
void Clean (lond idx) [Method on mglData]
void mgl_data_clean (HMDT dat, lond idx) [C function]
    Delete rows which values are equal to next row for given column idx.
```

```
join dat vdat [v2dat ...] [MGL command]
void Join (const mglDataA &vdat) [Method on mglData]
void Join (const mglDataA &vdat) [Method on mglDataC]
void mgl_data_join (HMDT dat, HCDT vdat) [C function]
void mgl_datac_join (HADT dat, HCDT vdat) [C function]
    Join data cells from vdat to dat. At this, function increase dat sizes according follow-
    ing: 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.
```

## 6.4 Data filling

```
list DAT v1 ... [MGL command]
    Creates new variable with name dat and fills it by numeric values of command ar-
    guments 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.

`void Set (const float *A, int NX, int NY=1, int NZ=1)` [Method on `mglData`]

`void Set (const double *A, int NX, int NY=1, int NZ=1)` [Method on `mglData`]

`void mgl_data_set_float (HMDT dat, const mreal *A, int NX, int NY, int NZ)` [C function]

`void mgl_data_set_double (HMDT dat, const double *A, int NX, int NY, int NZ)` [C function]

`void Set (const float *A, int NX, int NY=1, int NZ=1)` [Method on `mglDataC`]

`void Set (const double *A, int NX, int NY=1, int NZ=1)` [Method on `mglDataC`]

`void Set (const dual *A, int NX, int NY=1, int NZ=1)` [Method on `mglDataC`]

`void mgl_datac_set_float (HADT dat, const mreal *A, int NX, int NY, int NZ)` [C function]

`void mgl_datac_set_double (HADT dat, const double *A, int NX, int NY, int NZ)` [C function]

`void mgl_datac_set_complex (HADT dat, const dual *A, int NX, int NY, int NZ)` [C function]

Allocates memory and copies the data from the `flat float*` or `double*` array.

`void Set (const float **A, int N1, int N2)` [Method on `mglData`]

`void Set (const double **A, int N1, int N2)` [Method on `mglData`]

`void mgl_data_set_mreal2 (HMDT dat, const mreal **A, int N1, int N2)` [C function]

`void mgl_data_set_double2 (HMDT dat, const double **A, int N1, int N2)` [C function]

Allocates memory and copies the data from the `float**` or `double**` array with dimensions *N1, N2*, i.e. from array defined as `mreal a[N1][N2];`.

`void Set (const float ***A, int N1, int N2)` [Method on `mglData`]

`void Set (const double ***A, int N1, int N2)` [Method on `mglData`]

`void mgl_data_set_mreal3 (HMDT dat, const mreal ***A, int N1, int N2)` [C function]

```
void mgl_data_set_double3 (HMDT dat, const double ***A, int      [C function]
                        N1, int N2)
```

Allocates memory and copies the data from the `float***` or `double***` array with dimensions *N1*, *N2*, *N3*, i.e. from array defined as `mreal a[N1][N2][N3]` ;.

```
void Set (gsl_vector *v)                                     [Method on mglData]
```

```
void Set (gsl_vector *v)                                     [Method on mglDataC]
```

```
void mgl_data_set_vector (HMDT dat, gsl_vector *v)          [C function]
```

```
void mgl_datac_set_vector (HMDT dat, gsl_vector *v)          [C function]
```

Allocates memory and copies the data from the `gsl_vector *` structure.

```
void Set (gsl_matrix *m)                                     [Method on mglData]
```

```
void Set (gsl_matrix *m)                                     [Method on mglDataC]
```

```
void mgl_data_set_matrix (HMDT dat, gsl_matrix *m)          [C function]
```

```
void mgl_datac_set_matrix (HMDT dat, gsl_matrix *m)          [C function]
```

Allocates memory and copies the data from the `gsl_matrix *` structure.

```
void Set (const mglDataA &from)                               [Method on mglData]
```

```
void Set (HCDT from)                                          [Method on mglData]
```

```
void mgl_data_set (HMDT dat, HCDT from)                       [C function]
```

```
void Set (const mglDataA &from)                               [Method on mglDataC]
```

```
void Set (HCDT from)                                          [Method on mglDataC]
```

```
void mgl_datac_set (HMDT dat, HCDT from)                       [C function]
```

Copies the data from `mglData` (or `mglDataA`) instance *from*.

```
void Set (const mglDataA &re, const mglDataA &im)           [Method on mglDataC]
```

```
void Set (HCDT re, HCDT im)                                   [Method on mglDataC]
```

```
void SetAmpl (HCDT ampl, const mglDataA &phase)             [Method on mglDataC]
```

```
void mgl_datac_set_ri (HMDT dat, HCDT re, HCDT im)          [C function]
```

```
void mgl_datac_set_ap (HMDT dat, HCDT ampl, HCDT phase)      [C function]
```

Copies the data from `mglData` instances for real and imaginary parts of complex data arrays.

```
void Set (const std::vector<int> &d)                         [Method on mglData]
```

```
void Set (const std::vector<int> &d)                         [Method on mglDataC]
```

```
void Set (const std::vector<float> &d)                       [Method on mglData]
```

```
void Set (const std::vector<float> &d)                       [Method on mglDataC]
```

```
void Set (const std::vector<double> &d)                     [Method on mglData]
```

```
void Set (const std::vector<double> &d)                     [Method on mglDataC]
```

```
void Set (const std::vector<dual> &d)                       [Method on mglDataC]
```

Allocates memory and copies the data from the `std::vector<T>` array.

```
void Set (const char *str, int NX, int NY=1, int           [Method on mglData]
        NZ=1)
```

```
void mgl_data_set_values (const char *str, int NX, int NY,   [C function]
        int NZ)
```

```
void Set (const char *str, int NX, int NY=1, int           [Method on mglDataC]
        NZ=1)
```

Allocates memory and scanf the data from the string.

Allocate memory and set data from variable argument list of *double* values. Note, you need to specify decimal point ‘.’ for integer values! For example, the code `SetList(2,0.,1.);` is correct, but the code `SetList(2,0,1);` is incorrect.

Links external data array, i.e. don't delete this array at exit.

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.

Equidistantly fills the data values to range  $[v1, v2]$  in direction  $dir=\{\text{'x'}, 'y', 'z'}\}$ .

<i>dat 'eq'</i>	[MGL command]
<i>dat 'eq' vdat</i>	[MGL command]
<i>dat 'eq' vdat wdat</i>	[MGL command]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const char <i>*opt=""</i> )	[Method on mglData]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const mglDataA <i>&amp;vdat</i> , const char <i>*opt=""</i> )	[Method on mglData]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const mglDataA <i>&amp;vdat</i> , const mglDataA <i>&amp;wdat</i> , const char <i>*opt=""</i> )	[Method on mglData]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const char <i>*opt=""</i> )	[Method on mglDataC]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const mglDataA <i>&amp;vdat</i> , const char <i>*opt=""</i> )	[Method on mglDataC]
Fill (HMGL <i>gr</i> , const char <i>*eq</i> , const mglDataA <i>&amp;vdat</i> , const mglDataA <i>&amp;wdat</i> , const char <i>*opt=""</i> )	[Method on mglDataC]
mgl_data_fill_eq (HMGL <i>gr</i> , HMDT <i>dat</i> , const char <i>*eq</i> , HCDT <i>vdat</i> , HCDT <i>wdat</i> , const char <i>*opt</i> )	[C function]

```
void mgl_datac_fill_eq (HMGL gr, HADT dat, const char *eq,          [C function]
                      HCDT vdat, HCDT wdat, const char *opt)
```

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]
```

```
void Modify (const char *eq, int dim=0)                         [Method on mglData]
```

```
void Modify (const char *eq, const mglDataA &v)                [Method on mglData]
```

```
void Modify (const char *eq, const mglDataA &v, const          [Method on mglData]
             mglDataA &w)
```

```
void Modify (const char *eq, int dim=0)                         [Method on mglDataC]
```

```
void Modify (const char *eq, const mglDataA &v)                [Method on mglDataC]
```

```
void Modify (const char *eq, const mglDataA &v,                [Method on mglDataC]
             const mglDataA &w)
```

```
void mgl_data_modify (HMDT dat, const char *eq, int dim)       [C function]
```

```
void mgl_data_modify_vw (HMDT dat, const char *eq, HCDT v,     [C function]
                        HCDT w)
```

```
void mgl_datac_modify (HADT dat, const char *eq, int dim)      [C function]
```

```
void mgl_datac_modify_vw (HADT dat, const char *eq, HCDT v,    [C function]
                        HCDT w)
```

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]
```

```
void FillSample (const char *how)                               [Method on mglData]
```

```
void mgl_data_fill_sample (HMDT a, const char *how)             [C function]
```

Fills data by 'x' or 'k' samples for Hankel ('h') or Fourier ('f') transform.

```
datagrid dat xdat ydat zdat                                    [MGL command]
```

```
mglData Grid (HMGL gr, const mglDataA &x, const                [Method on mglData]
              mglDataA &y, const mglDataA &z, const char *opt="")
```

```
mglData Grid (const mglDataA &x, const mglDataA &y,            [Method on mglData]
              const mglDataA &z, mglPoint p1, mglPoint p2)
```

```
void mgl_data_grid (HMGL gr, HMDT u, HCDT x, HCDT y, HCDT z,   [C function]
                   const char *opt)
```

```
void mgl_data_grid_xy (HMDT u, HCDT x, HCDT y, HCDT z, mreal   [C function]
                      x1, mreal x2, mreal y1, mreal y2)
```

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 2.9.11 [Making regular data], page 127, for sample code and picture.

```

put dat val [i=: j=: k=:] [MGL command]
void Put (mreal val, int i=-1, int j=-1, int k=-1) [Method on mglData]
void Put (dual val, int i=-1, int j=-1, int k=-1) [Method on mglDataC]
void mgl_data_put_val (HMDT a, mreal val, int i, int j, int k) [C function]
void mgl_datac_put_val (HADT a, dual val, int i, int j, int k) [C function]
    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]
void Put (const mglDataA &v, int i=-1, int j=-1, int k=-1) [Method on mglData]
void Put (const mglDataA &v, int i=-1, int j=-1, int k=-1) [Method on mglDataC]
void mgl_data_put_dat (HMDT a, HCDT v, int i, int j, int k) [C function]
void mgl_datac_put_dat (HADT a, HCDT v, int i, int j, int k) [C function]
    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]
void Refill (const mglDataA &x, const mglDataA &v, mreal x1, mreal x2, long sl=-1) [Method on mglData]
void Refill (const mglDataA &x, const mglDataA &v, mglPoint p1, mglPoint p2, long sl=-1) [Method on mglData]
void Refill (const mglDataA &x, const mglDataA &y, const mglDataA &v, mglPoint p1, mglPoint p2, long sl=-1) [Method on mglData]
void Refill (const mglDataA &x, const mglDataA &y, const mglDataA &z, const mglDataA &v, mglPoint p1, mglPoint p2) [Method on mglData]
void Refill (HMGL gr, const mglDataA &x, const mglDataA &v, long sl=-1, const char *opt="") [Method on mglData]
void Refill (HMGL gr, const mglDataA &x, const mglDataA &y, const mglDataA &v, long sl=-1, const char *opt="") [Method on mglData]
void Refill (HMGL gr, const mglDataA &x, const mglDataA &y, const mglDataA &z, const mglDataA &v, const char *opt="") [Method on mglData]
void mgl_data_refill_x (HMDT a, HCDT x, HCDT v, mreal x1, mreal x2, long sl) [C function]
void mgl_data_refill_xy (HMDT a, HCDT x, HCDT y, HCDT v, mreal x1, mreal x2, mreal y1, mreal y2, long sl) [C function]
void mgl_data_refill_xyz (HMDT a, HCDT x, HCDT y, HCDT z, HCDT v, mreal x1, mreal x2, mreal y1, mreal y2, mreal z1, mreal z2) [C function]

```



```
void mgl_data_refill_gr (HMGL gr, HMDT a, HCDT x, HCDT y, HCDT z, HCDT v, long sl, const char *opt) [C function]
    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]
void RefillGS (const mglDataA &x, const mglDataA &v, [Method on mglData]
    mreal x1, mreal x2, long sl=-1)
void mgl_data_refill_gs (HMDT a, HCDT x, HCDT v, mreal x1, [C function]
    mreal x2, long sl)
    Fills by global cubic spline values of array v at the point x=X[i], where X are equidis-
    tantly 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]
void SetColumnId (const char *ids) [Method on mglData]
void SetColumnId (const char *ids) [Method on mglDataC]
void mgl_data_set_id (HMDT a, const char *ids) [C function]
void mgl_datac_set_id (HADT a, const char *ids) [C function]
    Sets the symbol ids for data columns. The string should contain one symbol 'a'...'z'
    per column. These ids are used in [column], page 264.
```

## 6.5 File I/O

```
read DAT 'fname' [MGL command]
bool Read (const char *fname) [Method on mglData]
bool Read (const char *fname) [Method on mglDataC]
int mgl_data_read (HMDT dat, const char *fname) [C function]
int mgl_datac_read (HADT dat, const char *fname) [C function]
    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]
bool Read (const char *fname, int mx, int my=1, int [Method on mglData]
    mz=1)
bool Read (const char *fname, int mx, int my=1, int [Method on mglDataC]
    mz=1)
int mgl_data_read_dim (HMDT dat, const char *fname, int mx, [C function]
    int my, int mz)
int mgl_datac_read_dim (HADT dat, const char *fname, int mx, [C function]
    int my, int mz)
    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]
bool ReadMat (const char *fname, int dim=2) [Method on mglData]
```

```
bool ReadMat (const char *fname, int dim=2) [Method on mglDataC]
int mgl_data_read_mat (HMDT dat, const char *fname, int dim) [C function]
int mgl_datac_read_mat (HADT dat, const char *fname, int dim) [C function]
    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]
void ReadRange (const char *templ, mreal from, mreal to, mreal step=1, bool as_slice=false) [Method on mglData]
void ReadRange (const char *templ, mreal from, mreal to, mreal step=1, bool as_slice=false) [Method on mglDataC]
int mgl_data_read_range (HMDT dat, const char *templ, mreal from, mreal to, mreal step, int as_slice) [C function]
int mgl_datac_read_range (HADT dat, const char *templ, mreal from, mreal to, mreal step, int as_slice) [C function]
    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]
void ReadAll (const char *templ, bool as_slice=false) [Method on mglData]
void ReadAll (const char *templ, bool as_slice=false) [Method on mglDataC]
int mgl_data_read_all (HMDT dat, const char *templ, int as_slice) [C function]
int mgl_datac_read_all (HADT dat, const char *templ, int as_slice) [C function]
    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]
bool ScanFile (const char *fname, const char *templ) [Method on mglData]
int mgl_data_scan_file (HMDT dat, const char *fname, const char *templ) [C function]
    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 2.9.22 [Saving and
    scanning file], page 140, for sample code and picture.
```

```
save dat 'fname' [MGL command]
void Save (const char *fname, int ns=-1) const [Method on mglDataA]
void mgl_data_save (HCDT dat, const char *fname, int ns) [C function]
void mgl_datac_save (HCDT dat, const char *fname, int ns) [C function]
    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 2.9.22 [Saving
    and scanning file], page 140, for sample code and picture.
```

```

readhdf DAT 'fname' 'dname' [MGL command]
void ReadHDF (const char *fname, const char *dname) [Method on mglData]
void ReadHDF (const char *fname, const char *dname) [Method on mglDataC]
void mgl_data_read_hdf (HMDT dat, const char *fname, const char *dname) [C function]
void mgl_datac_read_hdf (HADT dat, const char *fname, const char *dname) [C function]

```

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]
void SaveHDF (const char *fname, const char *dname, bool rewrite=false) const [Method on mglDataA]
void mgl_data_save_hdf (HCDT dat, const char *fname, const char *dname, int rewrite) [C function]
void mgl_datac_save_hdf (HCDT dat, const char *fname, const char *dname, int rewrite) [C function]

```

Saves data array named *dname* to HDF5 file. This function does nothing if HDF5 was disabled during library compilation.

```

datas 'fname' [MGL command]
int DatasHDF (const char *fname, char *buf, long size) [Method on mglDataA]
static
int mgl_datas_hdf (const char *fname, char *buf, long size) [C function]
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]
void Import (const char *fname, const char *scheme, mreal v1=0, mreal v2=1) [Method on mglData]
void mgl_data_import (HMDT dat, const char *fname, const char *scheme, mreal v1, mreal v2) [C function]

```

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 3.4 [Color scheme], page 149).

