

MathGL

for version 2.0

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

This manual is for MathGL (version 2.0), a collection of classes and routines for scientific plotting. Please report any errors in this manual to mathgl.abalakin@gmail.org.

Copyright © 2008 Alexey 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, with the Front-Cover Texts being “A GNU Manual,” and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License.”

(a) The FSF’s Back-Cover Text is: “You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom.”

Table of Contents

1 Overview	1
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.*	3
1.6 Utilities for parsing MGL	4
2 MathGL examples	5
2.1 Basic usage	6
2.1.1 Using MathGL window	6
2.1.2 Drawing to file	8
2.1.3 Animation	9
2.1.4 Drawing in memory	12
2.1.5 Using QMathGL	13
2.1.6 MathGL and PyQt	14
2.2 Advanced usage	15
2.2.1 Subplots	15
2.2.2 Axis and ticks	17
2.2.3 Curvilinear coordinates	21
2.2.4 Colorbars	22
2.2.5 Bounding box	23
2.2.6 Ternary axis	24
2.2.7 Text features	26
2.2.8 Legend sample	27
2.2.9 Cutting sample	28
2.3 Data handling	29
2.3.1 Array creation	29
2.3.2 Linking array	31
2.3.3 Change data	31
2.4 Data plotting	33
2.5 1D samples	36
2.5.1 Plot sample	37
2.5.2 Radar sample	38
2.5.3 Step sample	39
2.5.4 Tens sample	40
2.5.5 Area sample	41
2.5.6 Region sample	42
2.5.7 Stem sample	43
2.5.8 Bars sample	44
2.5.9 Barh sample	45
2.5.10 Cones sample	46
2.5.11 Chart sample	47

2.5.12	BoxPlot sample	48
2.5.13	Candle sample	49
2.5.14	Error sample	49
2.5.15	Mark sample	50
2.5.16	TextMark sample	51
2.5.17	Label sample	52
2.5.18	Tube sample	53
2.5.19	Tape sample	53
2.5.20	Torus sample	54
2.6	2D samples	55
2.6.1	Surf sample	56
2.6.2	SurfC sample	57
2.6.3	SurfA sample	58
2.6.4	Mesh sample	58
2.6.5	Fall sample	59
2.6.6	Belt sample	60
2.6.7	Boxs sample	60
2.6.8	Tile sample	61
2.6.9	TileS sample	62
2.6.10	Dens sample	63
2.6.11	Cont sample	63
2.6.12	ContF sample	64
2.6.13	ContD sample	65
2.6.14	ContV sample	66
2.6.15	Axial sample	67
2.6.16	Grad sample	68
2.7	3D samples	69
2.7.1	Surf3 sample	69
2.7.2	Surf3C sample	70
2.7.3	Surf3A sample	71
2.7.4	Cloud sample	71
2.7.5	Dens3 sample	72
2.7.6	Cont3 sample	73
2.7.7	ContF3 sample	74
2.7.8	Dens projection sample	74
2.7.9	Cont projection sample	75
2.7.10	ContF projection sample	76
2.7.11	TriPlot and QuadPlot	76
2.7.12	Dots sample	77
2.8	Vector field samples	78
2.8.1	Vect sample	79
2.8.2	Traj sample	80
2.8.3	Flow sample	81
2.8.4	Pipe sample	82
2.8.5	Dew sample	83
2.9	Hints	84
2.9.1	“Compound” graphics	84
2.9.2	Transparency and lighting	85

2.9.3	Types of transparency	86
2.9.4	Adding fog	88
2.9.5	Several light sources	89
2.9.6	Using primitives	90
2.9.7	STFA sample	93
2.9.8	Mapping visualization	94
2.9.9	Making histogram	95
2.9.10	Nonlinear fitting hints	96
2.9.11	PDE solving hints	97
2.9.12	MGL parser using	100
2.9.13	“Templates”	102
2.9.14	Stereo image	103
2.10	FAQ	103
3	General concepts	107
3.1	Coordinate axes	107
3.2	Color styles	108
3.3	Line styles	108
3.4	Color scheme	109
3.5	Font styles	110
3.6	Textual formulas	112
3.7	Command options	113
3.8	Interfaces	114
3.8.1	C/Fortran interface	114
3.8.2	C++/Python interface	114
4	MathGL core	116
4.1	Create and delete objects	116
4.2	Graphics setup	116
4.2.1	Transparency	116
4.2.2	Lighting	117
4.2.3	Fog	118
4.2.4	Default sizes	118
4.2.5	Cutting	119
4.2.6	Font settings	119
4.2.7	Palette and colors	120
4.2.8	Error handling	121
4.3	Axis settings	122
4.3.1	Ranges (bounding box)	122
4.3.2	Curved coordinates	123
4.3.3	Ticks	124
4.4	Subplots and rotation	126
4.5	Export picture	129
4.5.1	Export to file	130
4.5.2	Frames/Animation	134
4.5.3	Bitmap in memory	134
4.5.4	Parallelization	136
4.6	Primitives	136

4.7	Text printing	139
4.8	Axis and Colorbar	141
4.9	Legend	143
4.10	1D plotting	144
4.11	2D plotting	154
4.12	3D plotting	160
4.13	Dual plotting	164
4.14	Vector fields	167
4.15	Other plotting	171
4.16	Nonlinear fitting	177
4.17	Data manipulation	179
4.18	IDTF functions	181
5	Widget classes	182
5.1	mglWindow class	182
5.2	Fl_MathGL class	184
5.3	QMathGL class	186
6	Data processing	191
6.1	Public variables	191
6.2	Data constructor	192
6.3	Data resizing	192
6.4	Data filling	194
6.5	File I/O	197
6.6	Make another data	198
6.7	Data changing	201
6.8	Interpolation	203
6.9	Data information	203
6.10	Operators	206
6.11	Global functions	207
6.12	Evaluate expression	209
7	MGL scripts	211
7.1	MGL definition	211
7.2	Program flow commands	212
7.3	mglParse class	214
8	Other classes	216
8.0.1	Define new kind of plot (mglBase class)	216
8.0.2	User defined types (mglDataA class)	222
8.1	mglColor class	224
8.2	mglPoint class	225
8.3	mglFont class	227
8.3.1	Format of font files	228
A	Appendix A TeX-like symbols	230

Appendix B GNU Free Documentation License	238
Index	245

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 (2.0) 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](#)). 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 4.10 \[1D plotting\], page 144](#));
- two-dimensional plots (Mesh, Surf, Dens, Cont, ContF, Boxes, Axial, Fall, Belt, Tile, see [Section 4.11 \[2D plotting\], page 154](#));
- three-dimensional plots (Surf3, Dens3, Cont3, ContF3, Cloud-like, see [Section 4.12 \[3D plotting\], page 160](#));
- 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 4.13 \[Dual plotting\], page 164](#));
- and so on. For details see [Chapter 4 \[MathGL core\], page 116](#).

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, TGA, BMP format, or in vector EPS, SVG or TeX format, or in 3D formats OBJ, OFF, STL, or in IDTF 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 110](#)). 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 191](#)). 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 112](#)). 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 standard script `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-qt` 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-langall` for all languages. You can use WYSIWYG tool (`cmake-gui`) to view all of them, or type `cmake -D enable-all=on -D enable-langall=on .` in command line for enabling all features.

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 (or in any other folder and setup paths). By default, precompiled versions include the support of GSL (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, see [Download](#) section at homepage).

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 191](#)). 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 108](#)) 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 107](#)) – “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 `-lmgl`.

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 `-lmgl-wnd`. Fortran users also should add C++ library by the option `-lstdc++`.

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;
- **-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;
- **-L** loc set locale to loc;
- **-o** name set output file name;
- **-h** print help message.

Additionally 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/specifed animation parameters, tool will execute script several times – once for each value of \$0.

2 MathGL examples

This chapter contain information about basic and advanced MathGL, hints and samples for all types of graphics. I recommend you read first 2 sections one after another and at least look on [Section 2.9 \[Hints\], page 84](#) section. Also I recommend you to look at [Chapter 3 \[General concepts\], page 107](#) and [Section 2.10 \[FAQ\], page 103](#).

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 install MathGL package by command `octave:1> pkg install /usr/share/mathgl/octave/mathgl.tar.gz` from sudo octave)

```
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 `mglWindow` or `mglGLUT` (see [Chapter 5 \[Widget classes\]](#), page 182) and the following drawing in this window. There is a corresponding code:

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

```

}

//-----
int main(int argc,char **argv)
{
    mglWindow 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-wnd -lmgl
```

Alternatively you can create yours own class inherited from class `mglDraw` and reimplement the function `Draw()` in it:

```

#include <mgl2/window.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;
    mglWindow 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);
    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 `mglWindow`) 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 126](#)) 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 9](#)) 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 such kind of animation is rather slow and not well suitable for visualization of running calculations. For the last case one can use `Update()` function. The most simple case for doing this is to use `mglDraw` class and reimplement its `Calc()` method.

```
#include <mgl2/window.h>
class Foo : public mglDraw
{
```

```

mglPoint pnt; // some result of calculation
public:
    mglWindow *Gr; // graphics to be updated
    int Draw(mglGraph *gr);
    void Calc();
} foo;
//-----
void Foo::Calc()
{
    for(int i=0;i<30;i++) // do calculation
    {
        sleep(2); // which can be very long
        pnt = mglPoint(2*mgl_rnd()-1,2*mgl_rnd()-1);
        Gr->Update(); // update window
    }
}
//-----
int Foo::Draw(mglGraph *gr)
{
    gr->Line(mglPoint(),pnt,"Ar2");
    gr->Box();
    return 0;
}
//-----
int main(int argc,char **argv)
{
    mglWindow gr(&foo,"MathGL examples");
    foo.Gr = &gr; foo.Run();
    return gr.Run();
}

```

Previous sample can be run in C++ only since it use C++ class `mglDraw`. However similar idea can be used even in Fortran or SWIG-based (Python/Octave/...) if one use FLTK window. 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)
{
    mglWindow gr("test"); // create window
    gr.RunThr(); // run event loop in separate thread
    for(int i=0;i<10;i++) // do calculation
    {
        sleep(1); // which can be very long
        pnt = mglPoint(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(),"");
    }
}

```

```

    gr.Update();           // update window when you need it
}
return 0;   // finish calculations and close the window
}

```

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 methods 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.GetBits(),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.GetBits(), 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 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/qt.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();
}

```

2.1.6 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_Grayscale8)
        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_()

```

2.2 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`, `InPlot`, `Title`, `Aspect` and `Rotate` and so on (see [Section 4.4 \[Subplots and rotation\], page 126](#)). The order of their calling is strictly determined. First, one changes the position of plot in image area (functions `SubPlot`, `InPlot` and `MultiPlot`). Secondly, you can add the title of plot by `Title` function. After that one may rotate the plot (function `Rotate`). Finally, one may change aspects of axes (function `Aspect`). The following code illustrates the aforesaid it:

```

int sample(mglGraph *gr)
{

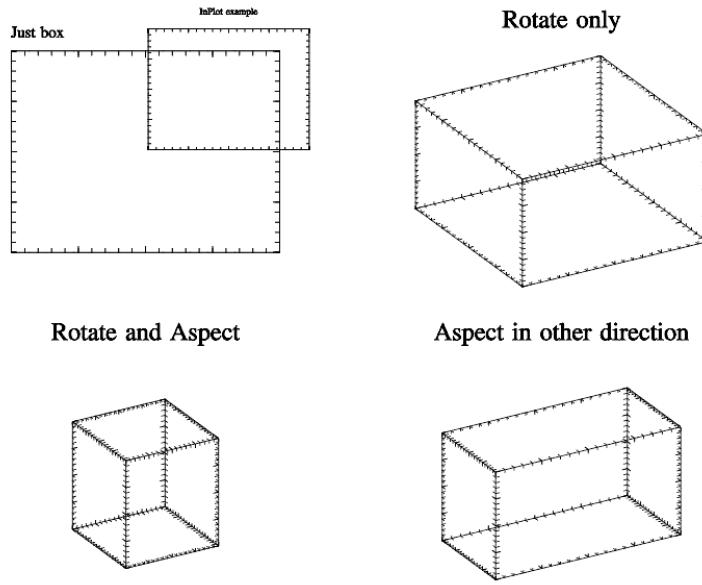
```

```

gr->SubPlot(2,2,0); gr->Box();
gr->Puts(mglPoint(-1,1.1),"Just box","");
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 4.7 \[Text printing\], page 139](#)). 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 26](#).