```

export dat 'fname' 'sch' [v1=0 v2=0] [MGL command]
void Export (const char *fname, const char *scheme, mreal v1=0, mreal v2=0, int ns=-1) const [Method on mglDataA]
void mgl_data_export (HMDT dat, const char *fname, const char *scheme, mreal v1, mreal v2, int ns) const [C function]

```

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 3.4 [Color scheme], page 149). If *v1* ≥ *v2* then the values of *v1*, *v2* are automatically determined as minimal and maximal value of the data array.

## 6.6 Make another data

```
subdata RES dat xx [yy=: zz=:] [MGL command]
mglData SubData (mreal xx, mreal yy=-1, mreal zz=-1) [Method on mglData]
    const
mglData SubData (mreal xx, mreal yy=-1, mreal [Method on mglDataC]
    zz=-1) const
HMDT mgl_data_subdata (HCDT dat, mreal xx, mreal yy, mreal zz) [C function]
    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]
mglData SubData (const mglDataA &xx, const mglDataA [Method on mglData]
    &yy, const mglDataA &zz) const
mglData SubData (const mglDataA &xx, const mglDataA [Method on mglDataC]
    &yy, const mglDataA &zz) const
mglData SubData (const mglDataA &xx, const mglDataA [Method on mglData]
    &yy) const
mglData SubData (const mglDataA &xx, const mglDataA [Method on mglDataC]
    &yy) const
mglData SubData (const mglDataA &xx) const [Method on mglData]
mglData SubData (const mglDataA &xx) const [Method on mglDataC]
HMDT mgl_data_subdata_ext (HCDT dat, HCDT xx, HCDT yy, HCDT [C function]
    zz)
HADT mgl_datac_subdata_ext (HCDT dat, HCDT xx, HCDT yy, HCDT [C function]
    zz)
```

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]
mglData Column (const char *eq) const [Method on mglData]
mglData Column (const char *eq) const [Method on mglDataC]
HMDT mgl_data_column (HCDT dat, const char *eq) [C function]
    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 261,
    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]
mglData Resize (int mx, int my=0, int mz=0, mreal [Method on mglData]
    x1=0, mreal x2=1, mreal y1=0, mreal y2=1, mreal z1=0, mreal z2=1)
    const
mglData Resize (int mx, int my=0, int mz=0, mreal [Method on mglDataC]
    x1=0, mreal x2=1, mreal y1=0, mreal y2=1, mreal z1=0, mreal z2=1)
    const
HMDT mgl_data_resize (HCDT dat, int mx, int my, int mz) [C function]
HMDT mgl_data_resize_box (HCDT dat, int mx, int my, int mz, [C function]
    mreal x1, mreal x2, mreal y1, mreal y2, mreal z1, mreal z2)
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]
mglData Evaluate (const mglDataA &idat, bool [Method on mglData]
    norm=true) const
mglData Evaluate (const mglDataA &idat, const [Method on mglData]
    mglDataA &jdat, bool norm=true) const
mglData Evaluate (const mglDataA &idat, const [Method on mglData]
    mglDataA &jdat, const mglDataA &kdat, bool norm=true) const
mglData Evaluate (const mglDataA &idat, bool [Method on mglDataC]
    norm=true) const
mglData Evaluate (const mglDataA &idat, const [Method on mglDataC]
    mglDataA &jdat, bool norm=true) const
mglData Evaluate (const mglDataA &idat, const [Method on mglDataC]
    mglDataA &jdat, const mglDataA &kdat, bool norm=true) const
HMDT mgl_data_evaluate (HCDT dat, HCDT idat, HCDT jdat, HCDT [C function]
    kdat, int norm)
```

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]
mglData Solve (mreal val, char dir, bool norm=true) [Method on mglData]
    const
mglData Solve (mreal val, char dir, const mglDataA [Method on mglData]
    &idat, bool norm=true) const
```

HMDT `mg1_data_solve` (HCDT *dat*, mreal *val*, char *dir*, HCDT *idat*, [C function]  
int *norm*)

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 consecutive 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 39, for sample code and picture.

roots RES '*func*' ini ['*var*'='x'] [MGL command]

roots RES '*func*' ini ['*var*'='x'] [MGL command]

mg1Data Roots (const char \**func*, char *var*) const [Method on mg1Data]

HMDT `mg1_data_roots` (const char \**func*, HCDT *ini*, char *var*) [C function]

mreal `mg1_find_root_txt` (const char \**func*, mreal *ini*, char  
*var*) [C function]

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]

mg1Data Hist (int *n*, mreal *v1*=0, mreal *v2*=1, int  
*nsub*=0) const [Method on mg1Data]

mg1Data Hist (const mg1DataA &*w*, int *n*, mreal *v1*=0,  
mreal *v2*=1, int *nsub*=0) const [Method on mg1Data]

mg1Data Hist (int *n*, mreal *v1*=0, mreal *v2*=1, int  
*nsub*=0) const [Method on mg1DataC]

mg1Data Hist (const mg1DataA &*w*, int *n*, mreal *v1*=0,  
mreal *v2*=1, int *nsub*=0) const [Method on mg1DataC]

HMDT `mg1_data_hist` (HCDT *dat*, int *n*, mreal *v1*, mreal *v2*, int  
*nsub*) [C function]

HMDT `mg1_data_hist_w` (HCDT *dat*, HCDT *w*, int *n*, mreal *v1*, mreal  
*v2*, int *nsub*) [C function]

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 4.18 [Data manipulation], page 235,

momentum RES *dat* '*how*' ['*dir*'='z'] [MGL command]

mg1Data Momentum (char *dir*, const char \**how*) const [Method on mg1Data]

mg1Data Momentum (char *dir*, const char \**how*) const [Method on mg1DataC]

HMDT `mg1_data_momentum` (HCDT *dat*, char *dir*, const char \**how*) [C function]

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]
mglData Sum (const char *dir) const [Method on mglData]
mglDataC Sum (const char *dir) const [Method on mglDataC]
HMDT mgl_data_sum (HCDT dat, const char *dir) [C function]
    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]
mglData Max (const char *dir) const [Method on mglData]
mglDataC Max (const char *dir) const [Method on mglDataC]
HMDT mgl_data_max_dir (HCDT dat, const char *dir) [C function]
    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]
mglData Min (const char *dir) const [Method on mglData]
mglDataC Min (const char *dir) const [Method on mglDataC]
HMDT mgl_data_min_dir (HCDT dat, const char *dir) [C function]
    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]
mglData Combine (const mglDataA &a) const [Method on mglData]
mglDataC Combine (const mglDataA &a) const [Method on mglDataC]
HMDT mgl_data_combine (HCDT dat, HCDT a) [C function]
    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]
mglData Trace () const [Method on mglData]
mglDataC Trace () const [Method on mglDataC]
HMDT mgl_data_trace (HCDT dat) [C function]
    Gets array of diagonal elements  $a[i,i]$  (for 2D case) or  $a[i,i,i]$  (for 3D case) where  $i=0\dots nx-1$ . Function return copy of itself for 1D case. Data array must have dimensions  $ny,nz \geq nx$  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]
mglData Correl (const mglDataA &b, const char *dir) const [Method on mglData]
mglDataC Correl (const mglDataA &b, const char *dir) const [Method on mglDataC]
mglDataC AutoCorrel (const char *dir) const [Method on mglDataC]
HMDT mgl_data_correl (HCDT a, HCDT b, const char *dir) [C function]
```

**HADT mgl\_datac\_correl** (HCDT *a*, HCDT *b*, const char \**dir*) [C function]  
 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 270, or [norm], page 271, before plotting it. Function return NULL or create empty data if data cannot be created for given arguments.

**mglData Real** () const [Method on mglDataC]  
**HMDT mgl\_datac\_real** (HCDT *dat*) [C function]  
 Gets array of real parts of the data.

**mglData Imag** () const [Method on mglDataC]  
**HMDT mgl\_datac\_imag** (HCDT *dat*) [C function]  
 Gets array of imaginary parts of the data.

**mglData Abs** () const [Method on mglDataC]  
**HMDT mgl\_datac\_abs** (HCDT *dat*) [C function]  
 Gets array of absolute values of the data.

**mglData Arg** () const [Method on mglDataC]  
**HMDT mgl\_datac\_arg** (HCDT *dat*) [C function]  
 Gets array of arguments of the data.

**pulse** RES *dat* '*dir*' [MGL command]  
**mglData Pulse** (const char \**dir*) const [Method on mglData]  
**HMDT mgl\_data\_pulse** (HCDT *dat*, const char \**dir*) [C function]  
 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*]|<sup>2</sup>) 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 267, [min], page 267, [momentum], page 266, [sum], page 267. See Section 2.9.16 [Pulse properties], page 134, for sample code and picture.

## 6.7 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]  
**void CumSum** (const char \**dir*) [Method on mglData]  
**void CumSum** (const char \**dir*) [Method on mglDataC]  
**void mgl\_data\_cumsum** (HMDT *dat*, const char \**dir*) [C function]  
**void mgl\_datac\_cumsum** (HADT *dat*, const char \**dir*) [C function]  
 Cumulative summation of the data in given direction or directions.



```

integrate dat 'dir' [MGL command]
void Integral (const char *dir) [Method on mglData]
void Integral (const char *dir) [Method on mglDataC]
void mgl_data_integral (HMDT dat, const char *dir) [C function]
void mgl_datac_integral (HADT dat, const char *dir) [C function]
    Integrates (like cumulative summation) the data in given direction or directions.

```

```

diff dat 'dir' [MGL command]
void Diff (const char *dir) [Method on mglData]
void Diff (const char *dir) [Method on mglDataC]
void mgl_data_diff (HMDT dat, const char *dir) [C function]
void mgl_datac_diff (HADT dat, const char *dir) [C function]
    Differentiates the data in given direction or directions.

```

```

diff dat xdat ydat [zdat=0] [MGL command]
void Diff (const mglDataA &x, const mglDataA &y) [Method on mglData]
void Diff (const mglDataA &x, const mglDataA &y, [Method on mglData]
           const mglDataA &z)
void mgl_data_diff_par (HMDT dat, HCDT x, HCDT y, HCDT z) [C function]
    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]
void Diff2 (const char *dir) [Method on mglData]
void Diff2 (const char *dir) [Method on mglDataC]
void mgl_data_diff2 (HMDT dat, const char *dir) [C function]
void mgl_datac_diff2 (HADT dat, const char *dir) [C function]
    Double-differentiates (like Laplace operator) the data in given direction.

```

```

sinfft dat 'dir' [MGL command]
void SinFFT (const char *dir) [Method on mglData]
void mgl_data_sinfft (HMDT dat, const char *dir) [C function]
    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]
void CosFFT (const char *dir) [Method on mglData]
void mgl_data_cosfft (HMDT dat, const char *dir) [C function]
    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).

```

`void FFT (const char *dir)` [Method on `mglDataC`]  
`void mgl_datac_fft (HADT dat, const char *dir)` [C function]  
 Do Fourier transform of the data in given direction or directions. If *dir* contain 'i' then inverse Fourier is used. The Fourier transform is  $\sum a_j \exp(ikj)$  (see [http://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform](http://en.wikipedia.org/wiki/Discrete_Fourier_transform)).

`hankel dat 'dir'` [MGL command]  
`void Hankel (const char *dir)` [Method on `mglData`]  
`void Hankel (const char *dir)` [Method on `mglDataC`]  
`void mgl_data_hankel (HMDT dat, const char *dir)` [C function]  
`void mgl_datac_hankel (HADT dat, const char *dir)` [C function]  
 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](http://en.wikipedia.org/wiki/Hankel_transform)).

`wavelet dat 'dir' k` [MGL command]  
`void Wavelet (const char *dir, int k)` [Method on `mglData`]  
`void mgl_data_wavelet (HMDT dat, const char *dir, int k)` [C function]  
 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]  
`void Swap (const char *dir)` [Method on `mglData`]  
`void Swap (const char *dir)` [Method on `mglDataC`]  
`void mgl_data_swap (HMDT dat, const char *dir)` [C function]  
`void mgl_datac_swap (HADT dat, const char *dir)` [C function]  
 Swaps the left and right part of the data in given direction (useful for Fourier spectrum).

`roll dat 'dir' num` [MGL command]  
`void Roll (char dir, num)` [Method on `mglData`]  
`void Roll (char dir, num)` [Method on `mglDataC`]  
`void mgl_data_roll (HMDT dat, char dir, num)` [C function]  
`void mgl_datac_roll (HADT dat, char dir, num)` [C function]  
 Rolls the data along direction *dir*. Resulting array will be `out[i] = ini[(i+num)%nx]` if *dir*='x'.

`mirror dat 'dir'` [MGL command]  
`void Mirror (const char *dir)` [Method on `mglData`]  
`void Mirror (const char *dir)` [Method on `mglDataC`]  
`void mgl_data_mirror (HMDT dat, const char *dir)` [C function]  
`void mgl_datac_mirror (HADT dat, const char *dir)` [C function]  
 Mirror the left-to-right part of the data in given direction. Looks like change the value index  $i \rightarrow n-i$ . Note, that the similar effect in graphics you can reach by using options (see Section 3.7 [Command options], page 153), for example, `surf dat; xrange 1 -1`.

```

sew dat ['dir'='xyz' da=2*pi] [MGL command]
void Sew (const char *dir, mreal da=2*M_PI) [Method on mglData]
void mgl_data_sew (HMDT dat, const char *dir, mreal da) [C function]
    Remove value steps (like phase jumps after inverse trigonometric functions) with
    period da in given direction.

smooth data type ['dir'='xyz'] [MGL command]
void Smooth (const char *dir="xyz", mreal delta=0) [Method on mglData]
void Smooth (const char *dir="xyz", mreal delta=0) [Method on mglDataC]
void mgl_data_smooth (HMDT dat, const char *dir, mreal delta) [C function]
void mgl_datac_smooth (HADT dat, const char *dir, mreal delta) [C function]
    Smooths the data on specified direction or directions. String dirs specifies the dimen-
    sions 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]
void Envelop (char dir='x') [Method on mglData]
void mgl_data_envelop (HMDT dat, char dir) [C function]
    Find envelop for data values along direction dir.

norm dat v1 v2 [sym=off dim=0] [MGL command]
void Norm (mreal v1=0, mreal v2=1, bool sym=false, [Method on mglData]
           int dim=0)
    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  $\geq dim$ .

normsl dat v1 v2 ['dir'='z' keep=on sym=off] [MGL command]
void NormSl (mreal v1=0, mreal v2=1, char dir='z', [Method on mglData]
             bool keep_en=true, bool sym=false)
void mgl_data_norm_slice (HMDT dat, mreal v1, mreal v2, char [C function]
                          dir, int keep_en, int sym)
    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

$$\sqrt{\sum a_{ij}(k) / \sum a_{ij}(0)}.$$


```

## 6.8 Interpolation

MGL scripts can use spline interpolation by [evaluate], page 265, or [refill], page 260, commands. Also you can use [resize], page 265, for obtaining a data array with new sizes.

However, there are much special faster functions in other modes (C/C++/Fortran/Python/...). ■

```

mreal Spline (mreal x, mreal y=0, mreal z=0) const [Method on mglData]
dual Spline (mreal x, mreal y=0, mreal z=0) const [Method on mglDataC]

```

```
mreal mgl_data_spline (HCDT dat, mreal x, mreal y, mreal z)          [C function]
dual mgl_datac_spline (HCDT dat, mreal x, mreal y, mreal z)          [C function]
    Interpolates data by cubic spline to the given point x in [0...nx-1], y in [0...ny-1], z
    in [0...nz-1].
```

```
mreal Spline1 (mreal x, mreal y=0, mreal z=0) const                    [Method on mglData]
dual Spline1 (mreal x, mreal y=0, mreal z=0) const                    [Method on mglDataC]
    Interpolates data by cubic spline to the given point x, y, z which assumed to be
    normalized in range [0, 1].
```

```
mreal Spline (mglPoint &dif, mreal x, mreal y=0,                      [Method on mglData]
               mreal z=0) const
mreal mgl_data_spline_ext (HCDT dat, mreal x, mreal y, mreal          [C function]
                          z, mreal *dx, mreal *dy, mreal *dz)
dual mgl_datac_spline_ext (HCDT dat, mreal x, mreal y, mreal          [C function]
                          z, dual *dx, dual *dy, dual *dz)
    Interpolates data by cubic spline to the given point x in [0...nx-1], y in [0...ny-1], z
    in [0...nz-1]. The values of derivatives at the point are saved in dif.
```

```
mreal Spline1 (mglPoint &dif, mreal x, mreal y=0,                    [Method on mglData]
               mreal z=0) const
    Interpolates data by cubic spline to the given point x, y, z which assumed to be
    normalized in range [0, 1]. The values of derivatives at the point are saved in dif.
```

```
mreal Linear (mreal x, mreal y=0, mreal z=0) const                  [Method on mglData]
dual Linear (mreal x, mreal y=0, mreal z=0) const                  [Method on mglDataC]
mreal mgl_data_linear (HCDT dat, mreal x, mreal y, mreal z)          [C function]
dual mgl_datac_linear (HCDT dat, mreal x, mreal y, mreal z)          [C function]
    Interpolates data by linear function to the given point x in [0...nx-1], y in [0...ny-1],
    z in [0...nz-1].
```

```
mreal Linear1 (mreal x, mreal y=0, mreal z=0) const                [Method on mglData]
dual Linear1 (mreal x, mreal y=0, mreal z=0) const                [Method on mglDataC]
    Interpolates data by linear function to the given point x, y, z which assumed to be
    normalized in range [0, 1].
```

```
mreal Linear (mglPoint &dif, mreal x, mreal y=0,                    [Method on mglData]
               mreal z=0) const
dual Linear (mglPoint &dif, mreal x, mreal y=0,                    [Method on mglDataC]
               mreal z=0) const
mreal mgl_data_linear_ext (HCDT dat, mreal x, mreal y, mreal          [C function]
                          z, mreal *dx, mreal *dy, mreal *dz)
dual mgl_datac_linear_ext (HCDT dat, mreal x, mreal y, mreal          [C function]
                          z, dual *dx, dual *dy, dual *dz)
    Interpolates data by linear function to the given point x in [0...nx-1], y in [0...ny-1],
    z in [0...nz-1]. The values of derivatives at the point are saved in dif.
```

```
mreal Linear1 (mglPoint &dif, mreal x, mreal y=0,          [Method on mglData]
               mreal z=0) const
```

```
dual Linear1 (mglPoint &dif, mreal x, mreal y=0,          [Method on mglDataC]
               mreal z=0) const
```

Interpolates data by linear function to the given point  $x, y, z$  which assumed to be normalized in range  $[0, 1]$ . The values of derivatives at the point are saved in *dif*.

## 6.9 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]
```

```
const char * PrintInfo () const                            [Method on mglDataA]
```

```
void PrintInfo (FILE *fp) const                            [Method on mglDataA]
```

```
const char * mgl_data_info (HCDT dat)                     [C function only]
```

```
    mgl_data_info (long dat, char *out, int len)           [Fortran subroutine]
```

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 273, 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]
```

```
long GetNx ()                                                [Method on mglDataA]
```

```
long GetNy ()                                                [Method on mglDataA]
```

```
long GetNz ()                                                [Method on mglDataA]
```

```
long mgl_data_get_nx (HCDT dat)                             [C function]
```

```
long mgl_data_get_ny (HCDT dat)                             [C function]
```

```
long mgl_data_get_nz (HCDT dat)                             [C function]
```

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

```

(dat) .max [MGL suffix]
mreal Maximal () const [Method on mglDataA]
mreal mgl_data_max (HCDT dat) [C function]
    Gets maximal value of the data.

(dat) .min [MGL suffix]
mreal Minimal () const [Method on mglDataA]
mreal mgl_data_min (HMDT dat) const [C function]
    Gets minimal value of the data.

mreal Minimal (int &i, int &j, int &k) const [Method on mglDataA]
mreal mgl_data_min_int (HCDT dat, int *i, int *j, int *k) [C function]
    Gets position of minimum to variables i, j, k and returns the minimal value.

mreal Maximal (int &i, int &j, int &k) const [Method on mglDataA]
mreal mgl_data_max_int (HCDT dat, int *i, int *j, int *k) [C function]
    Gets position of maximum to variables i, j, k and returns the maximal value.

mreal Minimal (mreal &x, mreal &y, mreal &z) const [Method on mglDataA]
mreal mgl_data_min_real (HCDT dat, mreal *x, mreal *y, mreal *z) [C function]
    Gets approximated (interpolated) position of minimum to variables x, y, z and returns
    the minimal value.

(dat) .mx [MGL suffix]
(dat) .my [MGL suffix]
(dat) .mz [MGL suffix]
mreal Maximal (mreal &x, mreal &y, mreal &z) const [Method on mglDataA]
mreal mgl_data_max_real (HCDT dat, mreal *x, mreal *y, mreal *z) [C function]
    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]
mreal Momentum (char dir, mreal &a, mreal &w) const [Method on mglDataA]
mreal Momentum (char dir, mreal &m, mreal &w, mreal [Method on mglDataA]
    &s, mreal &k) const
mreal mgl_data_momentum_val (HCDT dat, char dir, mreal *a, [C function]
    mreal *w, mreal *s, mreal *k)
    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  $\xi$  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]
mreal Find (const char *cond, int &i, int &j, int &k) [Method on mglDataA]
    const
mreal mgl_data_first (HCDT dat, const char *cond, int *i, int [C function]
    *j, int *k)
    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]
mreal Last (const char *cond, int &i, int &j, int &k) [Method on mglDataA]
    const
mreal mgl_data_last (HCDT dat, const char *cond, int *i, int [C function]
    *j, int *k)
    Find position (before specified in i, j, k) of last nonzero value of formula cond. Function return the data value at found position.
```

```
int Find (const char *cond, char dir, int i=0, int [Method on mglDataA]
    j=0, int k=0) const
mreal mgl_data_find (HCDT dat, const char *cond, int i, int j, [C function]
    int k)
    Return position of first in direction dir nonzero value of formula cond. The search is started from point {i,j,k}.
```

```
bool FindAny (const char *cond) const [Method on mglDataA]
mreal mgl_data_find_any (HCDT dat, const char *cond) [C function]
    Determines if any nonzero value of formula in the data array.
```

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

## 6.10 Operators

```
copy DAT dat2 ['eq']=" [MGL command]
void operator= (const mglDataA &d) [Method on mglData]
    Copies data from other variable.
```

<code>copy dat val</code>	[MGL command]
<code>void operator= (mreal val)</code>	[Method on mreal]
Set all data values equal to <i>val</i> .	
<code>multo dat dat2</code>	[MGL command]
<code>multo dat val</code>	[MGL command]
<code>void operator*= (const mglDataA &amp;d)</code>	[Method on mglData]
<code>void operator*= (mreal d)</code>	[Method on mglData]
<code>void mgl_data_mul_dat (HMDT dat, HCDT d)</code>	[C function]
<code>void mgl_data_mul_num (HMDT dat, mreal d)</code>	[C function]
Multiplies data element by the other one or by value.	
<code>divto dat dat2</code>	[MGL command]
<code>divto dat val</code>	[MGL command]
<code>void operator/= (const mglDataA &amp;d)</code>	[Method on mglData]
<code>void operator/= (mreal d)</code>	[Method on mglData]
<code>void mgl_data_div_dat (HMDT dat, HCDT d)</code>	[C function]
<code>void mgl_data_div_num (HMDT dat, mreal d)</code>	[C function]
Divides each data element by the other one or by value.	
<code>addto dat dat2</code>	[MGL command]
<code>addto dat val</code>	[MGL command]
<code>void operator+= (const mglDataA &amp;d)</code>	[Method on mglData]
<code>void operator+= (mreal d)</code>	[Method on mglData]
<code>void mgl_data_add_dat (HMDT dat, HCDT d)</code>	[C function]
<code>void mgl_data_add_num (HMDT dat, mreal d)</code>	[C function]
Adds to each data element the other one or the value.	
<code>subto dat dat2</code>	[MGL command]
<code>subto dat val</code>	[MGL command]
<code>void operator-= (const mglDataA &amp;d)</code>	[Method on mglData]
<code>void operator-= (mreal d)</code>	[Method on mglData]
<code>void mgl_data_sub_dat (HMDT dat, HCDT d)</code>	[C function]
<code>void mgl_data_sub_num (HMDT dat, mreal d)</code>	[C function]
Subtracts from each data element the other one or the value.	
<code>mglData operator+ (const mglDataA &amp;a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator+ (mreal a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator+ (const mglDataA &amp;a, mreal b)</code>	[Library Function]
Adds the other data or the number.	
<code>mglData operator- (const mglDataA &amp;a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator- (mreal a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator- (const mglDataA &amp;a, mreal b)</code>	[Library Function]
Subtracts the other data or the number.	



<code>mglData operator* (const mglDataA &amp;a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator* (mreal a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator* (const mglDataA &amp;a, mreal b)</code>	[Library Function]
Multiplies by the other data or the number.	
<code>mglData operator/ (const mglDataA &amp;a, const mglDataA &amp;b)</code>	[Library Function]
<code>mglData operator/ (const mglDataA &amp;a, mreal b)</code>	[Library Function]
Divides by the other data or the number.	

## 6.11 Global functions

These functions are not methods of `mglData` class. However it provide additional functionality to handle data. So I put it in this chapter.

<code>transform DAT 'type' real imag</code>	[MGL command]
<code>mglData mglTransform (const mglDataA &amp;real, const mglDataA &amp;imag, const char *type)</code>	[Global function]
<code>HMDT mgl_transform (HCDT real, HCDT imag, const char *type)</code>	[C function]
Does integral transformation of complex data <i>real</i> , <i>imag</i> on specified direction. The order of transformations is specified in string <i>type</i> : 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.	
<code>transforma DAT 'type' ampl phase</code>	[MGL command]
<code>mglData mglTransformA const mglDataA &amp;ampl, const mglDataA &amp;phase, const char *type)</code>	[Global function]
<code>HMDT mgl_transform_a HCDT ampl, HCDT phase, const char *type)</code>	[C function]
The same as previous but with specified amplitude <i>ampl</i> and phase <i>phase</i> of complex numbers.	
<code>fourier reDat imDat 'dir'</code>	[MGL command]
<code>fourier complexDat 'dir'</code>	[MGL command]
<code>void mglFourier const mglDataA &amp;re, const mglDataA &amp;im, const char *dir)</code>	[Global function]
<code>void mgl_data_fourier HCDT re, HCDT im, const char *dir)</code>	[C function]
Does Fourier transform of complex data <i>re</i> + <i>i</i> * <i>im</i> in directions <i>dir</i> . Result is placed back into <i>re</i> and <i>im</i> data arrays.	
<code>stfad RES real imag dn ['dir']='x']</code>	[MGL command]
<code>mglData mglSTFA (const mglDataA &amp;real, const mglDataA &amp;imag, int dn, char dir='x')</code>	[Global function]
<code>HMDT mgl_data_stfa (HCDT real, HCDT imag, int dn, char dir)</code>	[C function]
Short time Fourier transformation for real and imaginary parts. Output is amplitude of partial Fourier of length <i>dn</i> . For example if <i>dir</i> ='x', result will have size {int(nx/dn), dn, ny} and it will contain $res[i, j, k] =  \sum_d^n \exp(I * j * d) * (real[i * dn + d, k] + I * imag[i * dn + d, k])  / dn$ .	

```

pde RES 'ham' ini_re ini_im [dz=0.1 k0=100] [MGL command]
mglData mglPDE (HMGL gr, const char *ham, const mglDataA [Global function]
    &ini_re, const mglDataA &ini_im, mreal dz=0.1, mreal k0=100, const
    char *opt="")
mglDataC mglPDEc (HMGL gr, const char *ham, const [Global function]
    mglDataA &ini_re, const mglDataA &ini_im, mreal dz=0.1, mreal
    k0=100, const char *opt="")
HMDT mgl_pde_solve (HMGL gr, const char *ham, HCDT ini_re, [C function]
    HCDT ini_im, mreal dz, mreal k0, const char *opt)
HADT mgl_pde_solve_c (HMGL gr, const char *ham, HCDT ini_re, [C function]
    HCDT ini_im, mreal dz, mreal k0, const char *opt)

```

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<sup>2</sup> + q<sup>2</sup> - u<sup>2</sup>". You may specify imaginary part for wave absorption, like *ham* = "p<sup>2</sup> + i\*x\*(x>0)", but only if dependence on variable ‘*i*’ is linear (i.e.  $ham = h_{re} + i * h_{im}$ ). See also [qo2d], page 279, [qo3d], page 279. See Section 2.9.14 [PDE solving hints], page 130, for sample code and picture.

```

ray RES 'ham' x0 y0 z0 p0 q0 v0 [dt=0.1 tmax=10] [MGL command]
mglData mglRay (const char *ham, mglPoint r0, mglPoint [Global function]
    p0, mreal dt=0.1, mreal tmax=10)
HMDT mgl_ray_trace (const char *ham, mreal x0, mreal y0, [C function]
    mreal z0, mreal px, mreal py, mreal pz, mreal dt, mreal tmax)

```

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]
mglData mglODE (const char *df, const char *var, const [Global function]
    mglDataA &ini, mreal dt=0.1, mreal tmax=10)
HMDT mgl_ode_solve_str (const char *df, const char *var, HCDT [C function]
    ini, mreal dt, mreal tmax)
HMDT mgl_ode_solve (void (*df)(const mreal *x, mreal *dx, [C function]
    void *par), int n, const mreal *ini, mreal dt, mreal tmax)

```

```
HMDT mgl_ode_solve_ex (void (*df)(const mreal *x, mreal *dx,    [C function]
                        void *par), int n, const mreal *ini, mreal dt, mreal tmax, void
                        (*bord)(mreal *x, const mreal *xprev, void *par))
```

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  $\{n * \text{int}(tmax/dt+1)\}$ .

```
qo2d RES 'ham' ini_re ini_im ray [r=1 k0=100 xx yy] [MGL command]
```

```
mglData mglQO2d (const char *ham, const mglDataA &ini_re,    [Global function]
                const mglDataA &ini_im, const mglDataA &ray, mreal r=1, mreal
                k0=100)
```

```
mglData mglQO2d (const char *ham, const mglDataA &ini_re,    [Global function]
                const mglDataA &ini_im, const mglDataA &ray, mglData &xx, mglData
                &yy, mreal r=1, mreal k0=100)
```

```
mglDataC mglQO2dc (const char *ham, const mglDataA          [Global function]
                  &ini_re, const mglDataA &ini_im, const mglDataA &ray, mreal r=1,
                  mreal k0=100)
```

```
mglDataC mglQO2dc (const char *ham, const mglDataA          [Global function]
                  &ini_re, const mglDataA &ini_im, const mglDataA &ray, mglData &xx,
                  mglData &yy, mreal r=1, mreal k0=100)
```

```
HMDT mgl_qo2d_solve (const char *ham, HCDT ini_re, HCDT      [C function]
                    ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy)
```

```
HADT mgl_qo2d_solve_c (const char *ham, HCDT ini_re, HCDT   [C function]
                      ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy)
```

```
HMDT mgl_qo2d_func (dual (*ham)(mreal u, mreal x, mreal y,    [C function]
                             mreal px, mreal py, void *par), HCDT ini_re, HCDT ini_im, HCDT ray,
                             mreal r, mreal k0, HMDT xx, HMDT yy)
```

```
HADT mgl_qo2d_func_c (dual (*ham)(mreal u, mreal x, mreal     [C function]
                             y, mreal px, mreal py, void *par), HCDT ini_re, HCDT ini_im, HCDT ray,
                             mreal r, mreal k0, HMDT xx, HMDT yy)
```

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 made. You may use, for example, the array created by `[ray]`, page 278, function. Note, that the reference ray **must be** smooth enough to make accompanied coordinates 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 278, `[qo3d]`, page 279. See Section 2.9.14 [PDE solving hints], page 130, for sample code and picture.

```
qo3d RES 'ham' ini_re ini_im ray [r=1 k0=100 xx yy zz] [MGL command]
```

```
mglData mglQO3d (const char *ham, const mglDataA &ini_re,    [Global function]
                const mglDataA &ini_im, const mglDataA &ray, mreal r=1, mreal
                k0=100)
```

```

mgldata mglQ03d (const char *ham, const mgldataA &ini_re,      [Global function]
                const mgldataA &ini_im, const mgldataA &ray, mgldata &xx, mgldata
                &yy, mgldata &zz, mreal r=1, mreal k0=100)
mgldataC mglQ03dc (const char *ham, const mgldataA              [Global function]
                  &ini_re, const mgldataA &ini_im, const mgldataA &ray, mreal r=1,
                  mreal k0=100)
mgldataC mglQ03dc (const char *ham, const mgldataA              [Global function]
                  &ini_re, const mgldataA &ini_im, const mgldataA &ray, mgldata &xx,
                  mgldata &yy, mgldata &zz, mreal r=1, mreal k0=100)
HMDT mgl_qo3d_solve (const char *ham, HCDT ini_re, HCDT        [C function]
                    ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy, HMDT zz)
HADT mgl_qo3d_solve_c (const char *ham, HCDT ini_re, HCDT      [C function]
                      ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy, HMDT zz)
HMDT mgl_qo3d_func (dual (*ham)(mreal u, mreal x, mreal y,      [C function]
                           mreal z, mreal px, mreal py, mreal pz, void *par), HCDT ini_re, HCDT
                           ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy, HMDT zz)
HADT mgl_qo3d_func_c (dual (*ham)(mreal u, mreal x, mreal      [C function]
                              y, mreal z, mreal px, mreal py, mreal pz, void *par), HCDT ini_re,
                              HCDT ini_im, HCDT ray, mreal r, mreal k0, HMDT xx, HMDT yy, HMDT zz)

```

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 made. You may use, for example, the array created by `[ray]`, page 278, function. Note, that the reference ray **must be** smooth enough to make accompanied coordinates 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 278, `[qo2d]`, page 279. See Section 2.9.14 [PDE solving hints], page 130, for sample code and picture.

```

jacobian RES xdat ydat [zdat]                                [MGL command]
mgldata mglJacobian (const mgldataA &x, const mgldataA        [Global function]
                  &y)
mgldata mglJacobian (const mgldataA &x, const mgldataA        [Global function]
                  &y, const mgldataA &z)
HMDT mgl_jacobian_2d (HCDT x, HCDT y)                        [C function]
HMDT mgl_jacobian_3d (HCDT x, HCDT y, HCDT z)                [C function]

```

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  $\xi=\{i,j,k\}$ . 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]
mgldata mglTriangulation (const mgldataA &x, const            [Global function]
                        mgldataA &y)

```

**HMDT mgl\_trianguation\_2d** (HCDT *x*, HCDT *y*) [C function]  
 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 230, or [tricont], page 230, functions for visualization of reconstructed surface. See Section 2.9.11 [Making regular data], page 127, for sample code and picture.

**mglData mglGSplineInit** (const mglDataA &*x*, const mglDataA &*y*) [Global function]

**mglDataC mglGSplineCInit** (const mglDataA &*x*, const mglDataA &*y*) [Global function]

**HMDT mgl\_gspline\_init** (HCDT *x*, HCDT *y*) [C function]

**HADT mgl\_gsplinec\_init** (HCDT *x*, HCDT *y*) [C function]  
 Prepare coefficients for global cubic spline interpolation.

**mreal mglGSpline** (const mglDataA &*coef*, mreal *dx*, mreal *\*d1*=0, mreal *\*d2*=0) [Global function]

**dual mglGSplineC** (const mglDataA &*coef*, mreal *dx*, dual *\*d1*=0, dual *\*d2*=0) [Global function]

**mreal mgl\_gspline** (HCDT *coef*, mreal *dx*, mreal *\*d1*, mreal *\*d2*) [C function]

**dual mgl\_gsplinec** (HCDT *coef*, mreal *dx*, dual *\*d1*, dual *\*d2*) [C function]  
 Evaluate global cubic spline (and its 1st and 2nd derivatives *d1*, *d2* if they are not NULL) using prepared coefficients *coef* at point *dx*+*x0* (where *x0* is 1st element of data *x* provided to mglGSpline\*Init() function).

**ifs2d RES dat num [skip=20]** [MGL command]

**mglData mglIFS2d** (const mglDataA &*dat*, long *num*, long *skip*=20) [Global function]

**HMDT mgl\_data\_ifs\_2d** (HCDT *dat*, long *num*, long *skip*) [C function]  
 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

$$\begin{aligned} x[i+1] &= dat[0,i]*x[i] + dat[1,i]*y[i] + dat[4,i]; \\ y[i+1] &= dat[2,i]*x[i] + dat[3,i]*y[i] + dat[5,i]; \end{aligned}$$

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 2.7.14 [IFS sample], page 100, for sample code and picture.

**ifs3d RES dat num [skip=20]** [MGL command]

**mglData mglIFS3d** (const mglDataA &*dat*, long *num*, long *skip*=20) [Global function]

**HMDT mgl\_data\_ifs\_3d** (HCDT *dat*, long *num*, long *skip*) [C function]  
 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

$$\begin{aligned} x[i+1] &= dat[0,i]*x[i] + dat[1,i]*y[i] + dat[2,i]*z[i] + dat[9,i]; \\ y[i+1] &= dat[3,i]*x[i] + dat[4,i]*y[i] + dat[5,i]*z[i] + dat[10,i]; \\ z[i+1] &= dat[6,i]*x[i] + dat[7,i]*y[i] + dat[8,i]*z[i] + dat[10,i]; \end{aligned}$$

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 2.7.14 [IFS sample], page 100, for sample code and picture.

## 6.12 Evaluate expression

MathGL have a special classes `mg1Expr` and `mg1ExprC` for evaluating of formula specified by the string for real and complex numbers correspondingly. These classes are defined in `#include <mg12/data.h>` and `#include <mg12/datac.h>` correspondingly. It is the fast variant of formula evaluation. At creation it will be recognized and compiled to tree-like internal code. At evaluation stage only fast calculations are performed. 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. See Section 3.6 [Textual formulas], page 152.

```
mg1Expr (const char *expr) [Constructor on mg1Expr]
mg1ExprC (const char *expr) [Constructor on mg1ExprC]
HMEX mg1_create_expr (const char *expr) [C function]
HAEX mg1_create_cexpr (const char *expr) [C function]
```

Parses the formula *expr* and creates formula-tree. Constructor recursively parses the formula and creates a tree-like structure containing functions and operators for fast further evaluating by `Calc()` or `CalcD()` functions.

```
~mg1Expr () [Destructor on mg1Expr]
~mg1ExprC () [Destructor on mg1ExprC]
void mg1_delete_expr (HMEX ex) [C function]
void mg1_delete_cexpr (HAEX ex) [C function]
Deletes the instance of class mg1Expr.
```

```
mreal Eval (mreal x, mreal y, mreal z) [Method on mg1Expr]
dual Eval (dual x, dual y, dual z) [Method on mg1ExprC]
mreal mg1_expr_eval (HMEX ex, mreal x, mreal y, mreal z) [C function]
dual mg1_cexpr_eval (HAEX ex, dual x, dual y, dual z) [C function]
Evaluates the formula for 'x', 'r'=x, 'y', 'n'=y, 'z', 't'=z, 'a', 'u'=u.
```

```
mreal Eval (mreal var[26]) [Method on mg1Expr]
dual Eval (dual var[26]) [Method on mg1ExprC]
mreal mg1_expr_eval_v (HMEX ex, mreal *var) [C function]
dual mg1_expr_eval_v (HAEX ex, dual *var) [C function]
Evaluates the formula for variables in array var[0,...,'z'-'a'].
```

```
mreal Diff (char dir, mreal x, mreal y, mreal z) [Method on mg1Expr]
mreal mg1_expr_diff (HMEX ex, char dir, mreal x, mreal y, mreal z) [C function]
Evaluates the formula derivation respect to dir for 'x', 'r'=x, 'y', 'n'=y, 'z', 't'=z, 'a', 'u'=u.
```

```
mreal Diff (char dir, mreal var[26]) [Method on mg1Expr]
mreal mg1_expr_diff_v (HMEX ex, char dir, mreal *var) [C function]
Evaluates the formula derivation respect to dir for variables in array var[0,...,'z'-'a'].
```

## 6.13 Special data classes

This section describe special data classes `mgldataV`, `mgldataF`, `mgldataT` and `mgldataR` which sometime can noticeable speed up drawing or data handling. These classes are defined in `#include <mg12/data.h>`. Note, that all plotting and data handling routines can be done using usual `mgldata` or `mgldataC` classes. Also these special classes are usable in C++ code only.