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

```

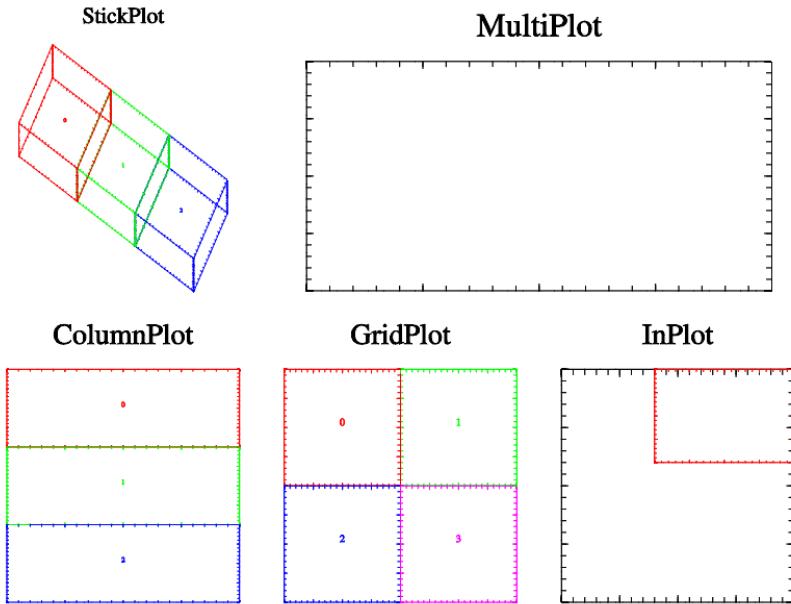
int sample(mglGraph *gr)
{
    gr->SubPlot(3,2,0); gr->Title("StickPlot");
    gr->StickPlot(3, 0, 20, 30); gr->Box("r"); gr->Puts(mglPoint(0),"0","r");
    gr->StickPlot(3, 1, 20, 30); gr->Box("g"); gr->Puts(mglPoint(0),"1","g");
    gr->StickPlot(3, 2, 20, 30); gr->Box("b"); gr->Puts(mglPoint(0),"2","b");
    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 122](#)). Ticks on axis are specified by function `SetTicks`, `SetTicksVal`, `SetTicksTime` (see [Section 4.3.3 \[Ticks\]](#), [page 124](#)). But usually

Function `Axis` 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` 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
}

```

```

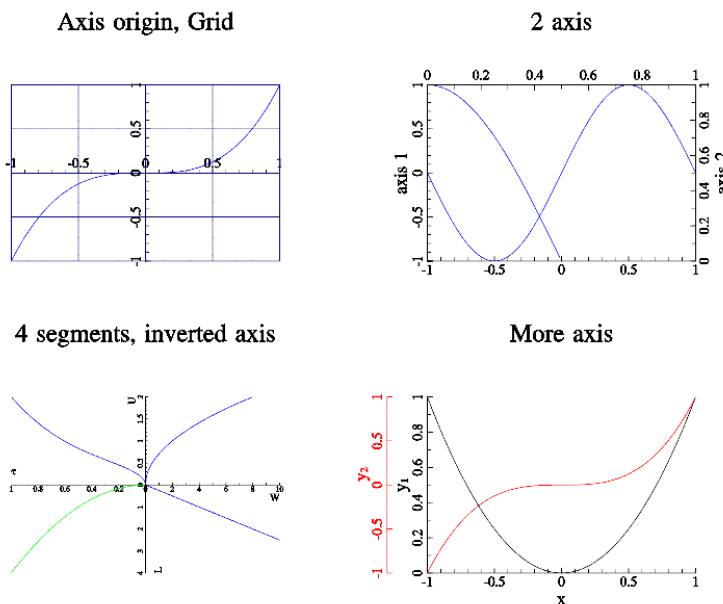
gr->Axis(); gr->Label('y',"axis 2",0); gr->FPlot("cos(pi*x)");

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).



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.

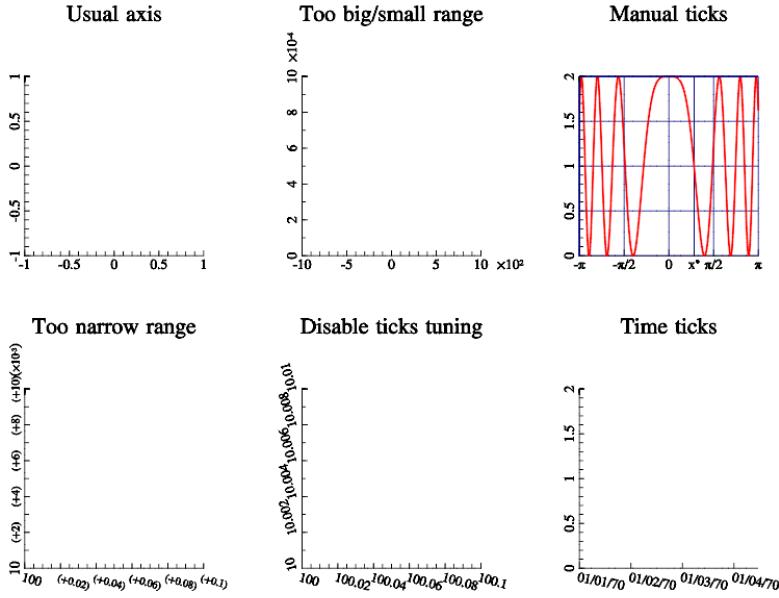
Also, one can specify its own ticks with arbitrary labels by help of `SetTicksVal` function. Or one can set ticks in time format. In last case MathGL will try to select optimal format for labels with automatic switching between years, months/days, hours/minutes/seconds or microseconds. However, you can 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(3,2,0); gr->Title("Usual axis"); gr->Axis();
    gr->SubPlot(3,2,1); gr->Title("Too big/small range");
    gr->SetRanges(-1000,1000,0,0.001); gr->Axis();
    gr->SubPlot(3,2,3); gr->Title("Too narrow range");
    gr->SetRanges(100,100.1,10,10.01); gr->Axis();
    gr->SubPlot(3,2,4); gr->Title("Disable ticks tuning");
    gr->SetTuneTicks(0); gr->Axis();

    gr->SubPlot(3,2,2); gr->Title("Manual ticks"); gr->SetRanges(-M_PI,M_PI, 0, 2);■
    float 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^*\\pi/2\\n\\pi");■
    gr->Axis(); gr->Grid(); gr->FPlot("2*cos(x^2)^2", "r2");

    gr->SubPlot(3,2,5); gr->Title("Time ticks"); gr->SetRange('x',0,3e5);
    gr->SetTicksTime('x',0); gr->Axis();
    return 0;
}
```



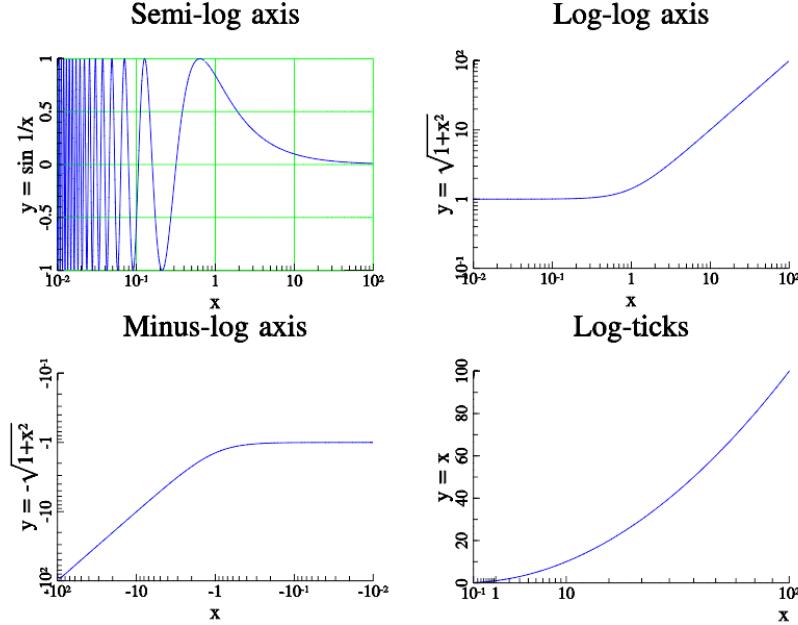
The last sample I want to show in this subsection is Log-axis. From MathGL's point of view, the log-axis is particular case of general curvilinear coordinates. So, we need first define new coordinates (see also [Section 2.2.3 \[Curvilinear coordinates\], page 21](#)) by help of `SetFunc` or `SetCoor` functions. At this one should wary 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->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->FPPlot("2*t-1","0.5","0","r2");
    gr->Axis(); gr->Grid();

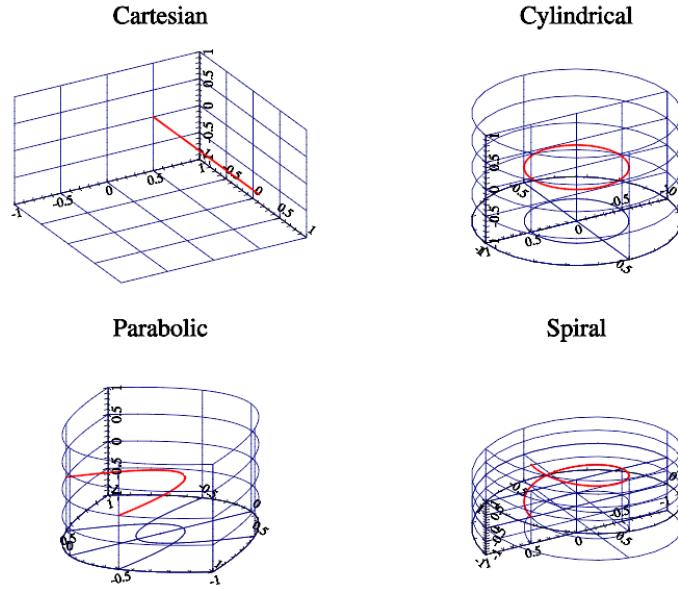
    gr->SetFunc("y*sin(pi*x)","y*cos(pi*x)",0);
    gr->SubPlot(2,2,1); gr->Title("Cylindrical"); gr->Rotate(50,60);
    gr->FPPlot("2*t-1","0.5","0","r2");
    gr->Axis(); gr->Grid();

    gr->SetFunc("2*y*x","y*y - x*x",0);
    gr->SubPlot(2,2,2); gr->Title("Parabolic"); gr->Rotate(50,60);
    gr->FPPlot("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;
}

```



2.2.4 Colorbars

MathGL handle colorbar 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; mglData_prepare2d(&a,0,&v);
}

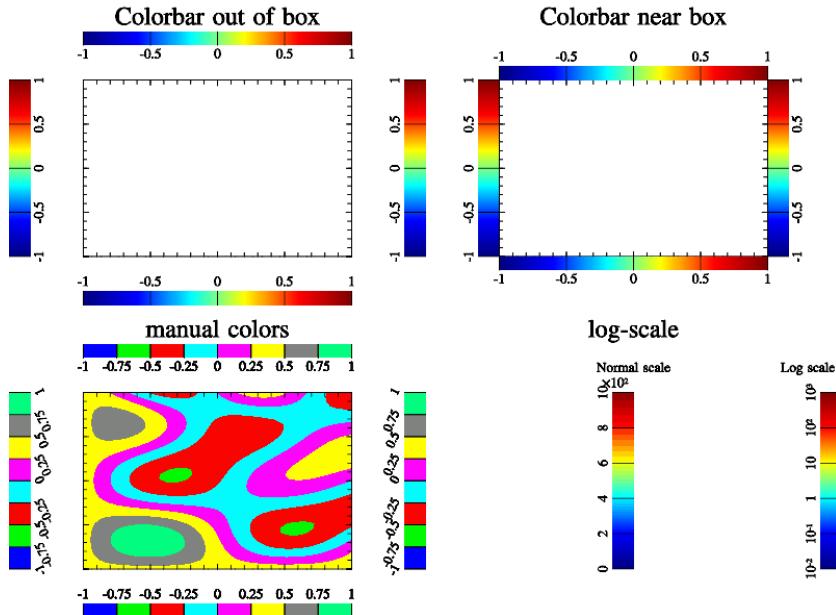
```

```

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("log-scale");
gr->SetRange('c',0.01,1e3);
gr->Colorbar(">",0.5,0); gr->Puts(mglPoint(0,1.2),"Normal scale");
gr->SetFunc("", "", "", "lg(c)");
gr->Colorbar(">"); gr->Puts(mglPoint(1.3,1.2),"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()` 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 86](#)). 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()` features is:

```

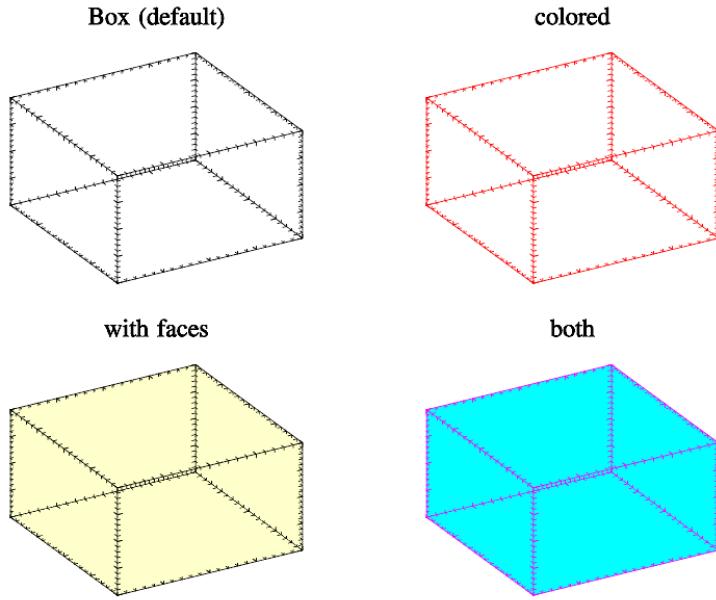
int sample(mglGraph *gr)
{
    gr->SubPlot(2,2,0); gr->Title("Box (default)"); gr->Rotate(50,60);
    gr->Box();
    gr->SubPlot(2,2,1); gr->Title("colored"); gr->Rotate(50,60);
    gr->Box("r");
    gr->SubPlot(2,2,2); gr->Title("with faces"); gr->Rotate(50,60);
    gr->Box("@");
}

```

```

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` 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#");
}

```

```

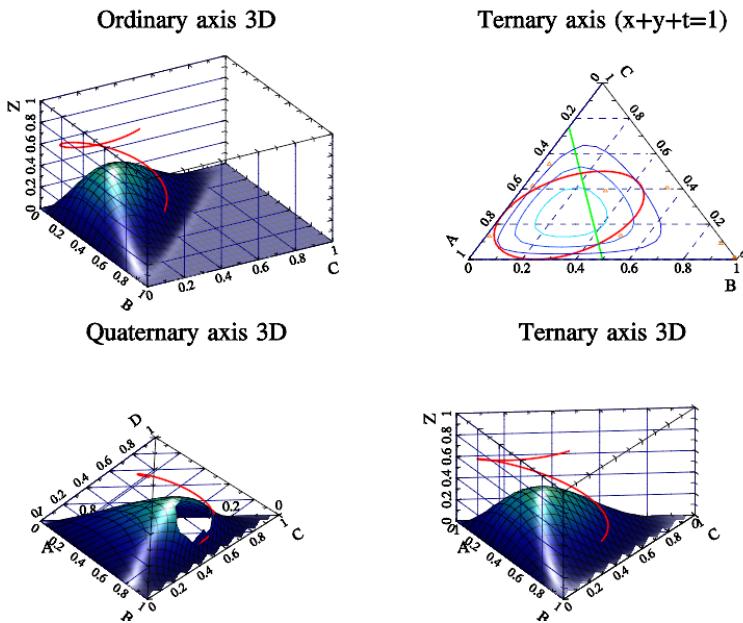
gr->Axis(); gr->Grid(); gr->Box();
gr->Label('x',"B",1); gr->Label('y',"C",1); gr->Label('z',"Z",1);

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 `Puts`) and for its automatic selection (like `Label`, `Legend` and so on). MathGL prints text always in specified position even if it lies outside the bounding box. The default size of font is specified by functions `SetFontFontSize*` (see [Section 4.2.6 \[Font settings\], page 119](#)). 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 110](#)).