### Class `mgldataV`

represent variable with values equidistantly distributed in given range.

```
mgldataV (const mgldataV & d) [Constructor on mgldataV]
    Copy constructor.

mgldataV (long nx=1, long ny=1, long nz=1, [Constructor on mgldataV]
    mreal v1=0, mreal v2=NaN, char dir='x')
    Create variable with "sizes" nxnynxnz which changes from v1 to v2 (or is constant
    if v2=NaN) along dir direction.

void Create (long nx=1, long ny=1, long nz=1) [Method on mgldataV]
    Set "sizes" nxnynxnz.

void Fill (mreal x1, mreal x2=NaN, char dir='x') [Method on mgldataV]
    Set ranges of the variable.

void Freq (mreal dp, char dir='x') [Method on mgldataV]
    Set as frequency variable with increment dp.
```

### Class `mgldataF`

represent function which values are evaluated (instead of access to data array as in `mgldata`).

```
mgldataF (const mgldataF & d) [Constructor on mgldataF]
    Copy constructor.

mgldataF (long nx=1, long ny=1, long nz=1) [Constructor on mgldataF]
    Create variable with "sizes" nxnynxnz with zero function.

void Create (long nx=1, long ny=1, long nz=1) [Method on mgldataF]
    Set "sizes" nxnynxnz.

void SetRanges (mg1Point p1, mg1Point p2) [Method on mgldataF]
    Set ranges for internal x,y,z variables.

void SetFormula (const char *func) [Method on mgldataF]
    Set string which will be evaluated at function calls. Note this variant is about 10
    times slower than SetFunc() one.

void SetFunc (mreal (*f)(mreal x,mreal y,mreal [Method on mgldataF]
    z,void *p), void *p=NULL)
    Set pointer to function which will be used for data.
```

## Class `mglDataT`

represent named reference to column of another data array.

```

mglDataT (const mglDataT & d)                                [Constructor on mglDataT]
    Copy constructor.

mglDataT (const mglDataA & d, long col=0)                    [Constructor on mglDataT]
    Create variable which reference col-th column of data d.

void SetInd (long col, wchar_t name)                          [Method on mglDataT]
void SetInd (long col, const wchar_t * name)                  [Method on mglDataT]
    Set reference to another column of the same data and its name.

```

## Class `mglDataR`

represent named reference to row of another data array.

```

mglDataR (const mglDataR & d)                                [Constructor on mglDataR]
    Copy constructor.

mglDataR (const mglDataA & d, long row=0)                    [Constructor on mglDataR]
    Create variable which reference row-th row of data d.

void SetInd (long row, wchar_t name)                          [Method on mglDataR]
void SetInd (long row, const wchar_t * name)                  [Method on mglDataR]
    Set reference to another row of the same data and its name.

```



## 7 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, see Section 1.6 [Utilities], page 4) or in the frame of the library using.

### 7.1 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. Command may have up to 1000 arguments (at least for now). 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 3.7 [Command options], page 153). 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 '\<br>' 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 264, 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 261, 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 6.6 [Make another data], page 264) 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 6.9 [Data information], page 273), names defined by `[define]`, page 286, 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 252, `[var]`, page 258, `[list]`, page 255, `[copy]`, page 253, `[read]`, page 261, `[hist]`, page 266, `[sum]`, page 267, and so on (see sections Section 6.2 [Data constructor], page 252, Section 6.4 [Data filling], page 255, and Section 6.6 [Make another data], page 264).

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).

## 7.2 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 4 [MathGL core], page 157, and Chapter 6 [Data processing], page 251. Note, that some of program flow commands (like `[define]`, page 286, `[ask]`, page 286, `[call]`, page 287, `[for]`, page 287, `[func]`, page 287) 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 287.

**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 288, command. See also [return], page 287.

**return** [MGL command]  
 Return from the function. See also [func], page 287.

**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).

<b>next</b>	[MGL command]
Finishes for cycle.	
<b>once val</b>	[MGL command]
The code between <b>once on</b> and <b>once off</b> will be executed only once. Useful for large data manipulation in programs like UDAV.	
<b>stop</b>	[MGL command]
Terminate execution.	
<b>rkstep eq1;... var1;... [dt=1]</b>	[MGL command]
Make one step for ordinary differential equation(s) {var1' = eq1, ... } with time-step <i>dt</i> . Here variable(s) 'var1', ... are the ones, defined in MGL script previously. The Runge-Kutta 4-th order method is used for solution.	

### 7.3 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 288, 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 below).

`'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 175.

`\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.

## 7.4 mglParse class

Class for parsing and executing MGL script. This class is defined in `#include <mgl2/mgl.h>`.

The main function of `mglParse` class is `Execute()`. Exactly this function parses and executes the script string-by-string. Also there are subservient functions for the finding and creation of a variable (object derived from `mglDataA`). These functions can be useful for displaying values of variables (arrays) in some external object (like, window) or for providing access to internal data. Function `AllowSetSize()` allows one to prevent changing the size of the picture inside the script (forbids the MGL command `setsize`).

<code>mglParse (bool setsize=false)</code>	[Constructor on <code>mglParse</code> ]
<code>mglParse (HMPR pr)</code>	[Constructor on <code>mglParse</code> ]
<code>mglParse (mglParse &amp;pr)</code>	[Constructor on <code>mglParse</code> ]
<code>HMPR mgl_create_parser ()</code>	[C function]

Constructor initializes all values with zero and set `AllowSetSize` value.

```

~mglParse () [Destructor on mglParse]
void mgl_delete_parser (HMPR p) [C function]
    Destructor delete parser

HMPR Self () [Method on mglParse]
    Returns the pointer to internal object of type HMPR.

void Execute (mglGraph *gr, const char *text) [Method on mglParse]
void Execute (mglGraph *gr, const wchar_t *text) [Method on mglParse]
void mgl_parse_text (HMGL gr, HMPR p, const char *text) [C function]
void mgl_parse_textw (HMGL gr, HMPR p, const wchar_t *text) [C function]
    Main function in the class. Function parse and execute line-by-line MGL script in
    array text. Lines are separated by newline symbol '\n' as usual.

void Execute (mglGraph *gr, FILE *fp, bool [Method on mglParse]
    print=false)
void mgl_parse_file (HMGL gr, HMPR p, FILE *fp, int print) [C function]
    The same as previous but read script from the file fp. If print=true then all warnings
    and information will be printed in stdout.

int Parse (mglGraph *gr, const char *str, long pos=0) [Method on mglParse]
int Parse (mglGraph *gr, const wchar_t *str, long [Method on mglParse]
    pos=0)
int mgl_parse_line (HMGL gr, HMPR p, const char *str, int pos) [C function]
int mgl_parse_linew (HMGL gr, HMPR p, const wchar_t *str, int [C function]
    pos)
    Function parses the string str and executes it by using gr as a graphics plotter.
    Returns the value depending on an error presence in the string str: 0 – no error, 1
    – wrong command argument(s), 2 – unknown command, 3 – string is too long, 4 –
    strings is not closed. Optional argument pos allows to save the string position in the
    document (or file) for using for|next command.

mglData Calc (const char *formula) [Method on mglParse]
mglData Calc (const wchar_t *formula) [Method on mglParse]
HMDT mgl_parser_calc (HMPR p, const char *formula) [C function]
HMDT mgl_parser_calcw (HMPR p, const wchar_t *formula) [C function]
    Function parses the string formula and return resulting data array. In difference to
    AddVar() or FindVar(), it is usual data array which should be deleted after usage.

mglDataC CalcComplex (const char *formula) [Method on mglParse]
mglDataC CalcComplex (const wchar_t *formula) [Method on mglParse]
HADT mgl_parser_calc_complex (HMPR p, const char *formula) [C function]
HADT mgl_parser_calc_complexw (HMPR p, const wchar_t [C function]
    *formula)
    Function parses the string formula and return resulting data array with complex
    values. In difference to AddVar() or FindVar(), it is usual data array which should
    be deleted after usage.

void AddParam (int n, const char *str) [Method on mglParse]
void AddParam (int n, const wchar_t *str) [Method on mglParse]

```



```
void mgl_parser_add_param (HMPR p, int id, const char *val)      [C function]
void mgl_parser_add_paramw (HMPR p, int id, const wchar_t
    *val)                                                        [C function]
```

Function set the value of  $n$ -th parameter as string *str* ( $n=0, 1 \dots 'z'-'a'+10$ ). String *str* shouldn't contain '\$' symbol.

```
mglVar * FindVar (const char *name)                            [Method on mglParse]
mglVar * FindVar (const wchar_t *name)                        [Method on mglParse]
HMDT mgl_parser_find_var (HMPR p, const char *name)          [C function]
HMDT mgl_parser_find_varw (HMPR p, const wchar_t *name)       [C function]
```

Function returns the pointer to variable with name *name* or zero if variable is absent. Use this function to put external data array to the script or get the data from the script. You must **not delete** obtained data arrays!

```
mglVar * AddVar (const char *name)                            [Method on mglParse]
mglVar * AddVar (const wchar_t *name)                        [Method on mglParse]
HMDT mgl_parser_add_var (HMPR p, const char *name)           [C function]
HMDT mgl_parser_add_varw (HMPR p, const wchar_t *name)       [C function]
```

Function returns the pointer to variable with name *name*. If variable is absent then new variable is created with name *name*. Use this function to put external data array to the script or get the data from the script. You must **not delete** obtained data arrays!

```
void DeleteVar (const char *name)                            [Method on mglParse (C++)]
void DeleteVar (const wchar_t *name)                        [Method on mglParse (C++)]
void mgl_parser_del_var (HMPR p, const char *name)           [C function]
void mgl_parser_del_varw (HMPR p, const wchar_t *name)       [C function]
```

Function delete the variable with given *name*.

```
void DeleteAll ()                                            [Method on mglParse (C++)]
void mgl_parser_del_all (HMPR p)                             [C function]
```

Function delete all variables and reset list of commands to default one in this parser.

```
void RestoreOnce ()                                         [Method on mglParse]
void mgl_parser_restore_once (HMPR p)                       [C function]
```

Restore Once flag.

```
void AllowSetSize (bool a)                                   [Method on mglParse]
void mgl_parser_allow_setsize (HMPR p, int a)               [C function]
```

Allow to parse [setsize], page 175, command or not.

```
void AllowFileIO (bool a)                                   [Method on mglParse]
void mgl_parser_allow_file_io (HMPR p, int a)               [C function]
```

Allow reading/saving files or not.

```
void AllowDllCall (bool a)                                   [Method on mglParse]
void mgl_parser_allow_dll_call (HMPR p, int a)              [C function]
```

Allow to parse [load], page 287, command or not.

```

void Stop () [Method on mglParse]
void mgl_parser_stop (HMPR p) [C function]
    Sends stop signal which terminate execution at next command.

long GetCmdNum () [Method on mglParse]
long mgl_parser_cmd_num (HMPR p) [C function]
    Return the number of registered MGL commands.

const char * GetCmdName (long id) [Method on mglParse]
const char * mgl_parser_cmd_name (HMPR p, long id) [C function]
    Return the name of command with given id.

int CmdType (const char *name) [Method on mglParse]
int mgl_parser_cmd_type (HMPR p, const char *name) [C function]
    Return the type of MGL command name. Type of commands are: 0 – not the
    command, 1 - data plot, 2 - other plot, 3 - setup, 4 - data handle, 5 - data create, 6
    - subplot, 7 - program, 8 - 1d plot, 9 - 2d plot, 10 - 3d plot, 11 - dd plot, 12 - vector
    plot, 13 - axis, 14 - primitives, 15 - axis setup, 16 - text/legend, 17 - data transform.

const char * CmdFormat (const char *name) [Method on mglParse]
const char * mgl_parser_cmd_frmt (HMPR p, const char *name) [C function]
    Return the format of arguments for MGL command name.

const char * CmdDesc (const char *name) [Method on mglParse]
const char * mgl_parser_cmd_desc (HMPR p, const char *name) [C function]
    Return the description of MGL command name.

void RK_Step (const char *eqs, const char *vars, [Method on mglParse]
             mreal dt=1)
void RK_Step (const wchar_t *eqs, const wchar_t [Method on mglParse]
             *vars, mreal dt=1)
void mgl_rk_step (HMPR p, const char *eqs, const char *vars, [C function]
                mreal dt)
void mgl_rk_step_w (HMPR p, const wchar_t *eqs, const [C function]
                   wchar_t *vars, mreal dt)
    Make one step for ordinary differential equation(s) {var1' = eq1, ... } with time-step
    dt. Here strings eqs and vars contain the equations and variable names separated by
    symbol ';'. The variable(s) 'var1', ... are the ones, defined in MGL script previously.
    The Runge-Kutta 4-th order method is used.

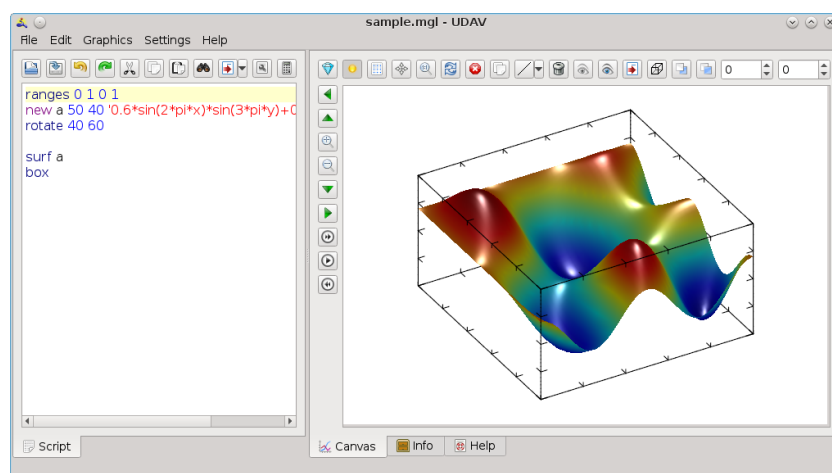
```

## 8 UDAV

UDAV (Universal Data Array Visualizer) is cross-platform program for data arrays visualization based on MathGL library. It support wide spectrum of graphics, simple script language and visual data handling and editing. It has window interface for data viewing, changing and plotting. Also it can execute MGL scripts, setup and rotate graphics and so on. UDAV hot-keys can be found in the appendix Section A.3 [Hot-keys for UDAV], page 322.

### 8.1 UDAV overview

UDAV have main window divided by 2 parts in general case and optional bottom panel(s). Left side contain tabs for MGL script and data arrays. Right side contain tabs with graphics itself, with list of variables and with help on MGL. Bottom side may contain the panel with MGL messages and warnings, and the panel with calculator.



Main window is shown on the figure above. You can see the script (at left) with current line highlighted by light-yellow, and result of its execution at right. Each panel have its own set of toolbuttons.

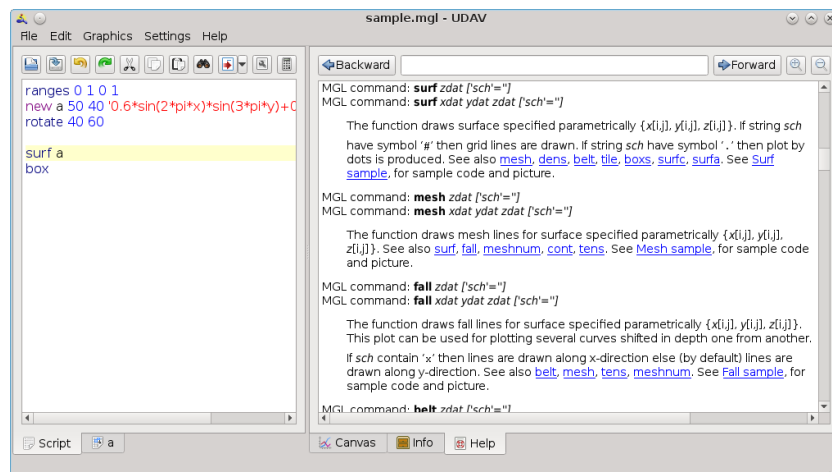
Editor toolbuttons allow: open and save script from/to file; undo and redo changes; cut, copy and paste selection; find/replace text; show dialogs for command arguments and for plot setup; show calculator at bottom.

Graphics toolbuttons allow: enable/disable additional transparency and lighting; show grid of absolute coordinates; enable mouse rotation; restore image view; refresh graphics (execute the script); stop calculation; copy graphics into clipboard; add primitives (line, curve, box, rhombus, ellipse, mark, text) to the image; change view angles manually. Vertical toolbuttons allow: shift and zoom in/out of image as whole; show next and previous frame of animation, or start animation (if one present).

Graphics panel support plot editing by mouse.

- Axis range can be changed by mouse wheel or by dragging image by middle mouse button. Right button show popup menu. Left button show the coordinates of mouse click. At this double click will highlight plot under mouse and jump to the corresponded string of the MGL script.

- Pressing "mouse rotation" toolbutton will change mouse actions: dragging by left button will rotate plot, middle button will shift the plot as whole, right button will zoom in/out plot as whole and add perspective, mouse wheel will zoom in/out plot as whole.
- Manual primitives can be added by pressing corresponding toolbutton. They can be shifted as whole at any time by mouse dragging. At this double click open dialog with its properties. If toolbutton "grid of absolute coordinates" is pressed then editing of active points for primitives is enabled.



Short command description and list of its arguments are shown at the status-bar, when you move cursor to the new line of code. You can press F1 to see more detailed help on special panel.



Also you can look the current list of variables, its dimensions and its size in the memory (right side of above figure). Toolbuttons allow: create new variable, edit variable, delete variable, preview variable plot and its properties, refresh list of variables. Pressing on any column will sort table according its contents. Double click on a variable will open panel with data cells of the variable, where you can view/edit each cell independently or apply a set of transformations.