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’.

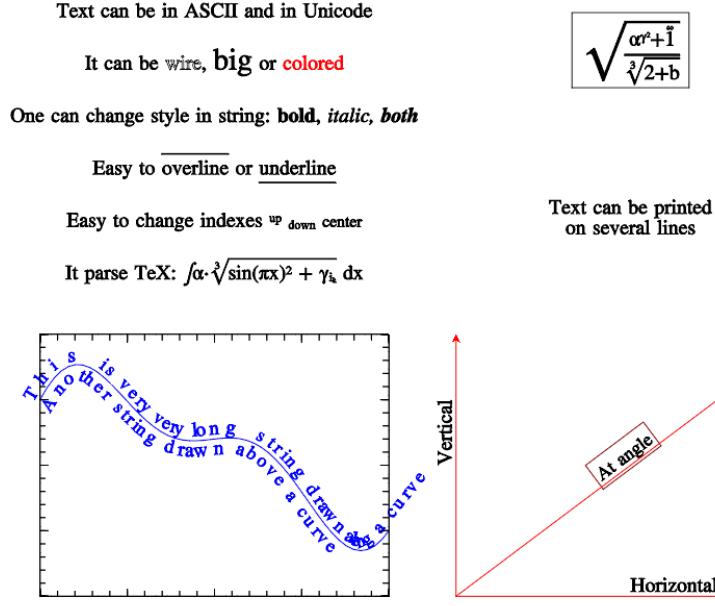
Example of MathGL font drawing is:

```
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 "
    "\\sqrt3{sin(pi x)^2 + \\gamma_{i_k}} dx");

    gr->SubPlot(2,2,1,"");
    gr->Puts(mglPoint(0,0.5), "\\sqrt{\\frac{\\alpha^2}{\\gamma^2}+\\overset 1{\\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 above 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;
}
```



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` function (see [Section 4.9 \[Legend\], page 143](#)); or use 'legend' option (see [Section 3.7 \[Command options\], page 113](#)), 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` 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` 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:

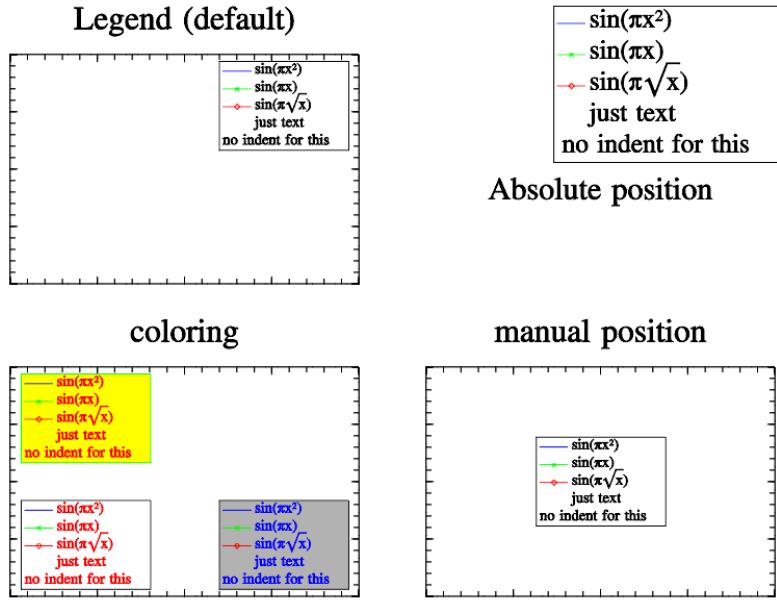
```
int sample(mglGraph *gr)
{
    gr->AddLegend("sin(\\"pi {x^2})","b");
    gr->AddLegend("sin(\\"pi x)","g*");
    gr->AddLegend("sin(\\"pi \\"sqrt{x})","rd");
    gr->AddLegend("just text","");
    gr->AddLegend("no indent for this","");
    gr->SubPlot(2,2,0,"");
    gr->Title("Legend (default)");
    gr->Box();
    gr->Legend(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(0.5,0.5);
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");
}

```

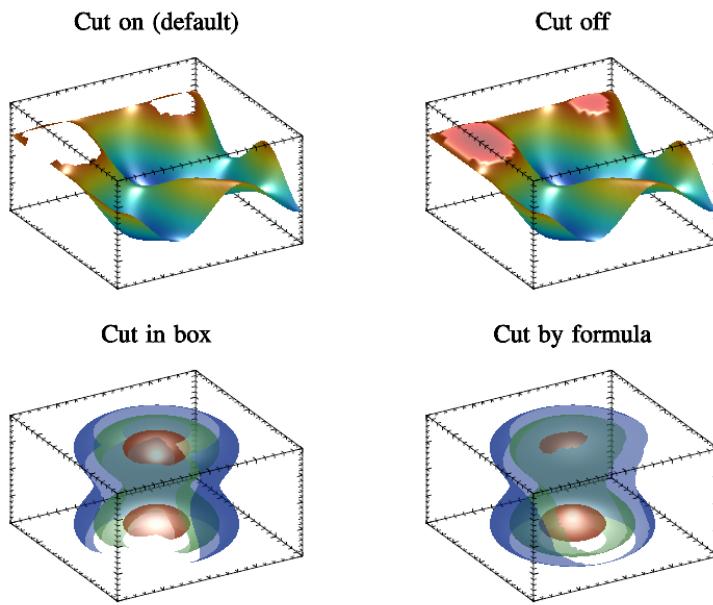
```

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 191](#)). There are several matters why I use class `mglData` but not a single array: it does not depend on type of data (float 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 30×40 is presented as flat (1d-) array with length $30 \times 40 = 1200$ ($\text{nx}=30$, $\text{ny}=40$). The element with indexes $\{i,j\}$ is $a[i + \text{nx} * j]$. 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 `float 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(float **dat, int N1, int N2);`. For C this is functions like `mgl_data_set_float2(HMDT d, const float **dat, int N1, int N2);`. At this, you should keep in mind that $\text{nx}=N2$ and $\text{ny}=N1$ 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 [Chapter 8 \[mglDataA class\], page 222](#)) 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

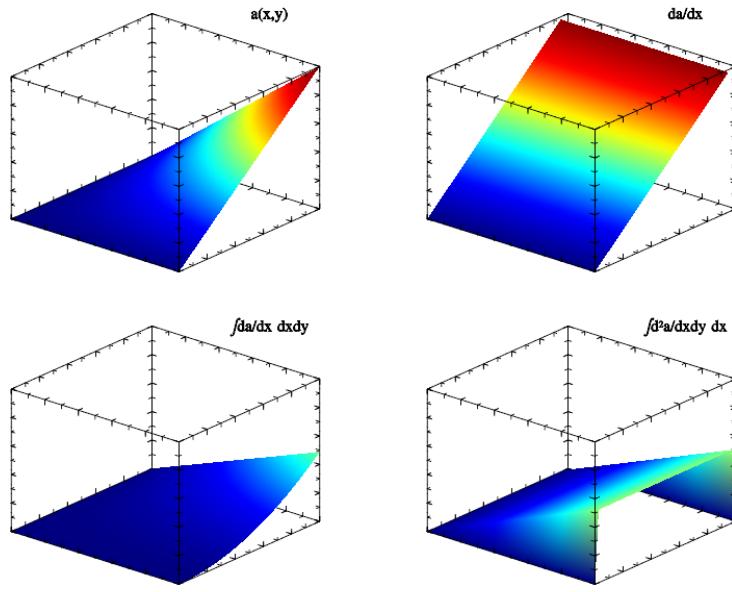
```
double *a = new double[50];
for(int i=0;i<50;i++)    a[i] = sin(M_PI*i/49.);

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 191](#)). 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 dx");
    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()`) 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*rnd");■

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

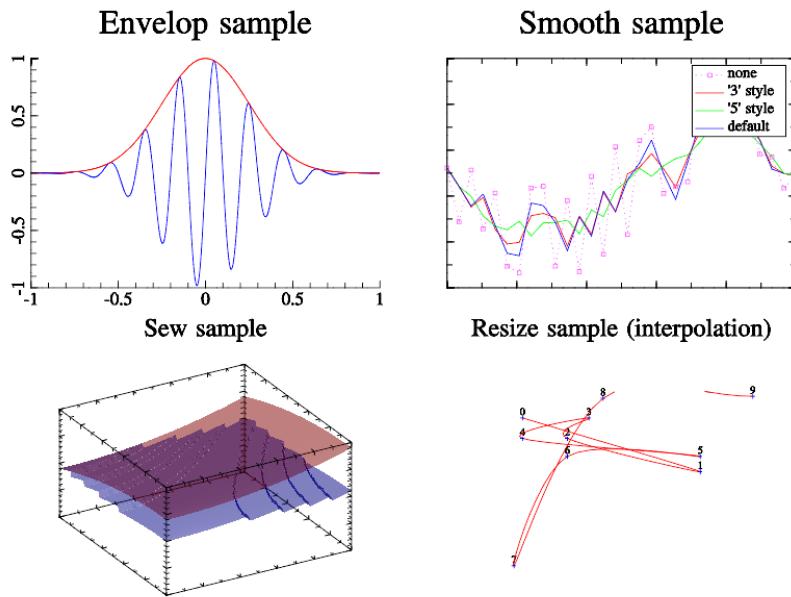
    gr->Plot(y0,"{m7}:s", "legend 'none'"); //gr->AddLegend("none", "k");
    gr->Plot(y1,"r", "legend ''3' style'");
    gr->Plot(y2,"g", "legend ''5' style'");
    gr->Plot(y3,"b", "legend 'default'");
    gr->Legend(); gr->Box();
```

```

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;
}

```



Finally one can create new data arrays on base of the existing one: extract slice, row or column of data (`SubData()`), summarize along a direction(s) (`Sum()`), find distribution of data elements (`Hist()`) and so on.

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 113).

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 108](#)). 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 120](#)).

Below I shall show the features of 1D plotting on base of `Plot()` function (see [\[plot\], page 145](#)). 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()` function. So MathGL uses the solid line with first color of palette (this is blue). Next subplot shows array y_1 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()` and `SubData()` 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()` and `SubData()` 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()` and other 2D plots (see Section 4.11 [2D plotting], page 154) 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 109). 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 interval Min^*Max

```

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()` or `Mesh()` for plots in plain or in volume). However, for isosurfaces (including surfaces of rotation `Axial()`) this flag switches the face drawing off. 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);
    float 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);
    float 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

Plot 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;  mgls_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");

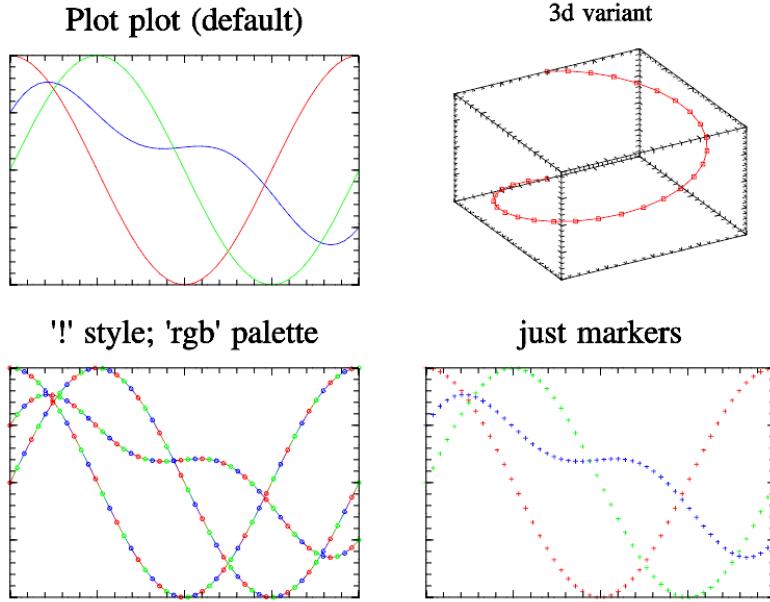
    gr->SubPlot(2,2,3,""); gr->Title("just markers");
    gr->Box();  gr->Plot(y, " +");
}

```

```

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

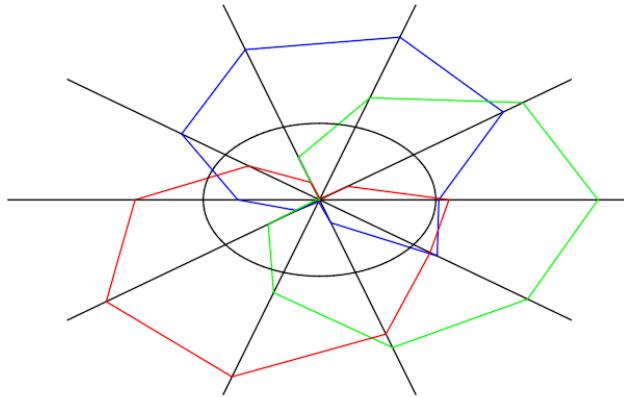
Radar plot is variant of Plot one, which make plot in polar coordinates and draw radial rays in point directions. If you just need a plot in polar coordinates then I recommend to use [Section 2.2.3 \[Curvilinear coordinates\]](#), page 21 or Plot in parabolic form with $x=r\cos(\phi)$; $y=r\sin(\phi)$. 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, '\\#')");
    gr->Radar(yr,"#");
    return 0;
}

```

Radar plot (with grid, '#')



2.5.3 Step sample

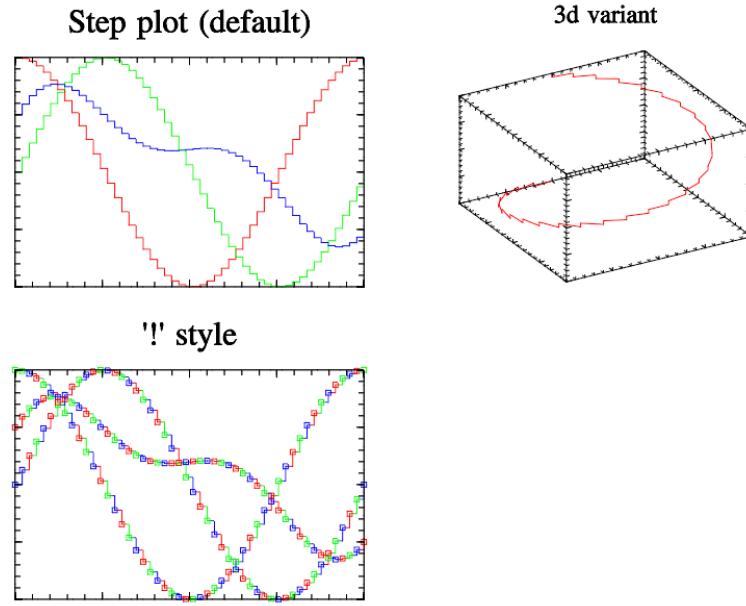
Step plot data as stairs. It have the same options as Plot. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y;  mglData 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;
}
```



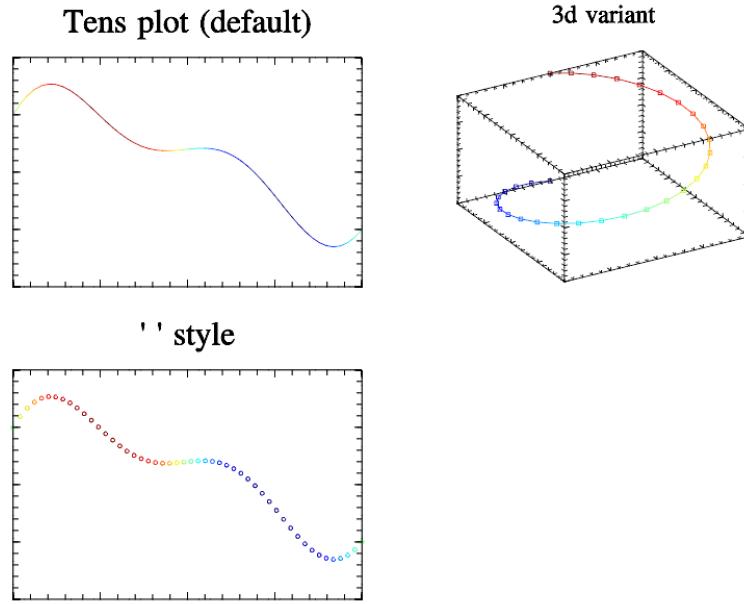
2.5.4 Tens sample

Tens is variant of Plot with smooth coloring along the curves. At this, color is determined as for surfaces (see [Section 3.4 \[Color scheme\], page 109](#)). 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

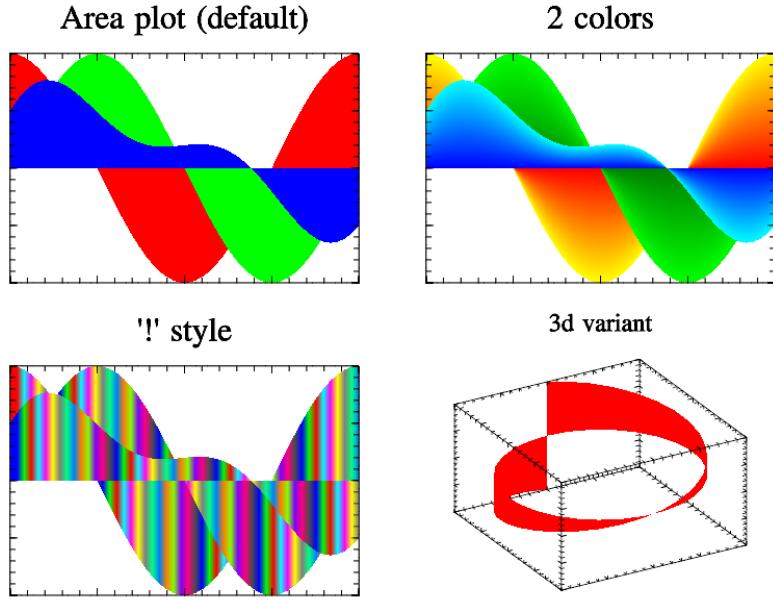
`Area` fill the area between curve and axis plane. It support gradient filling if 2 colors per curve is specified. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y;  mglS_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");
    return 0;
}
```



2.5.6 Region sample

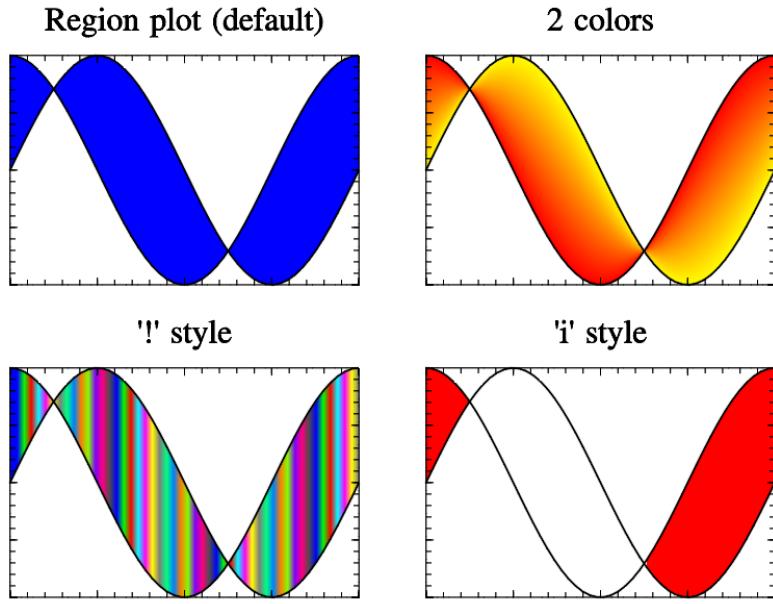
Region fill the area between 2 curves. It support gradient filling if 2 colors per curve is specified. Also it can fill only the region $y1 < y < y2$ if style ‘i’ is used. The sample code is:

```
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("!' style");
    gr->Box();  gr->Region(y1,y2,"!"); gr->Plot(y1,"k2"); gr->Plot(y2,"k2");

    gr->SubPlot(2,2,3,""); gr->Title("i' style");
    gr->Box();  gr->Region(y1,y2,"ir"); gr->Plot(y1,"k2"); gr->Plot(y2,"k2");
    return 0;
}
```



2.5.7 Stem sample

Stem draw vertical bars. It is most attractive if markers are drawn too. The sample code is:

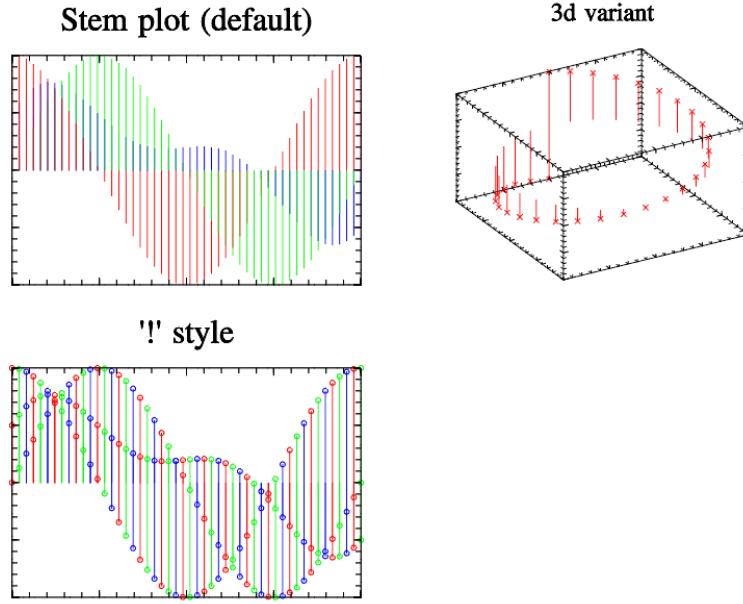
```

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("!' style");
    gr->Box(); gr->Stem(y,"o!rgb");
    return 0;
}

```



2.5.8 Bars sample

`Bars` 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");

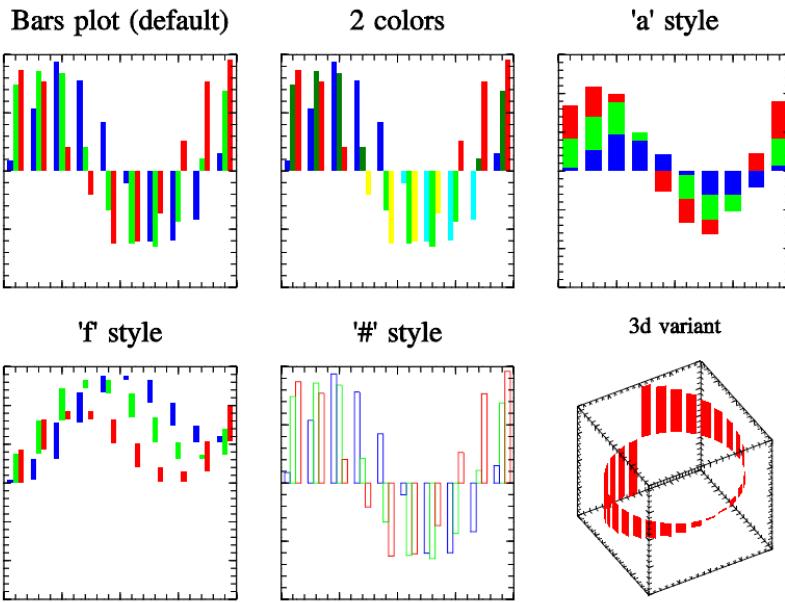
    gr->SubPlot(3,2,4,""); gr->Title("\#\# style");
    gr->Box(); gr->Bars(ys,"##");

    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;
}
```



2.5.9 Barh sample

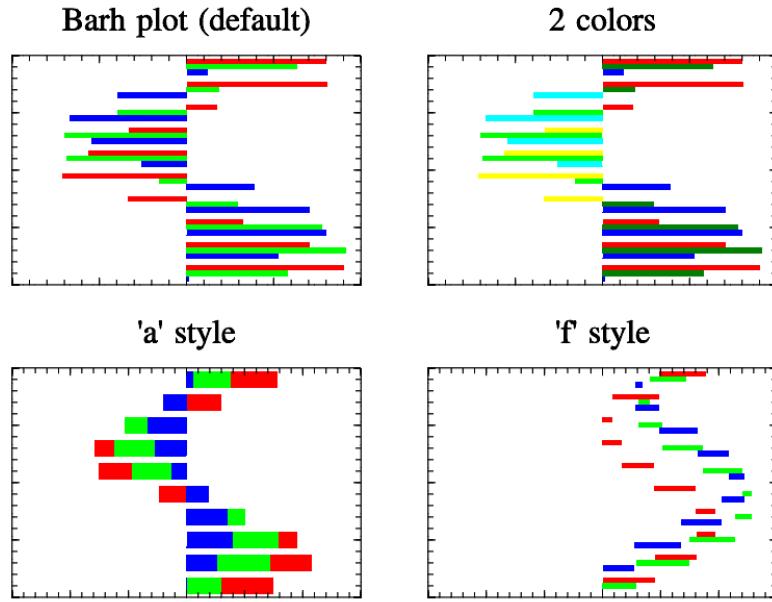
Barh is the similar to Bars but draw horizontal bars. The sample code is:

```
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

Cones is similar to Bars but draw cones. The sample code is:

```

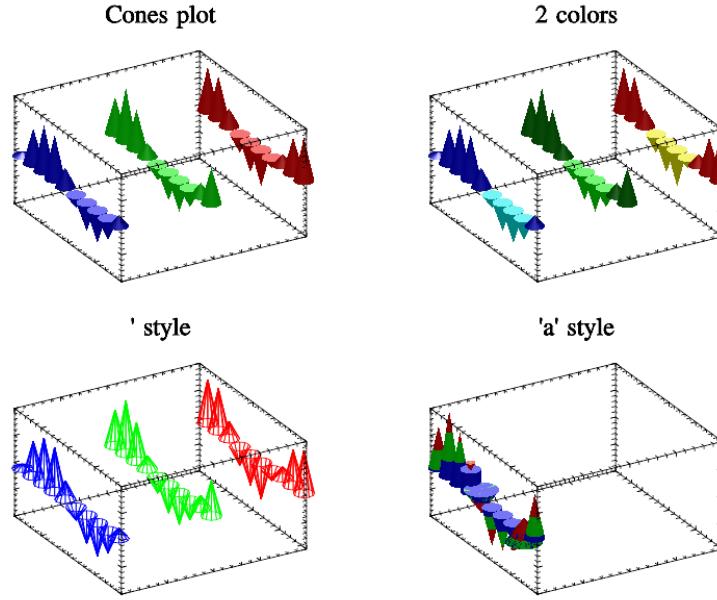
int sample(mglGraph *gr)
{
    mglData ys(10,3);    ys.Modify("0.8*sin(pi*(2*x+y/2))+0.2*rnd");
    gr->Light(true);    gr->SetOrigin(0,0,0);
    gr->SubPlot(2,2,0); gr->Title("Cones plot");
    gr->Rotate(50,60);  gr->Box();   gr->Cones(ys);