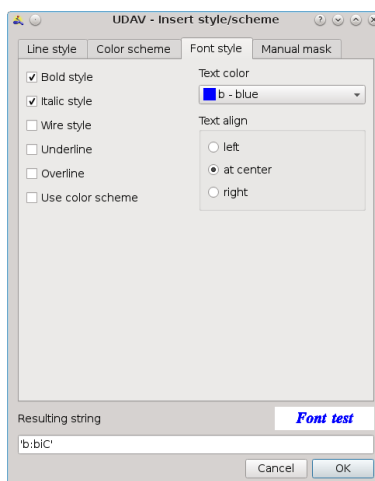
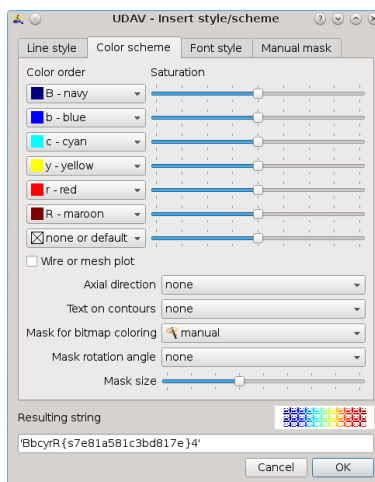
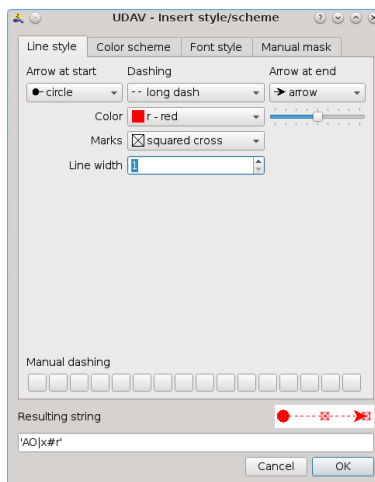
Finally, pressing F2 or F4 you can show/hide windows with messages/warnings and with calculator. Double click on a warning in message window will jump to corresponding line in editor. Calculator allow you type expression by keyboard as well as by toolbuttons. It know about all current variables, so you can use them in formulas.

## 8.2 UDAV dialogs

There are a set of dialogs, which allow change/add a command, setup global plot properties, or setup UDAV itself.



One of most interesting dialog (hotkey **Meta-C** or **Win-C**) is dialog which help to enter new command or change arguments of existed one. It allows consequently select the category of command, command name in category and appropriate set of command arguments. At this right side show detailed command description. Required argument(s) are denoted by bold text. Strings are placed in apostrophes, like '**txt**'. Buttons below table allow to call dialogs for changing style of command (if argument '**fmt**' is present in the list of command arguments); to set variable or expression for argument(s); to add options for command. Note, you can click on a cell to enter value, or double-click to call corresponding dialog.





Dialog for changing style can be called independently, but usually is called from *New command* dialog or by double click on primitive. It contain 3 tabs: one for pen style, one for color scheme, one for text style. You should select appropriate one. Resulting string of style and sample picture are shown at bottom of dialog. Usually it can be called from New command dialog.



Dialog for entering variable allow to select variable or expression which can be used as argument of a command. Here you can select the variable name; range of indexes in each directions; operation which will be applied (like, summation, finding minimal/maximal values and so on). Usually it can be called from New command dialog.



Dialog for command options allow to change Section 3.7 [Command options], page 153. Usually it can be called from New command dialog.



Another interesting dialog, which help to select and properly setup a [subplot], page 172, [inplot], page 173, [columnplot], page 173, [stickplot], page 173, and similar commands.



There is dialog for setting general plot properties, including tab for setting lighting properties. It can be called by hotkey ??? and put setup commands at the beginning of MGL script.





Also you can set or change script parameters ('\$0' ... '\$9', see Section 7.1 [MGL definition], page 285).



Finally, there is dialog for UDAV settings. It allow to change most of things in UDAV appearance and working, including colors of keywords and numbers, default font and image size, and so on (see figure above).

There are also a set of dialogs for data handling, but they are too simple and clear. So, I will not put them here.

### 8.3 UDAV hints

- You can shift axis range by pressing middle button and moving mouse. Also, you can zoom in/out axis range by using mouse wheel.
- You can rotate/shift/zoom whole plot by mouse. Just press 'Rotate' toolbutton, click image and hold a mouse button: left button for rotation, right button for zoom/perspective, middle button for shift.
- You may quickly draw the data from file. Just use: `udav 'filename.dat'` in command line.
- You can copy the current image to clipboard by pressing **Ctrl-Shift-C**. Later you can paste it directly into yours document or presentation.
- You can export image into a set of format (EPS, SVG, PNG, JPEG) by pressing right mouse button inside image and selecting 'Export as ...'.

- You can setup colors for script highlighting in Property dialog. Just select menu item 'Settings/Properties'.
- You can save the parameter of animation inside MGL script by using comment started from '##a ' or '##c ' for loops.
- New drawing never clears things drawn already. For example, you can make a surface with contour lines by calling commands 'surf' and 'cont' one after another (in any order).
- You can put several plots in the same image by help of commands 'subplot' or 'inplot'.
- All indexes (of data arrays, subplots and so on) are always start from 0.
- You can edit MGL file in any text editor. Also you can run it in console by help of commands: `mglconv`, `mglview`.
- You can use command 'once on|off' for marking the block which should be executed only once. For example, this can be the block of large data reading/creating/handling. Press F9 (or menu item 'Graphics/Reload') to re-execute this block.
- You can use command 'stop' for terminating script parsing. It is useful if you don't want to execute a part of script.
- You can type arbitrary expression as input argument for data or number. In last case (for numbers), the first value of data array is used.
- There is powerful calculator with a lot of special functions. You can use buttons or keyboard to type the expression. Also you can use existed variables in the expression.
- The calculator can help you to put complex expression in the script. Just type the expression (which may depend on coordinates x,y,z and so on) and put it into the script.
- You can easily insert file or folder names, last fitted formula or numerical value of selection by using menu Edit|Insert.
- The special dialog (Edit|Insert|New Command) help you select the command, fill its arguments and put it into the script.
- You can put several plotting commands in the same line or in separate function, for highlighting all of them simultaneously.

## 9 Other classes

There are few end-user classes: `mglGraph` (see Chapter 4 [MathGL core], page 157), `mglWindow` and `mglGLUT` (see Chapter 5 [Widget classes], page 237), `mglData` (see Chapter 6 [Data processing], page 251), `mglParse` (see Chapter 7 [MGL scripts], page 285). Exactly these classes I recommend to use in most of user programs. All methods in all of these classes are inline and have exact C/Fortran analogue functions. This give compiler independent binary libraries for MathGL.

However, sometimes you may need to extend MathGL by writing yours own plotting functions or handling yours own data structures. In these cases you may need to use low-level API. This chapter describes it.



The internal structure of MathGL is rather complicated. There are C++ classes `mglBase`, `mglCanvas`, ... for drawing primitives and positioning the plot (blue ones in the figure). There is a layer of C functions, which include interface for most important methods of these classes. Also most of plotting functions are implemented as C functions. After it, there are “inline” front-end classes which are created for user convenience (yellow ones in the figure). Also there are widgets for FLTK and Qt libraries (green ones in the figure).

Below I show how this internal classes can be used.

### 9.1 Define new kind of plot (`mglBase` class)

Basically most of new kinds of plot can be created using just MathGL primitives (see Section 4.7 [Primitives], page 184). However the usage of `mglBase` methods can give you higher speed of drawing and better control of plot settings.

All plotting functions should use a pointer to `mglBase` class (or `HMGL` type in C functions) due to compatibility issues. Exactly such type of pointers are used in front-end classes (`mglGraph`, `mglWindow`) and in widgets (`QMathGL`, `Fl_MathGL`).

MathGL tries to remember all vertexes and all primitives and plot creation stage, and to use them for making final picture by demand. Basically for making plot, you need to add vertexes by `AddPnt()` function, which return index for new vertex, and call one of primitive drawing function (like `mark_plot()`, `arrow_plot()`, `line_plot()`, `trig_plot()`, `quad_plot()`, `text_plot()`), using vertex indexes as argument(s). `AddPnt()` function use 2 mreal numbers for color specification. First one is positioning in textures – integer part is texture index, fractional part is relative coordinate in the texture. Second number is like a transparency of plot (or second coordinate in the 2D texture).

I don't want to put here detailed description of `mglBase` class. It was rather well documented in `mgl2/base.h` file. I just show an example of its usage on the base of circle drawing.

First, we should prototype new function `circle()` as C function.

```
#ifdef __cplusplus
extern "C" {
#endif
void circle(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *opt);
#ifdef __cplusplus
}
#endif
```

This is done for generating compiler independent binary. Because only C-functions have standard naming mechanism, the same for any compilers.

Now, we create a C++ file and put the code of function. I'll write it line by line and try to comment all important points.

```
void circle(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *opt)
{
```

First, we need to check all input arguments and send warnings if something is wrong. In our case it is negative value of `r` argument. We just send warning, since it is not critical situation – other plot still can be drawn.

```
    if(r<=0) { gr->SetWarn(mglWarnNeg,"Circle"); return; }
```

Next step is creating a group. Group keep some general setting for plot (like options) and useful for export in 3d files.

```
    static int cgid=1; gr->StartGroup("Circle",cgid++);
```

Now let apply options. Options are rather useful things, generally, which allow one easily redefine axis range(s), transparency and other settings (see Section 3.7 [Command options], page 153).

```
    gr->SaveState(opt);
```

I use global setting for determining the number of points in circle approximation. Note, that user can change `MeshNum` by options easily.

```
    const int n = gr->MeshNum>1?gr->MeshNum : 41;
```

Let try to determine plot specific flags. MathGL functions expect that most of flags will be sent in string. In our case it is symbol '@' which set to draw filled circle instead of border only (last will be default). Note, you have to handle NULL as string pointer.

```
bool fill = mglchr(stl, '@');
```

Now, time for coloring. I use palette mechanism because circle have few colors: one for filling and another for border. `SetPenPal()` function parse input string and write resulting texture index in *pal*. Function return the character for marker, which can be specified in string *str*. Marker will be plotted at the center of circle. I'll show on next sample how you can use color schemes (smooth colors) too.

```
long pal=0;
char mk=gr->SetPenPal(stl, &pal);
```

Next step, is determining colors for filling and for border. First one for filling.

```
mreal c=gr->NextColor(pal), d;
```

Second one for border. I use black color (call `gr->AddTexture('k')`) if second color is not specified.

```
mreal k=(gr->GetNumPal(pal)>1)?gr->NextColor(pal):gr->AddTexture('k');
```

If user want draw only border (`fill=false`) then I use first color for border.

```
if(!fill) k=c;
```

Now we should reserve space for vertexes. This functions need *n* for border, *n+1* for filling and 1 for marker. So, maximal number of vertexes is  $2*n+2$ . Note, that such reservation is not required for normal work but can sufficiently speed up the plotting.

```
gr->Reserve(2*n+2);
```

We've done with setup and ready to start drawing. First, we need to add vertex(es). Let define NAN as normals, since I don't want handle lighting for this plot,

```
mglPoint q(NAN, NAN);
```

and start adding vertexes. First one for central point of filling. I use -1 if I don't need this point. The arguments of `AddPnt()` function is: `mglPoint(x,y,z)` – coordinate of vertex, *c* – vertex color, *q* – normal at vertex, -1 – vertex transparency (-1 for default), 3 bitwise flag which show that coordinates will be scaled (0x1) and will not be cutted (0x2).

```
long n0, n1, n2, m1, m2, i;
n0 = fill ? gr->AddPnt(mglPoint(x,y,z), c, q, -1, 3) : -1;
```

Similar for marker, but we use different color *k*.

```
n2 = mk ? gr->AddPnt(mglPoint(x,y,z), k, q, -1, 3) : -1;
```

Draw marker.

```
if(mk) gr->mark_plot(n2, mk);
```

Time for drawing circle itself. I use -1 for *m1*, *n1* as sign that primitives shouldn't be drawn for first point *i=0*.

```
for(i=0, m1=n1=-1; i<n; i++)
{
```

Each function should check `Stop` variable and return if it is non-zero. It is done for interrupting drawing for system which don't support multi-threading.

```
if(gr->Stop) return;
```

Let find coordinates of vertex.

```
mreal t = i*2*M_PI/(n-1.);
mglPoint p(x+r*cos(t), y+r*sin(t), z);
```

Save previous vertex and add next one

```
n2 = n1; n1 = gr->AddPnt(p,c,q,-1,3);
```

and copy it for border but with different color. Such copying is much faster than adding new vertex using `AddPnt()`.

```
m2 = m1; m1 = gr->CopyNtoC(n1,k);
```

Now draw triangle for filling internal part

```
if(fill) gr->trig_plot(n0,n1,n2);
```

and draw line for border.

```
gr->line_plot(m1,m2);
```

```
}
```

Drawing is done. Let close group and return.

```
gr->EndGroup();
}
```

Another sample I want to show is exactly the same function but with smooth coloring using color scheme. So, I'll add comments only in the place of difference.

```
void circle_cs(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *op)
{
```

In this case let allow negative radius too. Formally it is not the problem for plotting (formulas the same) and this allow us to handle all color range.

```
//if(r<=0) { gr->SetWarn(mglWarnNeg,"Circle"); return; }
```

```
static int cgid=1; gr->StartGroup("CircleCS",cgid++);
gr->SaveState(opt);
const int n = gr->MeshNum>1?gr->MeshNum : 41;
bool fill = mglchr(stl,'@');
```

Here is main difference. We need to create texture for color scheme specified by user

```
long ss = gr->AddTexture(stl);
```

But we need also get marker and color for it (if filling is enabled). Let suppose that marker and color is specified after `':'`. This is standard delimiter which stop color scheme entering. So, just lets find it and use for setting pen.

```
const char *pen=0;
if(stl) pen = strchr(stl,':');
if(pen) pen++;
```

The substring is placed in `pen` and it will be used as line style.

```
long pal=0;
char mk=gr->SetPenPal(pen,&pal);
```

Next step, is determining colors for filling and for border. First one for filling.

```
mreal c=gr->GetC(ss,r);
```

Second one for border.

```
mreal k=gr->NextColor(pal);
```

The rest part is the same as in previous function.

```
if(!fill) k=c;

gr->Reserve(2*n+2);
mglPoint q(NAN,NAN);
long n0,n1,n2,m1,m2,i;
n0 = fill ? gr->AddPnt(mglPoint(x,y,z),c,q,-1,3):-1;
n2 = mk ? gr->AddPnt(mglPoint(x,y,z),k,q,-1,3):-1;
if(mk) gr->mark_plot(n2,mk);
for(i=0,m1=n1=-1;i<n;i++)
{
    if(gr->Stop) return;
    mreal t = i*2*M_PI/(n-1.);
    mglPoint p(x+r*cos(t), y+r*sin(t), z);
    n2 = n1; n1 = gr->AddPnt(p,c,q,-1,3);
    m2 = m1; m1 = gr->CopyNtoC(n1,k);
    if(fill) gr->trig_plot(n0,n1,n2);
    gr->line_plot(m1,m2);
}
gr->EndGroup();
}
```

The last thing which we can do is derive our own class with new plotting functions. Good idea is to derive it from `mglGraph` (if you don't need extended window), or from `mglWindow` (if you need to extend window). So, in our case it will be

```
class MyGraph : public mglGraph
{
public:
    inline void Circle(mglPoint p, mreal r, const char *stl="", const char *opt="")█
    { circle(p.x,p.y,p.z, r, stl, opt); }
    inline void CircleCS(mglPoint p, mreal r, const char *stl="", const char *opt="")█
    { circle_cs(p.x,p.y,p.z, r, stl, opt); }
};
```

Note, that I use `inline` modifier for using the same binary code with different compilers.

So, the complete sample will be

```
#include <mgl2/mgl.h>
//-----
#ifdef __cplusplus
extern "C" {
#endif
void circle(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *opt);
void circle_cs(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *op
#ifdef __cplusplus
}
```

```

#endif
//-----
class MyGraph : public mglGraph
{
public:
    inline void CircleCF(mglPoint p, mreal r, const char *stl="", const char *opt="")
    { circle(p.x,p.y,p.z, r, stl, opt); }
    inline void CircleCS(mglPoint p, mreal r, const char *stl="", const char *opt="")
    { circle_cs(p.x,p.y,p.z, r, stl, opt); }
};
//-----
void circle(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *opt)
{
    if(r<=0) { gr->SetWarn(mglWarnNeg,"Circle"); return; }
    static int cgid=1; gr->StartGroup("Circle",cgid++);
    gr->SaveState(opt);
    const int n = gr->MeshNum>1?gr->MeshNum : 41;
    bool fill = mglchr(stl,'@');
    long pal=0;
    char mk=gr->SetPenPal(stl,&pal);
    mreal c=gr->NextColor(pal), d;
    mreal k=(gr->GetNumPal(pal)>1)?gr->NextColor(pal):gr->AddTexture('k');
    if(!fill) k=c;
    gr->Reserve(2*n+2);
    mglPoint q(NAN,NAN);
    long n0,n1,n2,m1,m2,i;
    n0 = fill ? gr->AddPnt(mglPoint(x,y,z),c,q,-1,3):-1;
    n2 = mk ? gr->AddPnt(mglPoint(x,y,z),k,q,-1,3):-1;
    if(mk) gr->mark_plot(n2,mk);
    for(i=0,m1=n1=-1;i<n;i++)
    {
        if(gr->Stop) return;
        mreal t = i*2*M_PI/(n-1.);
        mglPoint p(x+r*cos(t), y+r*sin(t), z);
        n2 = n1; n1 = gr->AddPnt(p,c,q,-1,3);
        m2 = m1; m1 = gr->CopyNtoC(n1,k);
        if(fill) gr->trig_plot(n0,n1,n2);
        gr->line_plot(m1,m2);
    }
    gr->EndGroup();
}
//-----
void circle_cs(HMGL gr, mreal x, mreal y, mreal z, mreal r, const char *stl, const char *opt)
{
    static int cgid=1; gr->StartGroup("CircleCS",cgid++);
    gr->SaveState(opt);
    const int n = gr->MeshNum>1?gr->MeshNum : 41;

```



```

    bool fill = mglchr(stl, '@');
    long ss = gr->AddTexture(stl);
    const char *pen=0;
    if(stl) pen = strchr(stl, ':');
    if(pen) pen++;
    long pal=0;
    char mk=gr->SetPenPal(pen, &pal);
    mreal c=gr->GetC(ss, r);
    mreal k=gr->NextColor(pal);
    if(!fill) k=c;

    gr->Reserve(2*n+2);
    mglPoint q(NAN, NAN);
    long n0, n1, n2, m1, m2, i;
    n0 = fill ? gr->AddPnt(mglPoint(x, y, z), c, q, -1, 3) : -1;
    n2 = mk ? gr->AddPnt(mglPoint(x, y, z), k, q, -1, 3) : -1;
    if(mk) gr->mark_plot(n2, mk);
    for(i=0, m1=n1=-1; i<n; i++)
    {
        if(gr->Stop) return;
        mreal t = i*2*M_PI/(n-1.);
        mglPoint p(x+r*cos(t), y+r*sin(t), z);
        n2 = n1; n1 = gr->AddPnt(p, c, q, -1, 3);
        m2 = m1; m1 = gr->CopyNtoC(n1, k);
        if(fill) gr->trig_plot(n0, n1, n2);
        gr->line_plot(m1, m2);
    }
    gr->EndGroup();
}
//-----
int main()
{
    MyGraph gr;
    gr.Box();
    // first let draw circles with fixed colors
    for(int i=0; i<10; i++)
        gr.CircleCF(mglPoint(2*mgl_rnd()-1, 2*mgl_rnd()-1), mgl_rnd());
    // now let draw circles with color scheme
    for(int i=0; i<10; i++)
        gr.CircleCS(mglPoint(2*mgl_rnd()-1, 2*mgl_rnd()-1), 2*mgl_rnd()-1);
}

```

## 9.2 User defined types (mglDataA class)

`mglData` class have abstract predecessor class `mglDataA`. Exactly the pointers to `mglDataA` instances are used in all plotting functions and some of data processing functions. This was done for taking possibility to define yours own class, which will handle yours own data

(for example, complex numbers, or differently organized data). And this new class will be almost the same as `mglData` for plotting purposes.