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

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

    gr->SubPlot(2,2,3); 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");
    return 0;
}

```



2.5.11 Chart sample

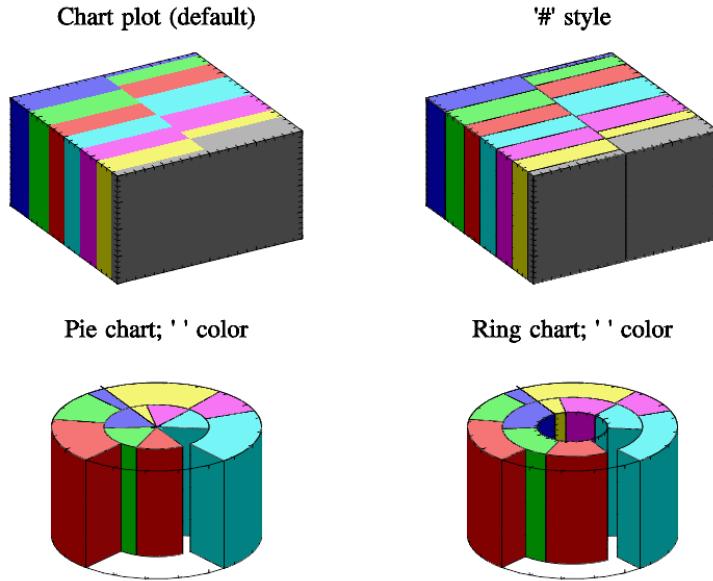
Chart draw colored boxes with width proportional to data values. Use ‘ ’ for empty box. Plot looks most attractive in polar coordinates – well known pie chart. The sample code is:

```
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;
}
```

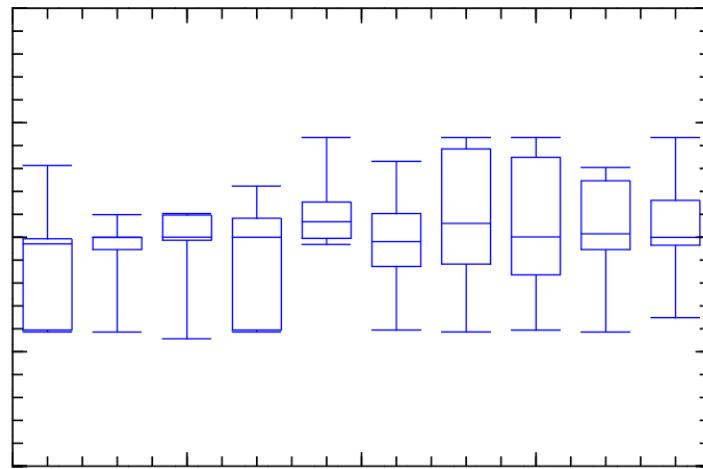


2.5.12 BoxPlot sample

BoxPlot draw box-and-whisker diagram. The sample code is:

```
int sample(mglGraph *gr)
{
    mg1Data 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

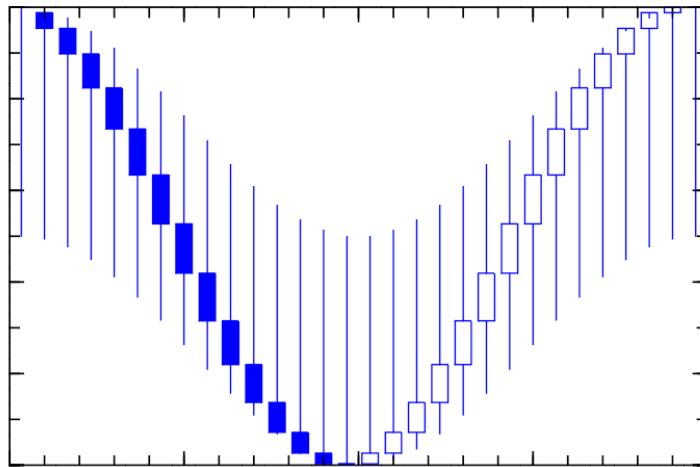


2.5.13 Candle sample

Candle 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 Error sample

Error draw error boxes around the points. You can draw default boxes or semi-transparent symbol (like marker, see Section 3.3 [Line styles], page 108). Also you can set individual color for each box. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y; mglS_prepare1d(&y);
    mglData x0(10), y0(10), ex0(10), ey0(10);
    float 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;
}

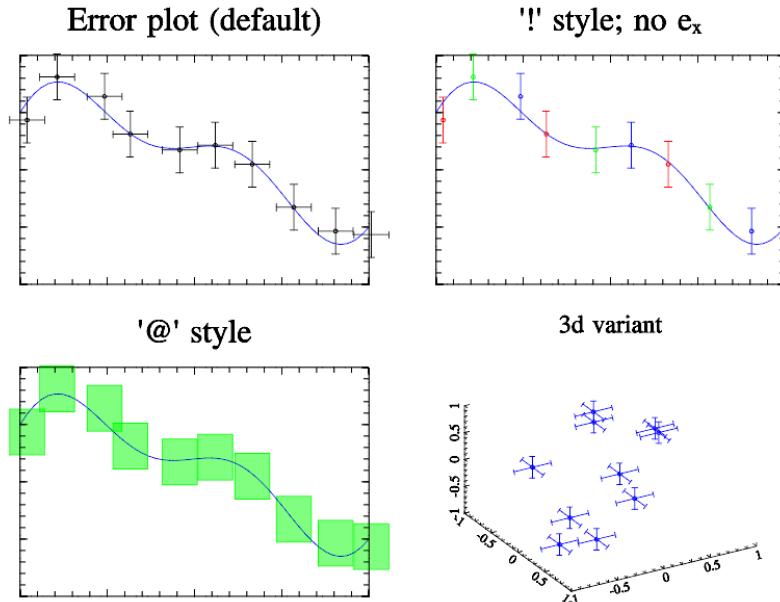
gr->SubPlot(2,2,0,""); gr->Title("Error plot (default)");
gr->Box(); gr->Plot(y.SubData(-1,0)); gr->Error(x0,y0,ex0,ey0,"ko");

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;
}

```



2.5.15 Mark sample

Mark draw markers at points. It is mostly the same as Plot but marker size can be variable. The sample code is:

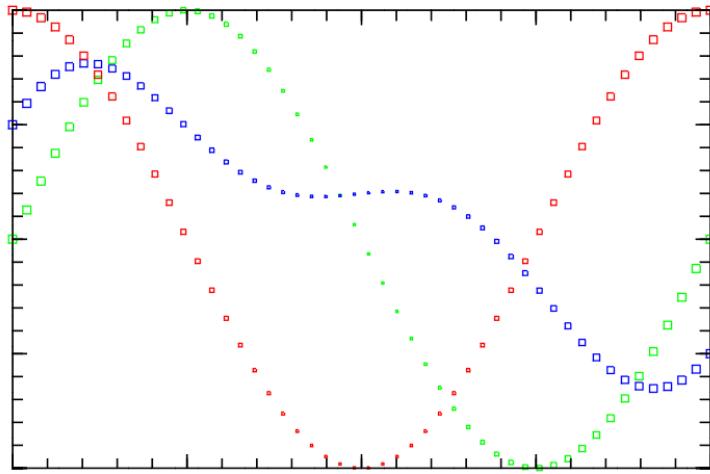
```

int sample(mglGraph *gr)
{
    mgldata y,y1; mgl_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)

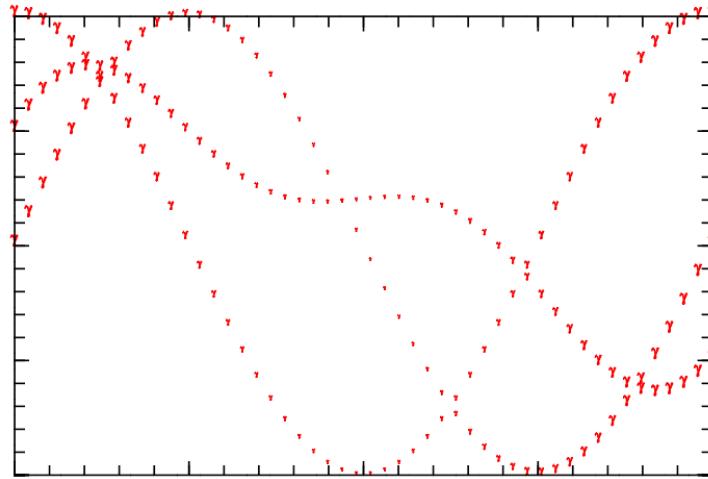


2.5.16 TextMark sample

TextMark like Mark but draw text instead of markers. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData y,y1; mglS_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)

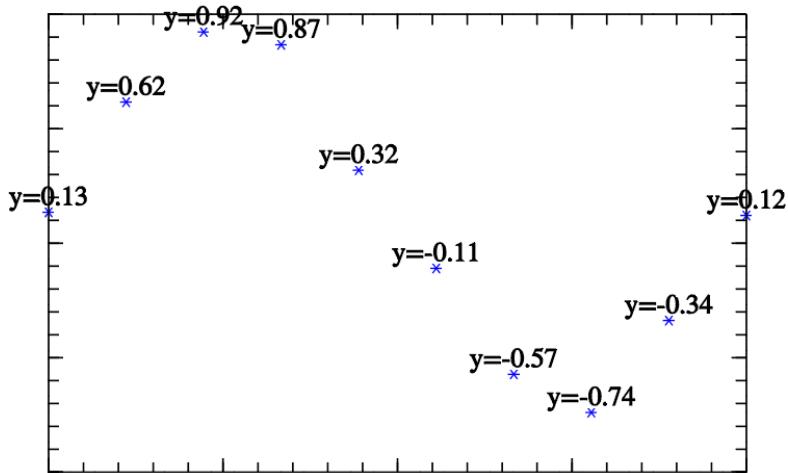


2.5.17 Label sample

Label 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



2.5.18 Tube sample

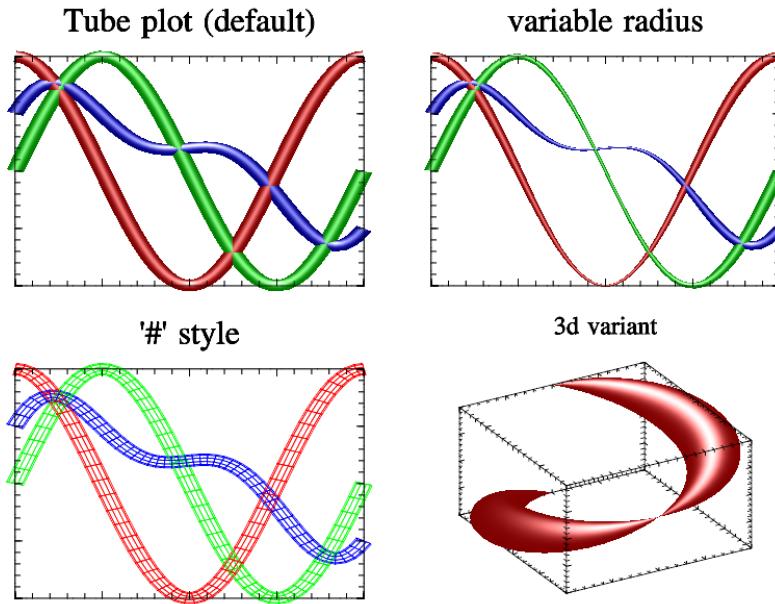
Tube draw tube with variable radius. The sample code is:

```
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;
}
```



2.5.19 Tape sample

Tape 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);
```

```

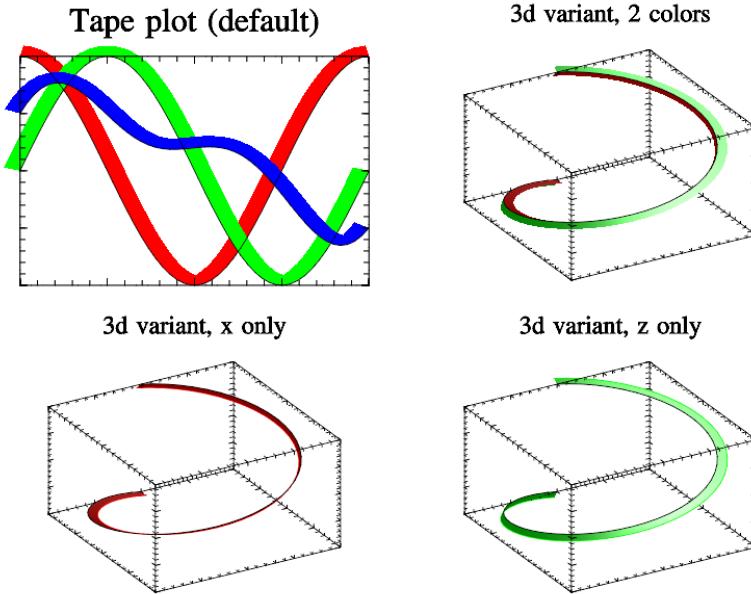
yc.Modify("sin(pi*(2*x-1))");
xc.Modify("cos(pi*2*x-pi)"); z.Fill(-1,1);
gr->SubPlot(2,2,0,""); gr->Title("Tape plot (default)");
gr->Box(); gr->Tape(y); gr->Plot(y,"k");

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;
}

```



2.5.20 Torus sample

Torus 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;
}

```

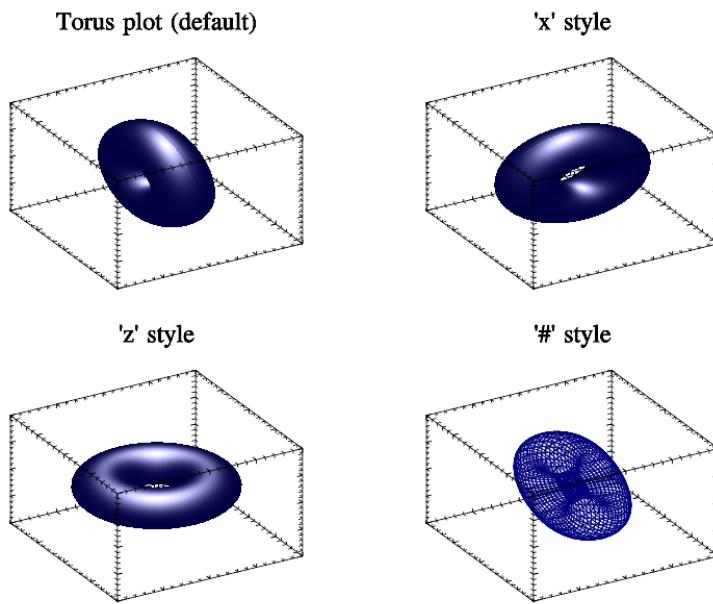
```

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;
}

```



2.6 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); }
    float 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'); }
    float 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

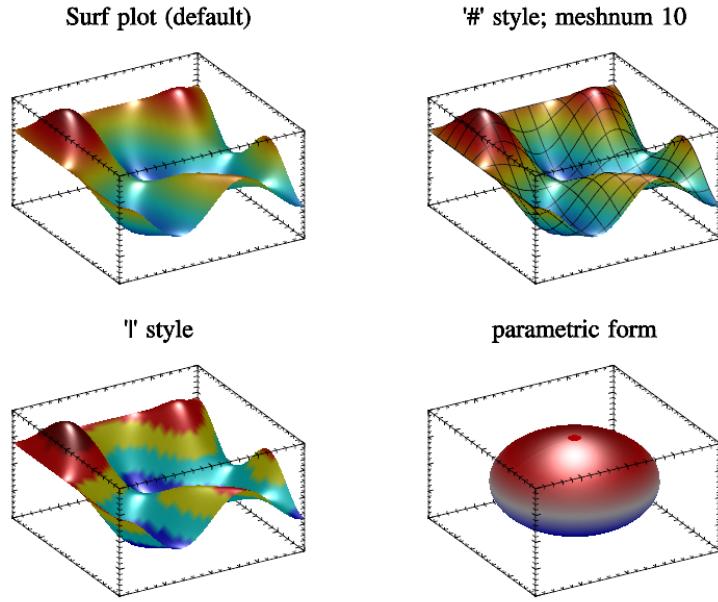
`Surf` is most standard way to visualize 2D data array. `Surf` use color scheme for coloring (see [Section 3.4 \[Color scheme\], page 109](#)). You can use '#' style for drawing black meshes on the surface. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a; mgls_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;
}
```

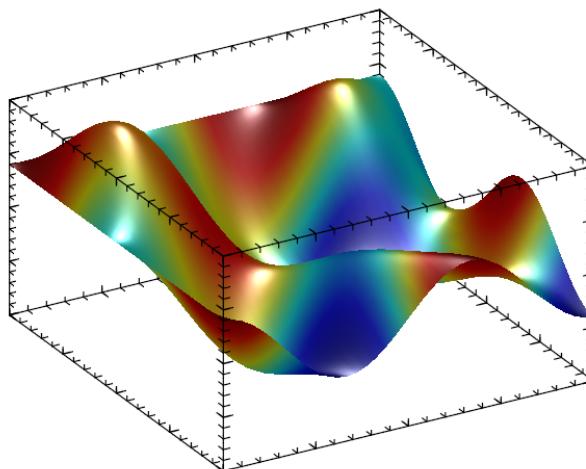


2.6.2 SurfC sample

SurfC is similar to Surf but its coloring is determined by another data. The sample code is:

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

SurfC plot

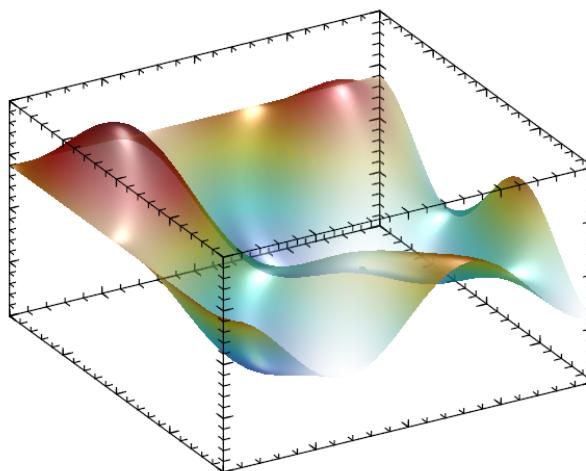


2.6.3 SurfA sample

`SurfA` is similar to `Surf` 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

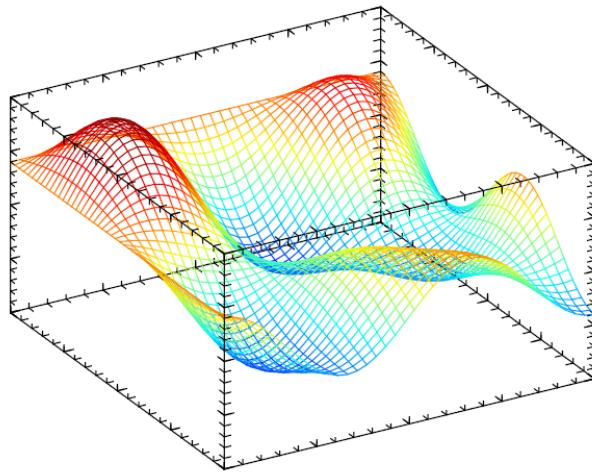


2.6.4 Mesh sample

`Mesh` draw wired surface. You can use `SetMeshNum` 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

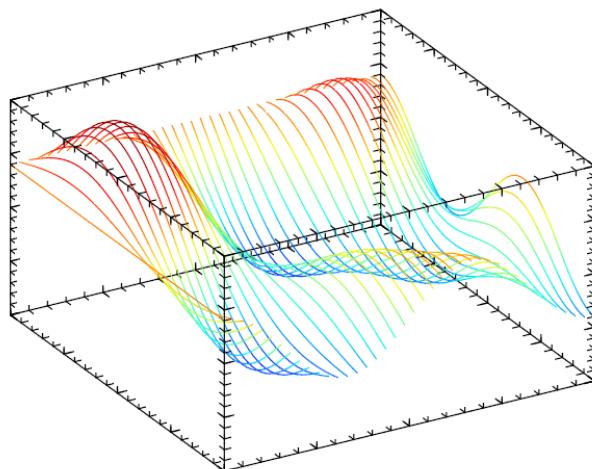


2.6.5 Fall sample

Fall draw waterfall surface. You can use `SetMeshNum` 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; mglS_prepare2d(&a);
    gr->Title("Fall plot"); gr->Rotate(50,60);
    gr->Box(); gr->Fall(a);
    return 0;
}
```

Fall plot

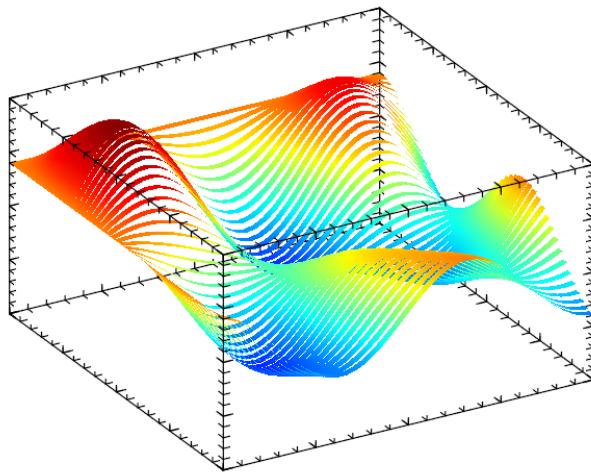


2.6.6 Belt sample

Belt 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



2.6.7 Boxs sample

Boxs draw surface by boxes. You can use ‘#’ for drawing wire plot. The sample code is:

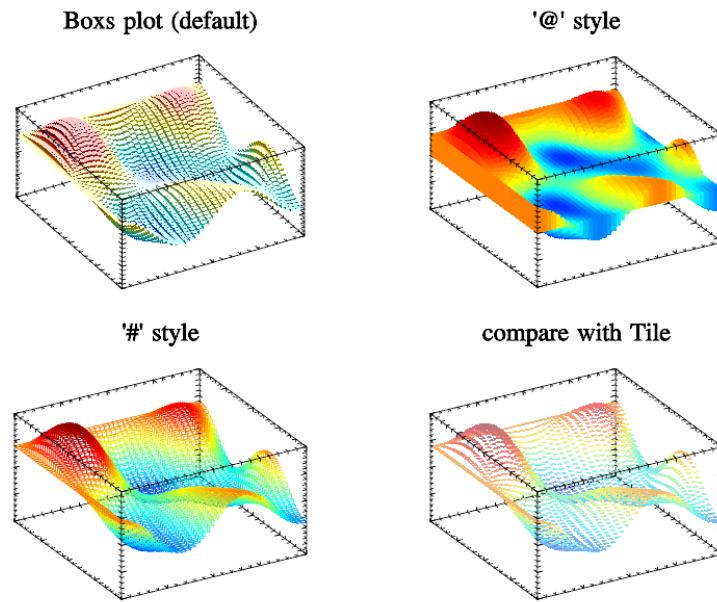
```
int sample(mglGraph *gr)
{
    mglData a;  mglS_prepare2d(&a);
    gr->SetOrigin(0,0,0); gr->Light(true);
    gr->SubPlot(2,2,0);  gr->Title("Boxs plot (default)");
    gr->Rotate(40,60);   gr->Box();  gr->Boxs(a);

    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;  
}
```

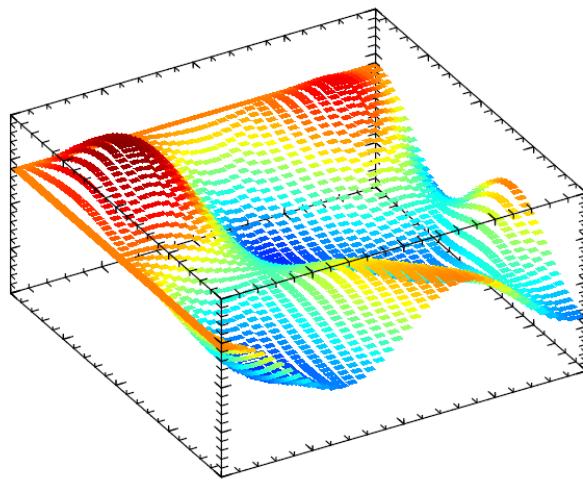


2.6.8 Tile sample

Tile 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

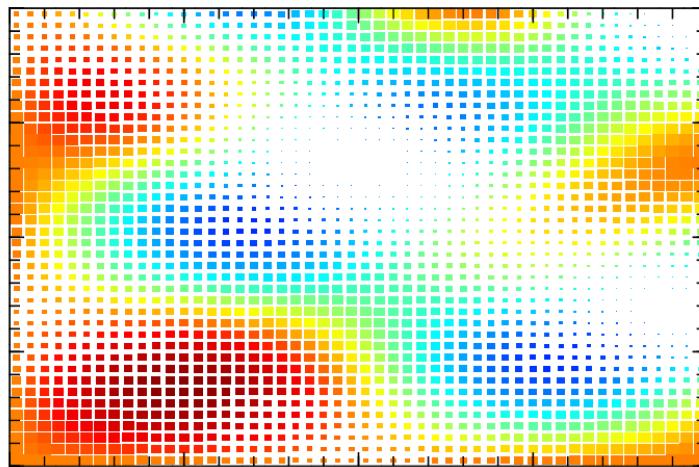


2.6.9 TileS sample

TileS is similar to Tile 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



2.6.10 Dens sample

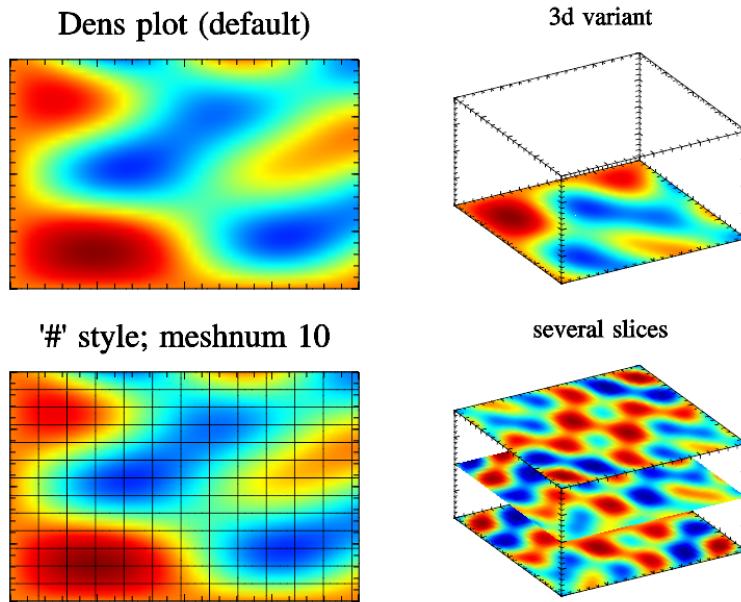
Dens draw density plot for surface. The sample code is:

```
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;
}
```