However, the most of data processing functions will be slower as if you used `mglData` instance. This is more or less understandable – I don't know how data in yours particular class will be organized, and couldn't optimize the these functions generally.

There are few virtual functions which must be provided in derived classes. This functions give:

- the sizes of the data (`GetNx`, `GetNy`, `GetNz`),
- give data value and numerical derivatives for selected cell (`v`, `dvx`, `dvy`, `dvz`),
- give maximal and minimal values (`Maximal`, `Minimal`) – you can use provided functions (like `mgl_data_max` and `mgl_data_min`), but yours own realization can be more efficient,
- give access to all element as in single array (`vthr`) – you need this only if you want using MathGL's data processing functions.

Let me, for example define class `mglComplex` which will handle complex number and draw its amplitude or phase, depending on flag `use_abs`:

```
#include <complex>
#include <mgl2/mgl.h>
#define dual std::complex<double>
class mglComplex : public mglDataA
{
public:
    long nx;          ///< number of points in 1st dimensions ('x' dimension)
    long ny;          ///< number of points in 2nd dimensions ('y' dimension)
    long nz;          ///< number of points in 3d dimensions ('z' dimension)
    dual *a;          ///< data array
    bool use_abs;      ///< flag to use abs() or arg()

    inline mglComplex(long xx=1,long yy=1,long zz=1)
    { a=0; use_abs=true; Create(xx,yy,zz); }
    virtual ~mglComplex() { if(a) delete []a; }

    /// Get sizes
    inline long GetNx() const { return nx; }
    inline long GetNy() const { return ny; }
    inline long GetNz() const { return nz; }
    /// Create or recreate the array with specified size and fill it by zero
    inline void Create(long mx,long my=1,long mz=1)
    { nx=mx; ny=my; nz=mz; if(a) delete []a;
      a = new dual[nx*ny*nz]; }
    /// Get maximal value of the data
    inline mreal Maximal() const { return mgl_data_max(this); }
    /// Get minimal value of the data
    inline mreal Minimal() const { return mgl_data_min(this); }
```

```

protected:
    inline mreal v(long i,long j=0,long k=0) const
    { return use_abs ? abs(a[i+nx*(j+ny*k)]) : arg(a[i+nx*(j+ny*k)]); }
    inline mreal vthr(long i) const
    { return use_abs ? abs(a[i]) : arg(a[i]); }
    inline mreal dvx(long i,long j=0,long k=0) const
    { register long i0=i+nx*(j+ny*k);
      std::complex<double> res=i>0? (i<nx-1? (a[i0+1]-a[i0-1])/2.:a[i0]-a[i0-1]) : a[i0+1]-a[i0-1];
      return use_abs? abs(res) : arg(res); }
    inline mreal dvx(long i,long j=0,long k=0) const
    { register long i0=i+nx*(j+ny*k);
      std::complex<double> res=j>0? (j<ny-1? (a[i0+nx]-a[i0-nx])/2.:a[i0]-a[i0-nx]) : a[i0+nx]-a[i0-nx];
      return use_abs? abs(res) : arg(res); }
    inline mreal dvz(long i,long j=0,long k=0) const
    { register long i0=i+nx*(j+ny*k), n=nx*ny;
      std::complex<double> res=k>0? (k<nz-1? (a[i0+n]-a[i0-n])/2.:a[i0]-a[i0-n]) : a[i0+n]-a[i0-n];
      return use_abs? abs(res) : arg(res); }
};

int main()
{
    mglComplex dat(20);
    for(long i=0;i<20;i++)
        dat.a[i] = 3*exp(-0.05*(i-10)*(i-10))*dual(cos(M_PI*i*0.3), sin(M_PI*i*0.3));
    mglGraph gr;
    gr.SetRange('y', -M_PI, M_PI); gr.Box();

    gr.Plot(dat,"r","legend 'abs'");
    dat.use_abs=false;
    gr.Plot(dat,"b","legend 'arg'");
    gr.Legend();
    gr.WritePNG("complex.png");
    return 0;
}

```

### 9.3 mglColor class

Structure for working with colors. This structure is defined in `#include <mgl2/type.h>`.

There are two ways to set the color in MathGL. First one is using of mreal values of red, green and blue channels for precise color definition. The second way is the using of character id. There are a set of characters specifying frequently used colors. Normally capital letter gives more dark color than lowercase one. See Section 3.3 [Line styles], page 147.

**mreal r, g, b, a** [Parameter of mglColor]

Reg, green and blue component of color.

**mglColor (mreal R, mreal G, mreal B, mreal A=1)** [Method on mglColor]

Constructor sets the color by mreal values of Red, Green, Blue and Alpha channels. These values should be in interval [0,1].

<code>mglColor (char c='k', mreal bright=1)</code>	[Method on <code>mglColor</code> ]
Constructor sets the color from character id. The black color is used by default. Parameter <i>br</i> set additional “lightness” of color.	
<code>void Set (mreal R, mreal G, mreal B, mreal A=1)</code>	[Method on <code>mglColor</code> ]
Sets color from values of Red, Green, Blue and Alpha channels. These values should be in interval [0,1].	
<code>void Set (mglColor c, mreal bright=1)</code>	[Method on <code>mglColor</code> ]
Sets color as “lighted” version of color <i>c</i> .	
<code>void Set (char p, mreal bright=1)</code>	[Method on <code>mglColor</code> ]
Sets color from symbolic id.	
<code>bool Valid ()</code>	[Method on <code>mglColor</code> ]
Checks correctness of the color.	
<code>mreal Norm ()</code>	[Method on <code>mglColor</code> ]
Gets maximal of spectral component.	
<code>bool operator== (const mglColor &amp;c)</code>	[Method on <code>mglColor</code> ]
<code>bool operator!= (const mglColor &amp;c)</code>	[Method on <code>mglColor</code> ]
Compare with another color	
<code>bool operator*= (mreal v)</code>	[Method on <code>mglColor</code> ]
Multiplies color components by number <i>v</i> .	
<code>bool operator+= (const mglColor &amp;c)</code>	[Method on <code>mglColor</code> ]
Adds color <i>c</i> component by component.	
<code>bool operator-= (const mglColor &amp;c)</code>	[Method on <code>mglColor</code> ]
Subtracts color <i>c</i> component by component.	
<code>mglColor operator+ (const mglColor &amp;a, const mglColor &amp;b)</code>	[Library Function]
Adds colors by its RGB values.	
<code>mglColor operator- (const mglColor &amp;a, const mglColor &amp;b)</code>	[Library Function]
Subtracts colors by its RGB values.	
<code>mglColor operator* (const mglColor &amp;a, mreal b)</code>	[Library Function]
<code>mglColor operator* (mreal a, const mglColor &amp;b)</code>	[Library Function]
Multiplies color by number.	
<code>mglColor operator/ (const mglColor &amp;a, mreal b)</code>	[Library Function]
Divide color by number.	
<code>mglColor operator! (const mglColor &amp;a)</code>	[Library Function]
Return inverted color.	

## 9.4 mglPoint class

Structure describes point in space. This structure is defined in `#include <mgl2/type.h>`

`mreal x, y, z, c` [Parameter of `mglPoint`]  
 Point coordinates  $\{x,y,z\}$  and one extra value  $c$  used for amplitude, transparency and so on. By default all values are zero.

`mglPoint (mreal X=0, mreal Y=0, mreal Z=0, mreal C=0)` [Method on `mglPoint`]  
 Constructor sets the color by mreal values of Red, Green, Blue and Alpha channels. These values should be in interval  $[0,1]$ .

`bool IsNaN ()` [Method on `mglPoint`]  
 Returns `true` if point contain NAN values.

`mreal norm ()` [Method on `mglPoint`]  
 Returns the norm  $\sqrt{x^2 + y^2 + z^2}$  of vector.

`void Normalize ()` [Method on `mglPoint`]  
 Normalizes vector to be unit vector.

`mreal val (int i)` [Method on `mglPoint`]  
 Returns point component:  $x$  for  $i=0$ ,  $y$  for  $i=1$ ,  $z$  for  $i=2$ ,  $c$  for  $i=3$ .

`mglPoint operator+ (const mglPoint &a, const mglPoint &b)` [Library Function]  
 Point of summation (summation of vectors).

`mglPoint operator- (const mglPoint &a, const mglPoint &b)` [Library Function]  
 Point of difference (difference of vectors).

`mglPoint operator* (mreal a, const mglPoint &b)` [Library Function]  
`mglPoint operator* (const mglPoint &a, mreal b)` [Library Function]  
 Multiplies (scale) points by number.

`mglPoint operator/ (const mglPoint &a, mreal b)` [Library Function]  
 Multiplies (scale) points by number  $1/b$ .

`mreal operator* (const mglPoint &a, const mglPoint &b)` [Library Function]  
 Scalar product of vectors.

`mglPoint operator/ (const mglPoint &a, const mglPoint &b)` [Library Function]  
 Return vector of element-by-element product.

`mglPoint operator^ (const mglPoint &a, const mglPoint &b)` [Library Function]  
 Cross-product of vectors.

<code>mglPoint operator&amp; (const mglPoint &amp;a, const mglPoint &amp;b)</code>	[Library Function]
The part of $a$ which is perpendicular to vector $b$ .	
<code>mglPoint operator  (const mglPoint &amp;a, const mglPoint &amp;b)</code>	[Library Function]
The part of $a$ which is parallel to vector $b$ .	
<code>mglPoint operator! (const mglPoint &amp;a)</code>	[Library Function]
Return vector perpendicular to vector $a$ .	
<code>mreal mgl_norm (const mglPoint &amp;a)</code>	[Library Function]
Return the norm $\sqrt{ a ^2}$ of vector $a$ .	
<code>bool operator== (const mglPoint &amp;a, const mglPoint &amp;b)</code>	[Library Function]
Return true if points are the same.	
<code>bool operator!= (const mglPoint &amp;a, const mglPoint &amp;b)</code>	[Library Function]
Return true if points are different.	

## 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.

<code>'space ' '</code>	empty line style (see Section 3.3 [Line styles], page 147); empty color in [chart], page 199.
<code>'!'</code>	set to use new color from palette for each point (not for each curve, as default) in Section 4.11 [1D plotting], page 194; set to disable ticks tuning in [axis], page 190, and [colorbar], page 191; set to draw [grid], page 192, lines at subticks coordinates too; define complex variable/expression in MGL script if placed at beginning.
<code>'#'</code>	set to use solid marks (see Section 3.3 [Line styles], page 147) or solid [error], page 201, boxes; set to draw wired plot for [axial], page 212, [surf3], page 213, [surf3a], page 218, [surf3c], page 217, [triplot], page 230, [quadplot], page 231, [area], page 196, [bars], page 198, [barh], page 198, [tube], page 205, [tape], page 196, [cone], page 186, [boxs], page 208, and draw boundary only for [circle], page 186, [ellipse], page 187, [rhomb], page 187; set to draw also mesh lines for [surf], page 207, [surfc], page 217, [surfa], page 218, [dens], page 209, [densx], page 227, [densy], page 227, [densz], page 227, [dens3], page 214, or boundary for [chart], page 199, [facex], page 185, [facey], page 185, [facez], page 185, [rect], page 185; set to draw boundary and box for [legend], page 193, [title], page 173, or grid lines for [table], page 204; set to draw grid for [radar], page 195; set to start flow threads and pipes from edges only for [flow], page 224, [pipe], page 226; set to use whole are for axis range in [subplot], page 172, [inplot], page 173; change text color inside a string (see Section 3.5 [Font styles], page 151); start comment in Chapter 7 [MGL scripts], page 285, or in Section 3.7 [Command options], page 153.
<code>'\$'</code>	denote parameter of Chapter 7 [MGL scripts], page 285.
<code>'%'</code>	set color scheme along 2 coordinates Section 3.4 [Color scheme], page 149.
<code>'&amp;'</code>	set to pass long integer number in tick template [xtick], page 169, [ytick], page 169, [ztick], page 169, [ctick], page 169. operation in Section 3.6 [Textual formulas], page 152.

- ‘,’ denote string in Chapter 7 [MGL scripts], page 285, or in Section 3.7 [Command options], page 153.
- ‘\*’ one of marks (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
operation in Section 3.6 [Textual formulas], page 152.
- ‘+’ one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes;  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
set to print ‘+’ for positive numbers in [axis], page 190, [label], page 203, [table], page 204;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘,’ separator for color positions (see Section 3.2 [Color styles], page 147) or items in a list.
- ‘-’ solid line style (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
place entries horizontally in [legend], page 193;  
set to use usual ‘-’ for negative numbers in [axis], page 190, [label], page 203, [table], page 204;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘.’ one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes;  
set to draw hachures instead of arrows for [vect], page 222, [vect3], page 223;  
set to use dots instead of faces for [cloud], page 214, [torus], page 205, [axial], page 212, [surf3], page 213, [surf3a], page 218, [surf3c], page 217, [surf], page 207, [surfa], page 218, [surfc], page 217, [dens], page 209, [map], page 221;  
delimiter of fractional parts for numbers.
- ‘/’ operation in Section 3.6 [Textual formulas], page 152.
- ‘:’ line dashing style (see Section 3.3 [Line styles], page 147);  
stop color scheme parsing (see Section 3.4 [Color scheme], page 149);  
range operation in Chapter 7 [MGL scripts], page 285.
- ‘;’ line dashing style (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
end of an option in Chapter 7 [MGL scripts], page 285, or in Section 3.7 [Command options], page 153.
- ‘<’ one of marks (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
style of [subplot], page 172, and [inplot], page 173;  
set position of [colorbar], page 191;  
style of [vect], page 222, [vect3], page 223;



- align left in [bars], page 198, [barh], page 198, [boxplot], page 199, [cones], page 199, [candle], page 200, [ohlc], page 201;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘>’ one of marks (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
style of [subplot], page 172, and [inplot], page 173;  
set position of [colorbar], page 191;  
style of [vect], page 222, [vect3], page 223;  
align right in [bars], page 198, [barh], page 198, [boxplot], page 199, [cones], page 199, [candle], page 200, [ohlc], page 201;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘=’ line dashing style (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
set to use equidistant columns for [table], page 204;  
set to use color gradient for [vect], page 222, [vect3], page 223;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘@’ set to draw box around text for [text], page 189, and similar functions;  
set to draw boundary and fill it for [circle], page 186, [ellipse], page 187, [rhomb], page 187;  
set to fill faces for [box], page 192;  
set to draw large semitransparent mark instead of error box for [error], page 201;  
set to draw edges for [cone], page 186;  
set to draw filled boxes for [boxs], page 208;  
reduce text size inside a string (see Section 3.5 [Font styles], page 151).
- ‘^’ one of marks (see Section 3.3 [Line styles], page 147);  
one of mask for face filling (see Section 3.4 [Color scheme], page 149);  
style of [subplot], page 172, and [inplot], page 173;  
set position of [colorbar], page 191;  
set outer position for [legend], page 193;  
inverse default position for [axis], page 190;  
switch to upper index inside a string (see Section 3.5 [Font styles], page 151);  
align center in [bars], page 198, [barh], page 198, [boxplot], page 199, [cones], page 199, [candle], page 200, [ohlc], page 201;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘\_’ empty arrow style (see Section 3.3 [Line styles], page 147);  
disable drawing of tick labels for [axis], page 190;  
style of [subplot], page 172, and [inplot], page 173;  
set position of [colorbar], page 191;

- set to draw contours at bottom for [cont], page 209, [contf], page 210, [contd], page 211, [contv], page 211, [tricont], page 230;  
switch to lower index inside a string (see Section 3.5 [Font styles], page 151).
- ‘[]’ contain symbols excluded from color scheme parsing (see Section 3.4 [Color scheme], page 149).
- ‘{ }’ contain extended specification of color (see Section 3.2 [Color styles], page 147),  
dashing (see Section 3.3 [Line styles], page 147) or mask (see Section 3.4 [Color scheme], page 149);  
denote special operation in Chapter 7 [MGL scripts], page 285;  
denote ‘meta-symbol’ for LaTeX like string parsing (see Section 3.5 [Font styles], page 151).
- ‘|’ line dashing style (see Section 3.3 [Line styles], page 147);  
set to use sharp color scheme (see Section 3.4 [Color scheme], page 149);  
set to limit width by subplot width for [table], page 204;  
delimiter in [list], page 255, command;  
operation in Section 3.6 [Textual formulas], page 152.
- ‘\’ string continuation symbol on next line for Chapter 7 [MGL scripts], page 285.
- ‘~’ disable drawing of tick labels for [axis], page 190, and [colorbar], page 191;  
disable first segment in [lamerey], page 206;  
one of mask for face filling (see Section 3.4 [Color scheme], page 149).
- ‘0,1,2,3,4,5,6,7,8,9’  
line width (see Section 3.3 [Line styles], page 147);  
brightness of a color (see Section 3.2 [Color styles], page 147);  
precision of numbers in [axis], page 190, [label], page 203, [table], page 204;  
kind of smoothing (for digits 1,3,5) in [smooth], page 271;  
digits for a value.
- ‘4,6,8’ set to draw square, hex- or octo-pyramids instead of cones in [cone], page 186, [cones], page 199.
- ‘A,B,C,D,E,F,a,b,c,d,e,f’  
can be hex-digit for color specification if placed inside {} (see Section 3.2 [Color styles], page 147).
- ‘A’ arrow style (see Section 3.3 [Line styles], page 147);  
set to use absolute position in whole picture for [text], page 189, [colorbar], page 191, [legend], page 193.
- ‘a’ set to use absolute position in subplot for [text], page 189;  
style of [bars], page 198, [barh], page 198, [cones], page 199.
- ‘B’ dark blue color (see Section 3.2 [Color styles], page 147).
- ‘b’ blue color (see Section 3.2 [Color styles], page 147);  
bold font face if placed after ‘:’ (see Section 3.5 [Font styles], page 151).