2.6.11 Cont sample

Cont 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)");
```

```

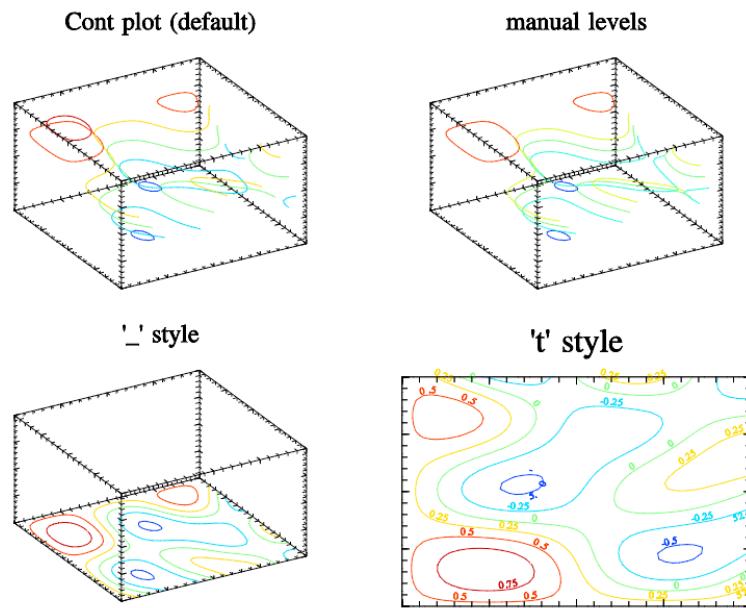
gr->Rotate(50,60); gr->Box(); gr->Cont(a);

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;
}

```



2.6.12 ContF sample

ContF 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); 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("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);

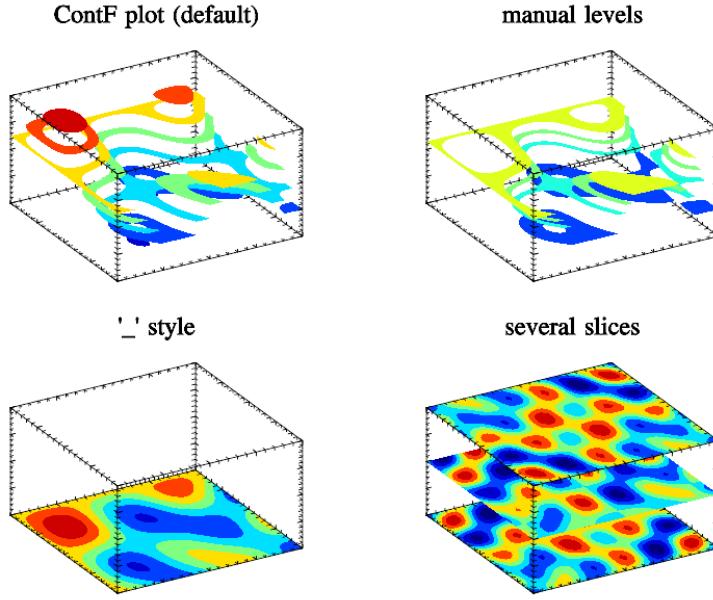
    gr->SubPlot(2,2,2); gr->Title("\_\_ style");
    gr->Rotate(50,60); gr->Box(); gr->ContF(a,"_");
}

```

```

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

```



2.6.13 ContD sample

ContD is similar to ContF but with manual contour colors. The sample code is:

```

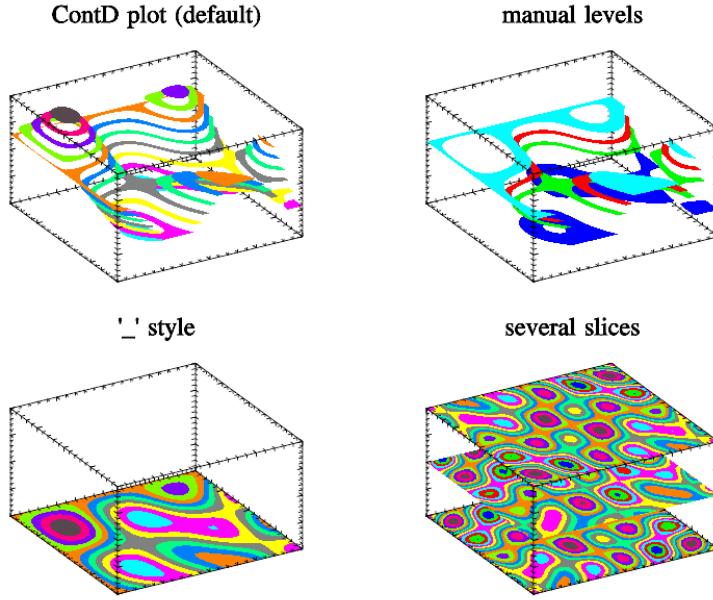
int sample(mglGraph *gr)
{
    mgldata a,v(5),a1(30,40,3); mglS_prepare2d(&a); v.a[0]=-0.5;
    v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v.a[4]=0.5;
    gr->SubPlot(2,2,0); gr->Title("ContD plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->ContD(a);

    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;
}

```



2.6.14 ContV sample

ContV draw vertical cylinders (belts) at contour lines. The sample code is:

```

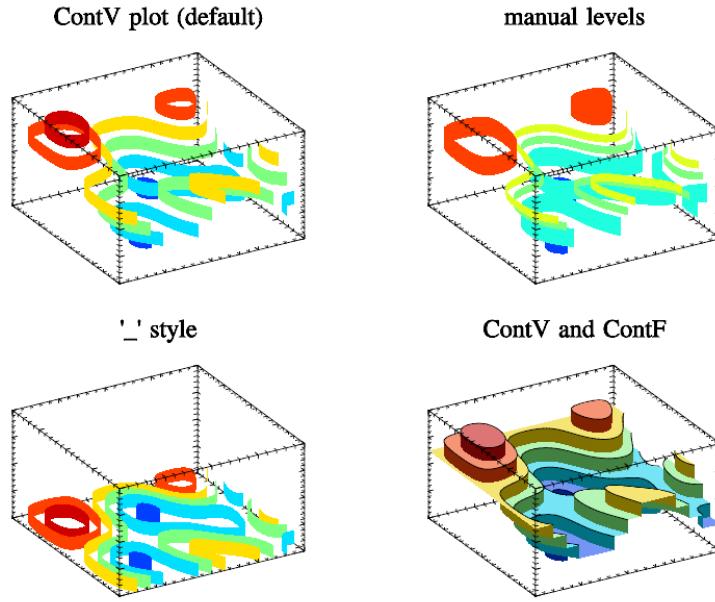
int sample(mglGraph *gr)
{
    mgldata a,v(5); mglS_prepare2d(&a); v.a[0]=-0.5;
    v.a[1]=-0.15; v.a[2]=0; v.a[3]=0.15; v.a[4]=0.5;
    gr->SubPlot(2,2,0); gr->Title("ContV plot (default)");
    gr->Rotate(50,60); gr->Box(); gr->ContV(a);

    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;
}

```



2.6.15 Axial sample

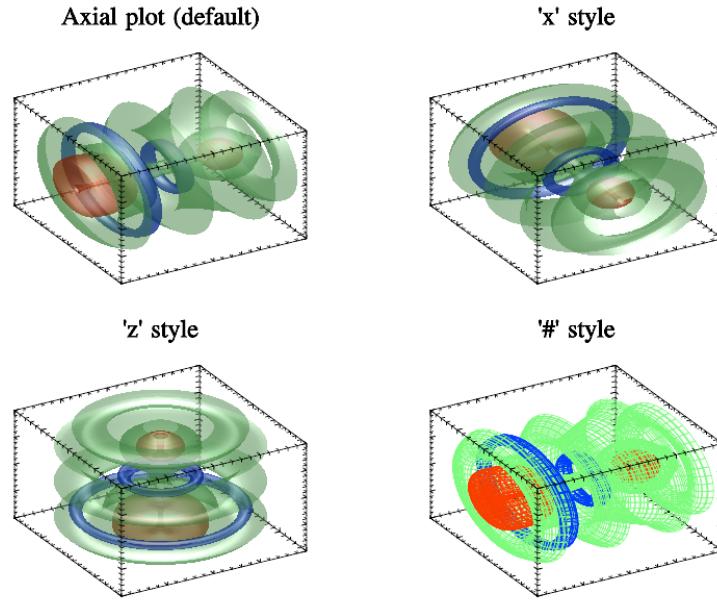
Axial draw surfaces of rotation for contour lines. You can draw wire surfaces ('#' style) or ones rotated in other directions ('x', 'z' styles). The sample code is:

```
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"); 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;
}
```

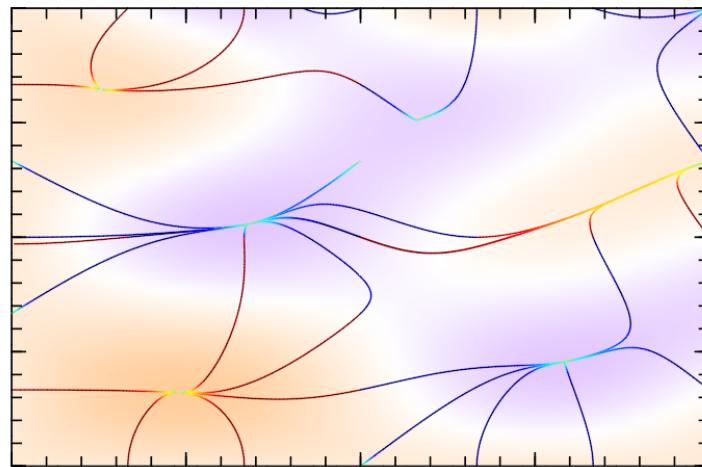


2.6.16 Grad sample

Grad 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 $a(i,j,k)=a(x(i),y(j),z(k))$, $i=1\dots n$, $j=1\dots m$, $k=1\dots l$ 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 mgls_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);
    float 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

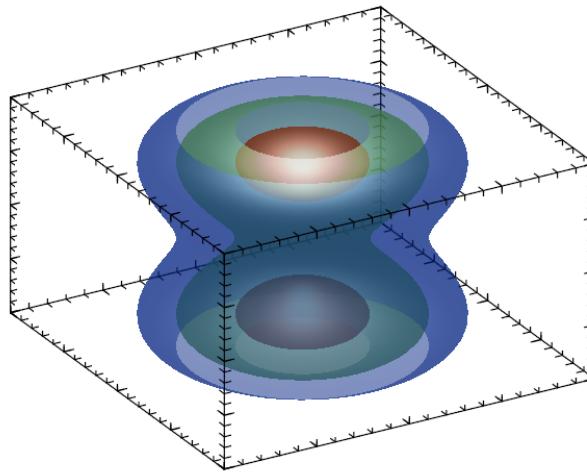
```
void mgls_prepare3d(HMDT a, HMDT b=0)
{
    register long i,j,k,n=61,m=50,l=40,i0;
    if(a) mgl_data_create(a,n,m,l);
    if(b) mgl_data_create(b,n,m,l);
    float x,y,z;
    for(i=0;i<n;i++)  for(j=0;j<m;j++)  for(k=0;k<l;k++)
    {
        x=2*i/(n-1.)-1; y=2*j/(m-1.)-1; z=2*k/(l-1.)-1; i0 = i+n*(j+m*k);
        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

Surf3 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; mgls_prepare3d(&c);
    gr->Title("Surf3 plot"); gr->Rotate(50,60);
    gr->Light(true); gr->Alpha(true);
    gr->Box(); gr->Surf3(c);
    return 0;
}
```

Surf3 plot

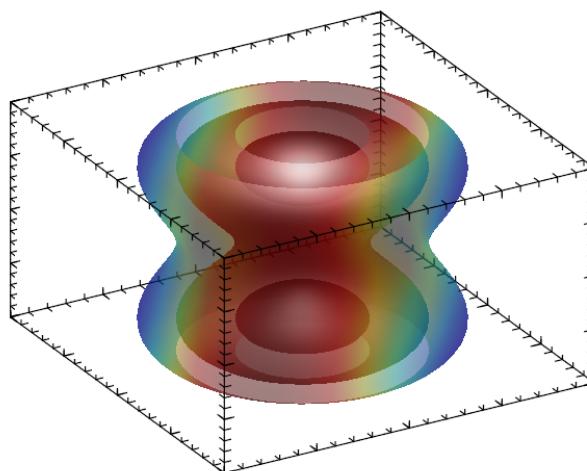


2.7.2 Surf3C sample

Surf3C is similar to Surf3 but its coloring is determined by another data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c,d;  mglS_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

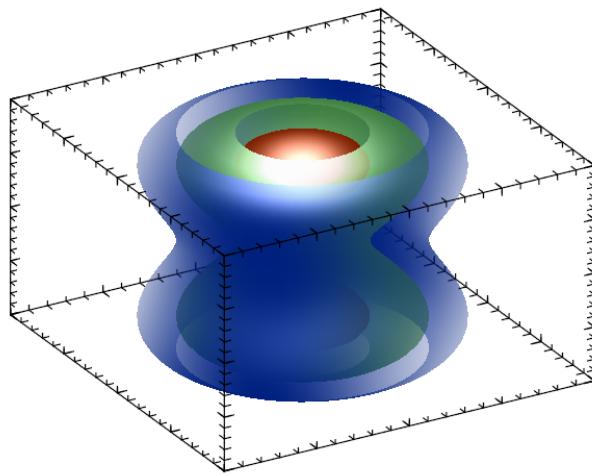


2.7.3 Surf3A sample

Surf3A is similar to Surf3 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 Cloud sample

Cloud 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");