‘C’	dark cyan color (see Section 3.2 [Color styles], page 147); align text to center if placed after ‘:’ (see Section 3.5 [Font styles], page 151).
‘c’	cyan color (see Section 3.2 [Color styles], page 147); name of color axis; cosine transform for [transform], page 277.
‘D’	arrow style (see Section 3.3 [Line styles], page 147); one of mask for face filling (see Section 3.4 [Color scheme], page 149).
‘d’	one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes; one of mask for face filling (see Section 3.4 [Color scheme], page 149); start hex-dash description if placed inside {} (see Section 3.3 [Line styles], page 147).
‘E’	dark green-yellow color (see Section 3.2 [Color styles], page 147).
‘e’	green-yellow color (see Section 3.2 [Color styles], page 147).
‘F’	set LaTeX-like format for numbers in [axis], page 190, [label], page 203, [table], page 204.
‘f’	style of [bars], page 198, [barh], page 198; style of [vect], page 222, [vect3], page 223; set fixed format for numbers in [axis], page 190, [label], page 203, [table], page 204; Fourier transform for [transform], page 277.
‘G’	dark green color (see Section 3.2 [Color styles], page 147).
‘g’	green color (see Section 3.2 [Color styles], page 147).
‘H’	dark gray color (see Section 3.2 [Color styles], page 147).
‘h’	gray color (see Section 3.2 [Color styles], page 147); Hankel transform for [transform], page 277.
‘I’	arrow style (see Section 3.3 [Line styles], page 147); set [colorbar], page 191, position near boundary.
‘i’	line dashing style (see Section 3.3 [Line styles], page 147); italic font face if placed after ‘:’ (see Section 3.5 [Font styles], page 151). set to use inverse values for [cloud], page 214, [pipe], page 226, [dew], page 224; set to fill only area with $y_1 < y < y_2$ for [region], page 197; inverse Fourier transform for [transform], page 277.
‘j’	line dashing style (see Section 3.3 [Line styles], page 147); one of mask for face filling (see Section 3.4 [Color scheme], page 149).
‘K’	arrow style (see Section 3.3 [Line styles], page 147).

'k'	black color (see Section 3.2 [Color styles], page 147).
'L'	dark green-blue color (see Section 3.2 [Color styles], page 147); align text to left if placed after ':' (see Section 3.5 [Font styles], page 151).
'l'	green-blue color (see Section 3.2 [Color styles], page 147).
'M'	dark magenta color (see Section 3.2 [Color styles], page 147).
'm'	magenta color (see Section 3.2 [Color styles], page 147).
'N'	dark sky-blue color (see Section 3.2 [Color styles], page 147).
'n'	sky-blue color (see Section 3.2 [Color styles], page 147).
'O'	arrow style (see Section 3.3 [Line styles], page 147); one of mask for face filling (see Section 3.4 [Color scheme], page 149).
'o'	one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes; one of mask for face filling (see Section 3.4 [Color scheme], page 149); over-line text if placed after ':' (see Section 3.5 [Font styles], page 151).
'P'	dark purple color (see Section 3.2 [Color styles], page 147).
'p'	purple color (see Section 3.2 [Color styles], page 147).
'Q'	dark orange or brown color (see Section 3.2 [Color styles], page 147).
'q'	orange color (see Section 3.2 [Color styles], page 147).
'R'	dark red color (see Section 3.2 [Color styles], page 147); align text to right if placed after ':' (see Section 3.5 [Font styles], page 151).
'r'	red color (see Section 3.2 [Color styles], page 147).
'S'	arrow style (see Section 3.3 [Line styles], page 147); one of mask for face filling (see Section 3.4 [Color scheme], page 149).
's'	one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes; one of mask for face filling (see Section 3.4 [Color scheme], page 149); start hex-mask description if placed inside {} (see Section 3.4 [Color scheme], page 149); sine transform for [transform], page 277.
't'	draw tubes instead of cones in [cone], page 186, [cones], page 199;
'T'	arrow style (see Section 3.3 [Line styles], page 147); place text under the curve for [text], page 189, [cont], page 209, [cont3], page 215.
't'	set to draw text labels for [cont], page 209, [cont3], page 215; name of t-axis (one of ternary axis); variable in Section 3.6 [Textual formulas], page 152, which usually is varied in range [0,1].

‘U’	dark blue-violet color (see Section 3.2 [Color styles], page 147); disable rotation of tick labels for [axis], page 190.
‘u’	blue-violet color (see Section 3.2 [Color styles], page 147); under-line text if placed after ‘:’ (see Section 3.5 [Font styles], page 151); name of u-axis (one of ternary axis); variable in Section 3.6 [Textual formulas], page 152, which usually denote array itself.
‘V’	arrow style (see Section 3.3 [Line styles], page 147); place text centering on vertical direction for [text], page 189.
‘v’	one of marks (see Section 3.3 [Line styles], page 147); set to draw vectors on flow threads for [flow], page 224, and on segments for [lamerey], page 206.
‘W’	bright gray color (see Section 3.2 [Color styles], page 147).
‘w’	white color (see Section 3.2 [Color styles], page 147); wired text if placed after ‘:’ (see Section 3.5 [Font styles], page 151); name of w-axis (one of ternary axis);
‘X’	arrow style (see Section 3.3 [Line styles], page 147).
‘x’	one of marks (see Section 3.3 [Line styles], page 147) or kind of [error], page 201, boxes; name of x-axis or x-direction or 1st dimension of a data array; start hex-color description if placed inside {} (see Section 3.2 [Color styles], page 147); style of [tape], page 196.
‘Y’	dark yellow or gold color (see Section 3.2 [Color styles], page 147).
‘y’	yellow color (see Section 3.2 [Color styles], page 147); 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 196.

## 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 execution 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 $\theta$ .



Meta-Shift-Left, Meta-Shift-Right	Change view angle $\phi$ .
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 File formats

This appendix contain description of file formats used by MathGL.

### B.1 Font files

Starting from v.1.6 the MathGL library uses new font files. The font is defined in 4 files with suffixes ‘\*.vfm’, ‘\*\_b.vfm’, ‘\*\_i.vfm’, ‘\*\_bi.vfm’. These files are text files containing the data for roman font, bold font, italic font and bold italic font. The files (or some symbols in the files) for bold, italic or bold italic fonts can be absent. In this case the roman glyph will be used for them. By analogy, if the bold italic font is absent but the bold font is present then bold glyph will be used for bold italic. You may create these font files by yourself from \*.ttf, \*.otf files with the help of program `font_tools`. This program can be found at MathGL home site.

The format of font files (\*.vfm – vector font for MathGL) is the following.

1. First string contains human readable comment and is always ignored.
2. Second string contains 3 numbers, delimited by space or tabulation. The order of numbers is the following: *numg* – the number of glyphs in the file (integer), *fact* – the factor for glyph sizing (mreal), *size* – the size of buffer for glyph description (integer).
3. After it *numg*-th strings with glyphs description are placed. Each string contains 6 positive numbers, delimited by space or tabulation. The order of numbers is the following: Unicode glyph ID, glyph width, number of lines in glyph, position of lines coordinates in the buffer (length is 2\*number of lines), number of triangles in glyph, position of triangles coordinates in the buffer (length is 6\*number of triangles).
4. The end of file contains the buffer with point coordinates at lines or triangles vertexes. The size of buffer (the number of integer) is *size*.

Each font file can be compressed by gzip.

Note: the closing contour line is done automatically (so the last segment may be absent). For starting new contour use a point with coordinates {0x3fff, 0x3fff}.

### B.2 MGLD format

MGLD is textual file, which contain all required information for drawing 3D image, i.e. it contain vertexes with colors and normales, primitives with all properties, textures, and glyph descriptions. MGLD file can be imported or viewed separately, without parsing data files itself.

MGLD file start from string

`MGLD npnts nprim ntutr nglfs # optional description`

which contain signature ‘MGLD’ and number of points *npnts*, number of primitives *nprim*, number of textures *ntutr*, number of glyph descriptions *nglfs*, and optional description. Empty strings and string with ‘#’ are ignored.

Next, file contain *npnts* strings with points coordinates and colors. The format of each string is

`x y z c t ta u v w r g b a`

Here  $x, y, z$  are coordinates,  $c, t$  are color indexes in texture,  $ta$  is normalized  $t$  according to current alpha setting,  $u, v, w$  are coordinates of normal vector (can be NAN if disabled),  $r, g, b, a$  are RGBA color values.

Next, file contain *nprim* strings with properties of primitives. The format of each string is

```
type n1 n2 n3 n4 id s w p
```

Here *type* is kind of primitive (0 - mark, 1 - line, 2 - triangle, 3 - quadrangle, 4 - glyph), *n1...n4* is index of point for vertexes, *id* is primitive identification number, *s* and *w* are size and width if applicable, *p* is scaling factor for glyphs.

Next, file contain *ntxt* strings with descriptions of textures. The format of each string is

```
smooth alpha colors
```

Here *smooth* set to enable smoothing between colors, *alpha* set to use half-transparent texture, *colors* contain color scheme itself as it described in Section 3.4 [Color scheme], page 149.

Finally, file contain *nglfs* entries with description of each glyph used in the figure. The format of entries are

```
nT nL
xA yA xB yB xC yC ...
xP yP ...
```

Here *nT* is the number of triangles; *nL* is the number of line vertexes; *xA, yA, xB, yB, xC, yC* are coordinates of triangles; and *xP, yP, xQ, yQ* are coordinates of lines. Line coordinate *xP=0x3fff, yP=0x3fff* denote line breaking.

### B.3 JSON format

MathGL can save points and primitives of 3D object. It contain a set of variables listed below.

<b>'width'</b>	width of the image;
<b>'height'</b>	height of the image
<b>'depth'</b>	depth of the image, usually $=\sqrt{\text{width} \times \text{height}}$ ;
<b>'npnts'</b>	number of points (vertexes);
<b>'pnts'</b>	array of coordinates of points (vertexes), each element is array in form $[x, y, z]$ ;
<b>'nprim'</b>	number of primitives;
<b>'prim'</b>	array of primitives, each element is array in form $[\text{type}, n1, n2, n3, n4, \text{id}, s, w, p, z, \text{color}]$ . Here <i>type</i> is kind of primitive (0 - mark, 1 - line, 2 - triangle, 3 - quadrangle, 4 - glyph), <i>n1...n4</i> is index of point for vertexes and <i>n2</i> can be index of glyph coordinate, <i>s</i> and <i>w</i> are size and width if applicable, <i>z</i> is average z-coordinate, <i>id</i> is primitive identification number, <i>p</i> is scaling factor for glyphs.
<b>'ncoor'</b>	number of glyph positions

<code>'coord'</code>	array of glyph positions, each element is array in form [dx,dy]
<code>'nglfs'</code>	number of glyph descriptions
<code>'glfs'</code>	array of glyph descriptions, each element is array in form [nL, [xP0, yP0, xP1, yP1 ...]]. Here nL is the number of line vertexes; and xP, yP, xQ, yQ are coordinates of lines. Line coordinate xP=0x3fff, yP=0x3fff denote line breaking.

## Appendix C Plotting time

Table below show plotting time in seconds for all samples in file `examples/samples.cpp` (<http://sourceforge.net/p/mathgl/code/HEAD/tree/mathgl-2x/examples/samples.cpp>). The test was done in my laptop (i5-2430M) with 64-bit Debian.

Few words about the speed. Firstly, direct bitmap drawing (Quality=4,5,6) is faster than buffered one (Quality=0,1,2), but sometimes it give incorrect result (see [cloud], page 214) and don't allow to export in vector or 3d formats (like EPS, SVG, PDF ...). Secondly, lower quality is faster than high one generally, i.e. Quality=1 is faster than Quality=2, and Quality=0 is faster than Quality=1. However, if plot contain a lot of faces (like [cloud], page 214, [surf3], page 213, [pipe], page 226, [dew], page 224) then Quality=0 may become slow, especially for small images. Finally, smaller images are drawn faster than larger ones.

Results for image size 800\*600 (default one).

Name	q=0	q=1	q=2	q=4	q=5	q=6	q=8
alpha	0.0994	0.106	0.137	0.034	0.0565	0.068	0.0332
area	0.0813	0.0666	0.11	0.0342	0.0426	0.0794	0.0435
aspect	0.0924	0.142	0.118	0.0281	0.0376	0.0693	0.0246
axial	0.706	0.606	0.911	0.243	0.264	0.475	0.0657
axis	0.136	0.163	0.155	0.0299	0.0293	0.0494	0.0506
barh	0.0811	0.105	0.121	0.0299	0.0326	0.0454	0.0273
bars	0.739	0.134	0.134	0.0477	0.0832	0.0662	0.0641
belt	0.0576	0.0984	0.0916	0.0429	0.0307	0.0465	0.0252
box	0.0938	0.105	0.127	0.0312	0.0352	0.0549	0.0501
boxplot	0.0469	0.127	0.0534	0.0587	0.0252	0.035	0.0152
boxs	0.242	0.223	0.331	0.0567	0.0683	0.189	0.095
candle	0.0842	0.132	0.0634	0.0541	0.0509	0.0532	0.016
chart	0.506	0.427	0.715	0.182	0.199	0.477	0.154
cloud	0.0927	4.72	5.16	0.0308	1.2	1.64	0.0265
colorbar	0.266	0.273	0.246	0.174	0.182	0.253	0.267
combined	0.603	0.401	0.458	0.306	0.276	0.362	0.256
cones	0.308	0.189	0.241	0.144	0.132	0.172	0.109
cont	0.079	0.0795	0.124	0.0374	0.0374	0.0606	0.0326
cont_xyz	0.14	0.106	0.249	0.33	0.159	0.115	0.152
conta	0.135	0.106	0.132	0.0822	0.101	0.0894	0.083
contd	0.209	0.181	0.2	0.0722	0.075	0.109	0.0666
contf	0.156	0.157	0.177	0.0645	0.0658	0.0881	0.0584
contf_xyz	0.214	0.211	0.175	0.0934	0.111	0.104	0.0851
contfa	0.293	0.224	0.259	0.142	0.165	0.176	0.14
contv	0.113	0.105	0.135	0.051	0.0482	0.071	0.0386
correl	0.0597	0.0798	0.0772	0.0617	0.0697	0.0569	0.0284
curvcoor	0.0906	0.0892	0.134	0.0434	0.0432	0.0688	0.0403
cut	1.21	0.776	0.795	0.843	0.632	0.683	0.62
dat_diff	0.144	0.104	0.13	0.105	0.0779	0.0954	0.0927
dat_extra	0.442	0.279	0.304	0.122	0.213	0.257	0.0869
data1	5.13	4.08	3.72	3.74	3.45	3.95	3.35

data2	3.31	2.72	2.81	3	2.98	2.82	2.61
dens	0.105	0.118	0.168	0.0456	0.0547	0.085	0.0426
dens_xyz	0.158	0.16	0.23	0.0827	0.168	0.106	0.148
densa	0.127	0.138	0.159	0.0915	0.0981	0.11	0.0805
dew	1.7	0.831	0.765	0.204	0.172	0.182	0.0895
dots	0.148	0.119	0.13	0.0535	0.0535	0.0674	0.0636
error	0.104	0.0724	0.0861	0.0999	0.121	0.111	0.0285
error2	0.0625	0.0672	0.0873	0.0463	0.0359	0.0558	0.0365
export	0.177	0.221	0.249	0.112	0.124	0.154	0.17
fall	0.135	0.0346	0.0641	0.0363	0.0485	0.0797	0.084
fit	3.11	0.169	0.13	0.0734	0.0727	2.99	0.123
flow	0.513	0.386	0.497	0.272	0.423	0.377	0.262
fog	0.1	0.113	0.117	0.0297	0.039	0.0712	0.0616
fonts	2.67	2.63	3.03	2.74	2.67	2.88	2.94
grad	0.124	0.119	0.192	0.127	0.0672	0.109	0.0476
hist	0.216	0.202	0.226	0.131	0.149	0.142	0.0786
indirect	0.0602	0.0582	0.0586	0.0844	0.0792	0.152	0.0262
inplot	0.148	0.139	0.0964	0.0399	0.0624	0.0449	0.0495
label	0.146	0.0577	0.169	0.0578	0.0959	0.079	0.0289
legend	0.168	0.227	0.23	0.0345	0.0364	0.073	0.0382
light	0.107	0.0814	0.0821	0.0804	0.073	0.0774	0.073
loglog	0.0776	0.0741	0.113	0.0451	0.0375	0.0559	0.0335
map	0.178	0.119	0.188	0.0767	0.0887	0.111	0.0304
mark	0.0785	0.141	0.0551	0.028	0.0311	0.0365	0.016
mask	0.0854	0.131	0.115	0.0296	0.0412	0.0547	0.0213
mesh	0.11	0.112	0.0894	0.0368	0.0518	0.0508	0.0724
mirror	0.197	0.104	0.146	0.0363	0.0433	0.0695	0.0384
molecule	0.153	0.107	0.199	0.0653	0.0657	0.086	0.037
ode	1.49	0.618	0.718	0.62	0.595	0.72	0.556
ohlc	0.0347	0.0487	0.0551	0.0406	0.0281	0.0389	0.0167
param1	0.316	0.247	0.321	0.119	0.155	0.158	0.0958
param2	0.505	0.428	0.496	0.152	0.138	0.187	0.104
param3	3.33	3.59	3.37	1.82	1.89	1.94	1.65
paramv	4.31	4.7	4.25	1.21	1.13	1.23	1.07
parser	0.074	0.102	0.128	0.0973	0.154	0.104	0.145
pde	0.943	0.934	0.987	1.01	0.816	0.79	0.809
pipe	4.13	2.98	2.46	0.597	0.848	0.815	0.588
plot	0.0702	0.0848	0.112	0.0409	0.0409	0.0568	0.029
primitives	0.0998	0.0958	0.175	0.0313	0.0363	0.0815	0.0265
projection	0.138	0.14	0.222	0.0463	0.046	0.157	0.104
projection5	0.124	0.132	0.197	0.0414	0.0423	0.1	0.0353
qo2d	0.375	0.331	0.364	0.244	0.279	0.287	0.244
radar	0.0348	0.0503	0.0598	0.0582	0.0297	0.083	0.0147
refill	0.391	0.28	0.233	0.161	0.133	0.155	0.0867
region	0.0748	0.0719	0.117	0.0291	0.0319	0.0566	0.0256
schemes	0.123	0.102	0.184	0.0371	0.0392	0.0717	0.0469
several_light	0.0862	0.102	0.128	0.0284	0.0327	0.0912	0.0246

solve	0.126	0.0944	0.148	0.06	0.078	0.0841	0.0579
stem	0.061	0.0558	0.124	0.0312	0.0315	0.0479	0.0265
step	0.132	0.0962	0.0708	0.0321	0.0322	0.191	0.0398
stereo	0.0706	0.0877	0.137	0.0304	0.0334	0.0672	0.0394
stfa	0.177	0.0918	0.146	0.0518	0.0657	0.0935	0.059
style	0.176	0.166	0.226	0.0408	0.0426	0.082	0.0252
surf	0.191	0.191	0.233	0.0669	0.0704	0.105	0.118
surf3	3.1	2.9	2.88	2.89	2.78	3.57	1.16
surf3a	0.754	0.471	0.526	0.399	0.394	0.511	0.349
surf3c	0.736	0.448	0.502	0.383	0.387	0.494	0.333
surfa	0.0647	0.112	0.0916	0.0298	0.0311	0.0626	0.025
surfc	0.0588	0.103	0.139	0.0313	0.0311	0.0602	0.0248
table	0.194	0.226	0.279	0.0358	0.0361	0.0809	0.0438
tape	0.1	0.108	0.125	0.0302	0.0326	0.0526	0.0405
tens	0.0959	0.109	0.0642	0.0727	0.0965	0.074	0.0254
ternary	0.142	0.137	0.205	0.041	0.0418	0.0942	0.0355
text	0.123	0.124	0.166	0.045	0.0567	0.0758	0.025
text2	0.132	0.132	0.167	0.0721	0.0872	0.095	0.0467
textmark	0.224	0.0722	0.0939	0.126	0.0474	0.0575	0.0242
ticks	0.186	0.145	0.207	0.0341	0.042	0.0689	0.0609
tile	0.0861	0.0811	0.11	0.0277	0.0299	0.0451	0.0618
tiles	0.0558	0.0581	0.0772	0.0272	0.0303	0.0454	0.0583
torus	0.121	0.116	0.193	0.038	0.0448	0.104	0.0314
traj	0.0354	0.0395	0.0789	0.0467	0.0253	0.0364	0.0173
triangulation	0.0603	0.214	0.139	0.0455	0.0734	0.0845	0.0265
tripplot	0.0692	0.0519	0.131	0.0497	0.0325	0.0943	0.0124
tube	0.151	0.135	0.294	0.0491	0.0601	0.108	0.0609
type0	0.251	0.187	0.248	0.113	0.0785	0.157	0.0567
type1	0.249	0.179	0.274	0.0796	0.0786	0.145	0.0564
type2	0.246	0.212	0.25	0.0868	0.104	0.151	0.0625
vect	0.166	0.155	0.255	0.13	0.132	0.181	0.174
vecta	0.121	0.0841	0.108	0.0755	0.0672	0.155	0.0686
venn	0.0182	0.0527	0.0913	0.0368	0.0385	0.0706	0.013

Results for image size 1920\*1440 (print quality)

Name	q=0	q=1	q=2	q=4	q=5	q=6	q=8
alpha	0.16	0.184	0.362	0.0868	0.105	0.246	0.0674
area	0.102	0.165	0.327	0.0795	0.124	0.255	0.0656
aspect	0.094	0.0939	0.15	0.0664	0.0674	0.11	0.0618
axial	0.909	0.985	2.03	0.371	0.464	1.28	0.105
axis	0.121	0.121	0.198	0.0705	0.0683	0.123	0.0951
barh	0.0886	0.105	0.181	0.068	0.0795	0.137	0.13
bars	0.11	0.126	0.207	0.0868	0.0973	0.162	0.0754
belt	0.0944	0.12	0.212	0.0674	0.089	0.192	0.0629
box	0.122	0.161	0.25	0.0735	0.0951	0.174	0.0641
boxplot	0.0696	0.0699	0.118	0.0633	0.0634	0.106	0.0514

boxs	0.296	0.354	0.786	0.111	0.213	0.58	0.0872
candle	0.0762	0.0787	0.143	0.0789	0.0859	0.135	0.0548
chart	0.572	0.778	2.39	0.244	0.429	1.64	0.2
cloud	0.0818	7.16	9.38	0.0691	2.43	4.96	0.063
colorbar	0.221	0.248	0.363	0.227	0.285	0.378	0.11
combined	0.583	0.464	0.697	0.339	0.375	0.586	0.293
cones	0.289	0.24	0.452	0.184	0.191	0.375	0.131
cont	0.124	0.123	0.208	0.0785	0.0778	0.14	0.0774
cont_xyz	0.134	0.13	0.184	0.113	0.113	0.163	0.108
conta	0.128	0.154	0.289	0.112	0.129	0.162	0.177
contd	0.239	0.255	0.354	0.12	0.142	0.236	0.103
contf	0.203	0.226	0.334	0.111	0.13	0.226	0.0992
contf_xyz	0.17	0.215	0.305	0.126	0.151	0.236	0.118
contfa	0.287	0.369	0.493	0.18	0.203	0.401	0.165
contv	0.162	0.176	0.306	0.089	0.103	0.212	0.076
correl	0.099	0.0894	0.169	0.0815	0.0855	0.129	0.0659
curvcoor	0.132	0.131	0.231	0.084	0.0843	0.151	0.124
cut	1.25	0.842	1.06	0.743	0.768	1.13	0.644
dat_diff	0.127	0.163	0.261	0.115	0.135	0.216	0.0846
dat_extra	0.328	0.26	0.351	0.17	0.267	0.3	0.114
data1	4.98	3.75	3.98	3.6	3.52	3.9	3.29
data2	3.28	2.74	2.82	2.81	2.78	3.2	2.61
dens	0.153	0.244	0.407	0.1	0.141	0.273	0.0771
dens_xyz	0.15	0.211	0.358	0.123	0.162	0.282	0.0961
densa	0.183	0.211	0.351	0.12	0.154	0.296	0.0912
dew	1.7	0.942	1.03	0.224	0.241	0.361	0.129
dots	0.161	0.167	0.217	0.104	0.11	0.148	0.0639
error	0.101	0.109	0.176	0.11	0.113	0.194	0.0688
error2	0.0931	0.112	0.209	0.0742	0.0808	0.156	0.0675
export	0.287	0.363	0.531	0.174	0.22	0.36	0.253
fall	0.079	0.0752	0.157	0.0652	0.064	0.122	0.0868
fit	0.109	0.824	0.173	0.0894	0.0918	0.136	0.0793
flow	0.39	0.395	0.566	0.313	0.423	0.435	0.3
fog	0.111	0.169	0.362	0.0716	0.127	0.3	0.063
fonts	3	3.02	3.14	2.75	2.57	2.88	2.66
grad	0.119	0.229	0.424	0.103	0.158	0.306	0.0859
hist	0.294	0.303	0.358	0.209	0.217	0.274	0.0886
indirect	0.108	0.0908	0.147	0.111	0.114	0.16	0.0556
inplot	0.102	0.0998	0.174	0.0747	0.0726	0.123	0.0664
label	0.0836	0.0803	0.129	0.0808	0.08	0.12	0.0529
legend	0.185	0.195	0.3	0.0758	0.0864	0.171	0.0622
light	0.273	0.271	0.299	0.277	0.274	0.299	0.274
loglog	0.12	0.123	0.198	0.076	0.0779	0.133	0.0692
map	0.107	0.159	0.255	0.12	0.149	0.316	0.0684
mark	0.083	0.0937	0.126	0.072	0.0723	0.11	0.0507
mask	0.12	0.24	0.257	0.0723	0.141	0.169	0.0597
mesh	0.0866	0.0906	0.189	0.0661	0.0932	0.156	0.0638



mirror	0.163	0.164	0.305	0.08	0.094	0.209	0.0679
molecule	0.155	0.166	0.354	0.0754	0.111	0.263	0.0628
ode	0.652	0.654	0.788	0.631	0.636	0.793	0.635
ohlc	0.0668	0.0668	0.117	0.0675	0.0734	0.104	0.053
param1	0.288	0.295	0.518	0.137	0.158	0.352	0.147
param2	0.555	0.54	0.767	0.591	0.232	0.51	0.151
param3	3.15	3.63	3.78	1.86	2.03	2.35	1.61
paramv	4.55	4.9	4.7	1.14	1.19	1.35	1.11
parser	0.118	0.116	0.195	0.115	0.122	0.225	0.0954
pde	0.883	0.961	1.13	0.834	0.843	0.953	0.781
pipe	4.27	3.23	3.04	0.664	0.831	0.993	0.52
plot	0.154	0.119	0.221	0.0921	0.0933	0.143	0.0835
primitives	0.143	0.188	0.43	0.0749	0.108	0.313	0.0879
projection	0.182	0.207	0.494	0.0878	0.103	0.349	0.0768
projection5	0.17	0.2	0.427	0.0835	0.0956	0.323	0.11
qo2d	0.325	0.423	0.596	0.286	0.341	0.5	0.26
radar	0.0724	0.0738	0.115	0.076	0.0804	0.135	0.0499
refill	0.329	0.263	0.38	0.197	0.167	0.286	0.114
region	0.0964	0.235	0.265	0.0729	0.0986	0.191	0.0663
schemes	0.158	0.205	0.358	0.0947	0.116	0.231	0.0633
several_light	0.121	0.136	0.494	0.0704	0.0966	0.427	0.0609
solve	0.138	0.134	0.247	0.108	0.104	0.204	0.0741
stem	0.103	0.101	0.171	0.079	0.0745	0.135	0.0653
step	0.114	0.11	0.157	0.0782	0.0755	0.119	0.0636
stereo	0.12	0.138	0.308	0.0715	0.0996	0.274	0.065
stfa	0.128	0.168	0.366	0.0982	0.122	0.273	0.0732
style	0.181	0.202	0.338	0.0814	0.0929	0.19	0.0614
surf	0.235	0.233	0.395	0.12	0.137	0.284	0.0811
surf3	3.33	3.34	4.02	3.66	3.31	5.02	1.18
surf3a	0.825	0.581	0.901	0.469	0.641	1.34	0.378
surf3c	0.797	0.577	0.91	0.471	0.637	1.34	0.375
surfa	0.0979	0.141	0.358	0.0698	0.0953	0.264	0.0639
surfc	0.0992	0.135	0.3	0.0682	0.0964	0.252	0.0619
table	0.213	0.205	0.33	0.0784	0.0796	0.159	0.0633
tape	0.1	0.117	0.211	0.07	0.0808	0.147	0.0662
tens	0.0891	0.0865	0.14	0.108	0.11	0.153	0.181
ternary	0.18	0.199	0.376	0.0867	0.0968	0.263	0.07
text	0.164	0.165	0.253	0.0863	0.088	0.155	0.0635
text2	0.145	0.145	0.211	0.109	0.109	0.168	0.0717
textmark	0.115	0.113	0.171	0.0847	0.0815	0.133	0.16
ticks	0.18	0.184	0.277	0.0756	0.074	0.161	0.063
tile	0.0826	0.116	0.209	0.0671	0.0839	0.16	0.0638
tiles	0.0856	0.134	0.217	0.0685	0.0901	0.157	0.0678
torus	0.168	0.202	0.521	0.0812	0.114	0.372	0.0758
traj	0.0641	0.065	0.13	0.066	0.0625	0.112	0.0529
triangulation	0.0895	0.127	0.304	0.0839	0.117	0.271	0.0707
triplot	0.0721	0.176	0.581	0.0639	0.118	0.439	0.0497

tube	0.151	0.219	0.48	0.0922	0.132	0.347	0.0719
type0	0.296	0.289	0.656	0.135	0.196	0.555	0.117
type1	0.294	0.286	0.661	0.138	0.196	0.566	0.116
type2	0.311	0.304	0.67	0.155	0.215	0.574	0.135
vect	0.19	0.192	0.31	0.175	0.164	0.273	0.12
vecta	0.0971	0.0982	0.186	0.108	0.118	0.31	0.0698
venn	0.064	0.24	0.397	0.0631	0.155	0.323	0.0474

## Appendix D GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### 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. Secondly, 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

## A

AddLegend	193
AddLight	158
AddTick	168
Adjust	168
Alpha	158
alpha	153
AlphaDef	158
alphadef	153
Ambient	158
Area	194
Arrows	147
ArrowSize	160
ask	286
Aspect	172
AutoCorrel	264
Axial	207
Axis	167, 190
AxisStl	168

## B

Ball	184
Barh	194
Bars	194
BarWidth	160
Beam	213
Belt	207
Box	190
BoxPlot	194
Boxs	207

## C

call	287
Candle	194
Chart	194
chdir	286
Clean	253
ClearLegend	193
Clf	183
CloseGIF	180
Cloud	213
Color scheme	149
Colorbar	190
Column	264
ColumnPlot	172
Combine	183, 264
Cone	184
Cones	194
Cont	207
Cont3	213
ContD	207
ContF	207

ContF3	213
ContFXYZ	227
ContXYZ	227
CopyFont	161
Correl	264
CosFFT	268
CRange	166
Create	253
Crop	253
Crust	227
CTick	168
CumSum	268
Curve	184
Cut	161
cut	154
CutOff	161

## D

DataGrid	235
defchr	287
define	286
defnum	287
Delete	253
Dens	207
Dens3	213
DensXYZ	227
Dew	222
Diff	268
Diff2	268
Dots	227
Drop	184

## E

else	287
elseif	287
EndFrame	180
endif	287
Envelop	268
Error	184, 194
Evaluate	264
Export	261
Extend	253

**F**

Face .....	184
FaceX .....	184
FaceY .....	184
FaceZ .....	184
Fall .....	207
fgets .....	188
Fill .....	235, 255
Find .....	275
FindAny .....	275
Fit .....	232
Fit2 .....	232
Fit3 .....	232
FitS .....	232
Fl_MathGL .....	237, 241
Flow .....	222
FlowP .....	222
Fog .....	159
Font .....	161
Font styles .....	151
fontsize .....	154
for .....	287
FPlot .....	227
FSurf .....	227
func .....	287

**G**

GetNumFrame .....	180
GetNx .....	273
GetNy .....	273
GetNz .....	273
GetWarn .....	163
Glyph .....	184
Grad .....	207
Grid .....	190, 207
Grid3 .....	213

**H**

Hankel .....	268
Hist .....	235, 264

**I**

if .....	287
Import .....	261
InPlot .....	172
Insert .....	253
Integral .....	268

**J**

Join .....	253
------------	-----

**L**

Label .....	188, 190, 194
Last .....	275
Legend .....	193
legend .....	154
Light .....	158
Line .....	184
Line style .....	147
Linear .....	272
Linear1 .....	272, 273
List .....	255
load .....	287
LoadBackground .....	183
LoadFont .....	161

**M**

Map .....	217
Mark .....	184, 194
Mark style .....	147
MarkSize .....	160
MathGL overview .....	1
MathGL setup .....	157
Max .....	264
Maximal .....	274
Mesh .....	207
MeshNum .....	160
meshnum .....	154
Message .....	163
mgColor .....	311
mgData .....	252
mgDraw .....	240
mgExpr .....	282
mgExprC .....	282
mgFitPts .....	232
mgGLUT .....	237
mgGraph .....	157
mgParse .....	291
mgPoint .....	313
mgWnd .....	237, 238
Min .....	264
Minimal .....	274
Mirror .....	268
Modify .....	255
Momentum .....	264, 274
MPI_Recv .....	183
MPI_Send .....	183
MultiPlot .....	172

**N**

NeedStop .....	165
NewFrame .....	180
next .....	288
Norm .....	268
NormSl .....	268

**O**

once .....	288
Origin .....	166

**P**

Palette .....	162
Perspective .....	172
Pipe .....	222
Plot .....	194
Pop .....	172
PrintInfo .....	273
Push .....	172
Puts .....	188
PutsFit .....	232
Putsw .....	188

**Q**

QMathGL .....	237, 243
QuadPlot .....	227

**R**

Radar .....	194
Ranges .....	166
Rasterize .....	183
Read .....	261
ReadAll .....	261
ReadHDF .....	261
ReadMat .....	261
ReadRange .....	261
Rearrange .....	253
Refill .....	255
Region .....	194
ResetFrames .....	180
Resize .....	264
RestoreFont .....	161
return .....	287
rkstep .....	288
Roll .....	268
Roots .....	264
Rotate .....	172
RotateN .....	172
RotateText .....	161

**S**

Save .....	261
SaveHDF .....	261
Set .....	255
SetAlphaDef .....	158
SetAmbient .....	158
SetArrowSize .....	160
SetAxisStl .....	168
SetBarWidth .....	160
SetCoor .....	167
SetCut .....	161
SetCutBox .....	161
SetEventFunc .....	165
SetFontDef .....	161
SetFontSize .....	161
SetFontSizeCM .....	161
SetFontSizeIN .....	161
SetFontSizePT .....	161
SetFunc .....	167
SetLegendBox .....	193
SetLegendMarks .....	193
SetMarkSize .....	160
SetMask .....	163
SetMaskAngle .....	163
SetMeshNum .....	160
SetOrigin .....	166
SetOriginTick .....	168
SetPalette .....	162
SetPlotId .....	160
SetRange .....	166
SetRanges .....	166
SetRotatedText .....	161
SetSize .....	175
SetTickLen .....	168
SetTickRotate .....	168
SetTicks .....	168
SetTickSkip .....	168
SetTicksVal .....	168
SetTickTempl .....	168
SetTickTime .....	168
SetTranspType .....	158
SetTuneTicks .....	168
SetWarn .....	163
Sew .....	268
ShowImage .....	176
SinFFT .....	268
Smooth .....	268
Sort .....	253
Sphere .....	184
Spline .....	271
Spline1 .....	272
Squeeze .....	253
StartGIF .....	180
Stem .....	194
Step .....	194
STFA .....	217
StickPlot .....	172
Stop .....	165

stop .....	288
SubData .....	264
SubPlot .....	172
Sum .....	264
Surf .....	207
Surf3 .....	213
Surf3A .....	217
Surf3C .....	217
SurfA .....	217
SurfC .....	217
Swap .....	268

## T

Tape .....	194
Tens .....	194
Ternary .....	167
Text .....	188
TextMark .....	194
Textual formulas .....	152
TickLen .....	168
Tile .....	207
TileS .....	217
Title .....	172
Torus .....	194
Trace .....	264
Traj .....	222
Transpose .....	253
TranspType .....	158
TriCont .....	227
TriPlot .....	227
Tube .....	194

## V

value .....	154
Var .....	255
Vect .....	222
View .....	172

## W

widgets .....	7, 237, 241, 243, 248
window .....	7, 237, 238
Write .....	176
WriteBMP .....	176
WriteBPS .....	176
WriteEPS .....	176
WriteFrame .....	176
WriteGIF .....	176
WriteJPEG .....	176
WriteOBJ .....	176
WritePNG .....	176
WritePRC .....	176
WriteSVG .....	176
WriteTEX .....	176
WriteTGA .....	176
WriteWGL .....	176
wxMathGL .....	248

## X

xrange .....	153
XRange .....	166
XTick .....	168

## Y

YRange .....	166
yrange .....	153
YTick .....	168

## Z

ZRange .....	166
zrange .....	154
ZTick .....	168