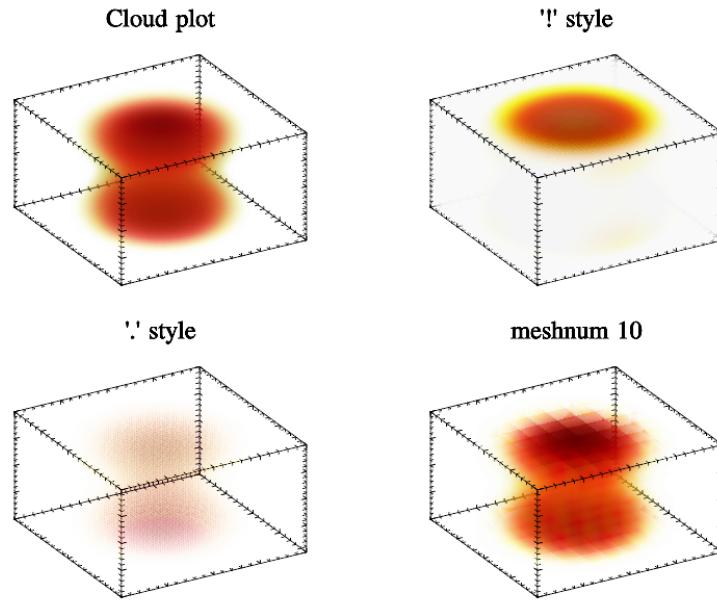
    gr->SubPlot(2,2,1); gr->Title("!' style");
    gr->Rotate(50,60); gr->Box(); gr->Cloud(c,"!wyrRk");

    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;
}

```



2.7.5 Dens3 sample

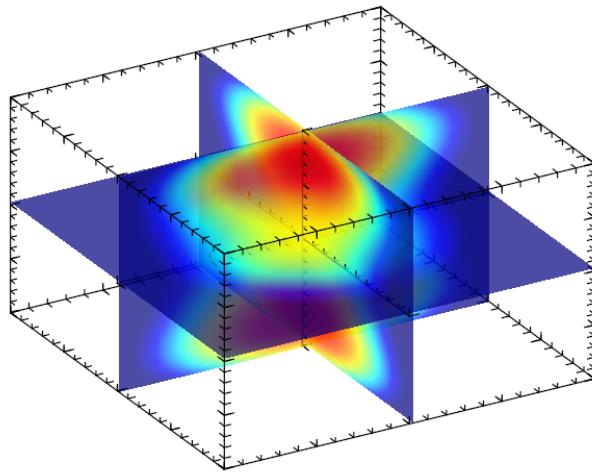
Dens3 draw just usual density plot but at slices of 3D data. The sample code is:

```

int sample(mglGraph *gr)
{
    mglData c; mglS_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

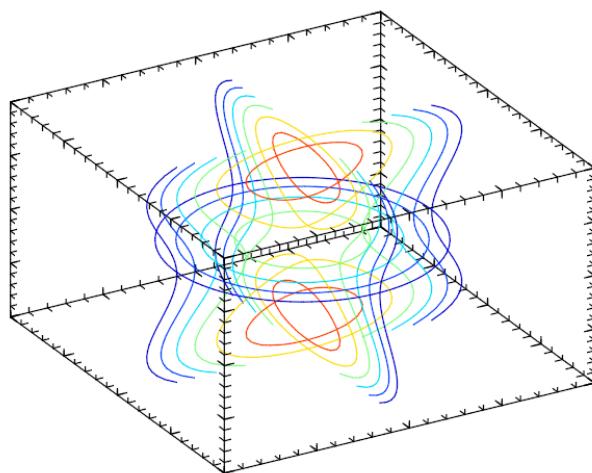


2.7.6 Cont3 sample

Cont3 draw just usual contour lines but at slices of 3D data. The sample code is:

```
int sample(mglGraph *gr)
{
    mglData c;  mglS_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

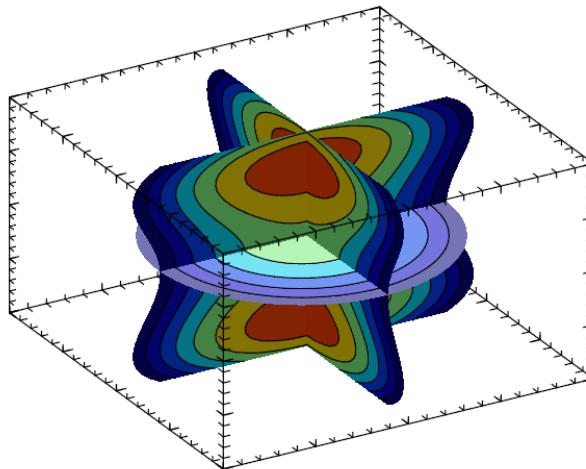


2.7.7 ContF3 sample

ContF3 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

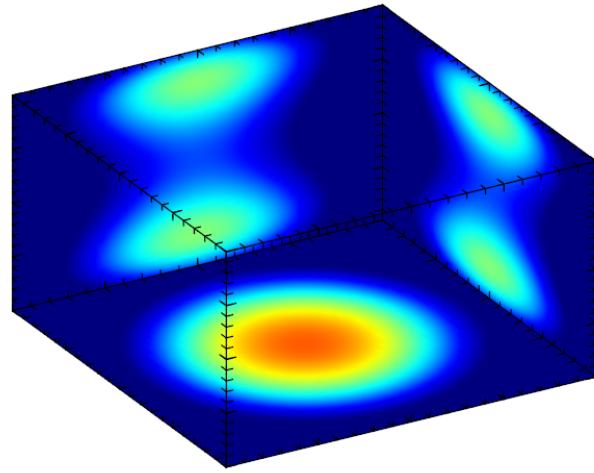


2.7.8 Dens projection sample

Dens[XYZ] 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; mglS_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

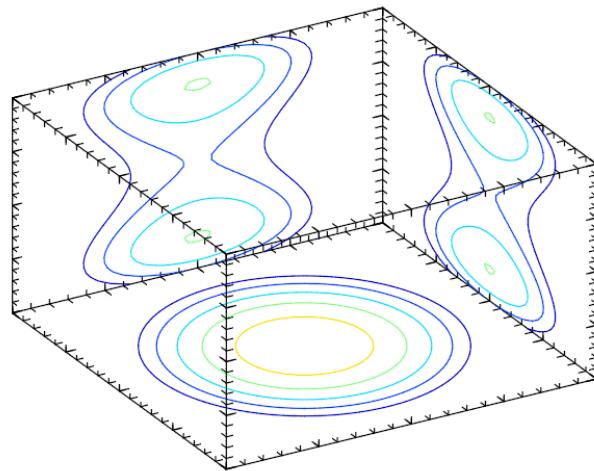


2.7.9 Cont projection sample

Cont [XYZ] 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

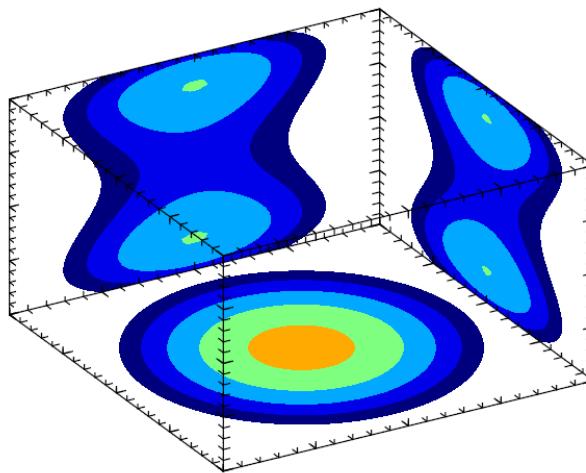


2.7.10 ContF projection sample

`Dens[XYZ]` 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; mglData 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



2.7.11 TriPlot and QuadPlot

`TriPlot` and `QuadPlot` 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)
{
    float 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};
    float 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,
    mglData qq(6,4,q), xx(8,xc), yy(8,yc), zz(8,zc);
    gr->Light(true); //gr->Alpha(true);
    gr->SubPlot(2,1,0); gr->Title("QuadPlot sample"); gr->Rotate(50,60);
    gr->QuadPlot(qq,xx,yy,zz,"yr");
    gr->QuadPlot(qq,xx,yy,zz,"k#");

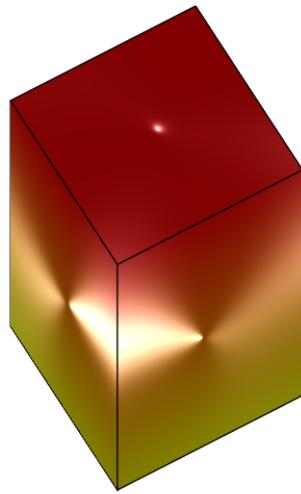
    float t[] = {0,1,2, 0,1,3, 0,2,3, 1,2,3};
```

```

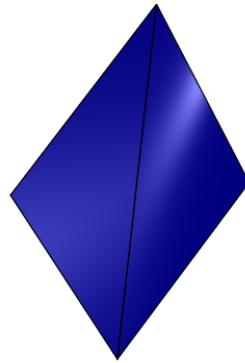
float xt[] = {-1,1,0,0}, yt[] = {-1,-1,1,0}, zt[] = {-1,-1,-1,1};
mglData tt(4,3,t), uu(4,xt), vv(4,yt), ww(4,zt);
gr->SubPlot(2,1,1); gr->Title("TriPlot sample"); gr->Rotate(50,60);
gr->TriPlot(tt,uu,vv,ww,"b");
gr->TriPlot(tt,uu,vv,ww,"k#");
return 0;
}

```

QuadPlot sample



TriPlot sample



2.7.12 Dots sample

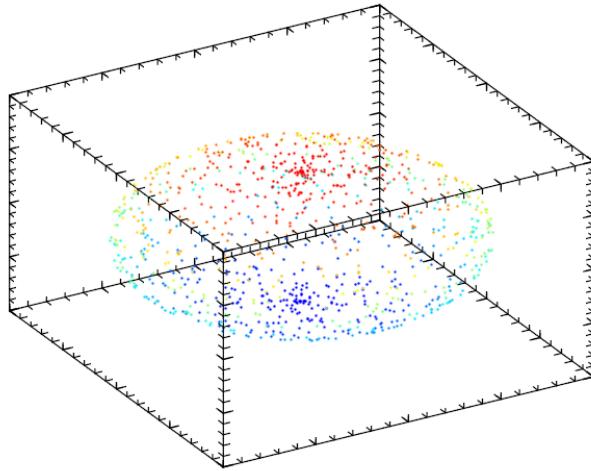
`Dots` is another way to draw irregular points. `Dots` use color scheme for coloring (see Section 3.4 [Color scheme], page 109). The sample code is:

```

int sample(mglGraph *gr)
{
    int i, n=1000;
    mglData x(n),y(n),z(n);
    for(i=0;i<n;i++)
    {
        float 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);
    }
    gr->Title("Dots sample"); gr->Rotate(50,60);
    gr->Box(); gr->Dots(x,y,z);
    return 0;
}

```

Dots 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 mgls_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);
    float 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 mgls_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);
    float 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(mglData *a, mglData *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);
    float 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(mglData *ex, mglData *ey, mglData *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);
    float 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

`Vect` 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 109). The sample code is:

```
int sample(mglGraph *gr)
{
    mglData a,b;  mgls_prepare2v(&a,&b);
    gr->SubPlot(3,2,0,""); gr->Title("Vect plot (default)");
    gr->Box();  gr->Vect(a,b);

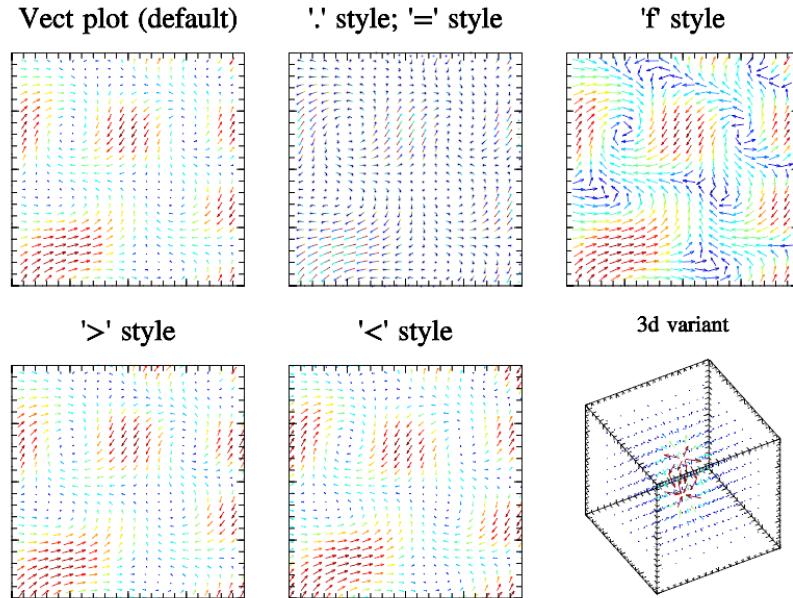
    gr->SubPlot(3,2,1,""); gr->Title("'.' style; '=' style");
    gr->Box();  gr->Vect(a,b,".=.");

    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; mgls_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 Traj sample

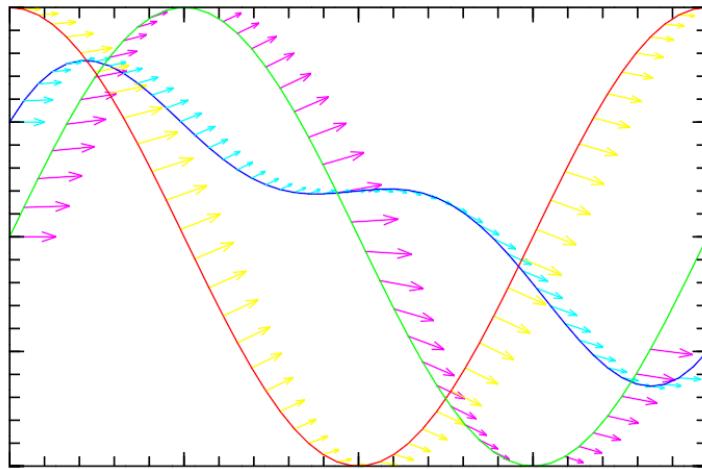
`Traj` is 1D analogue of `Vect`. It draw vectors from specified points. The sample code is:

```

int sample(mglGraph *gr)
{
    mglData x,y,y1,y2;  mglS_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



2.8.3 Flow sample

`Flow` 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 109). 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;  mglS_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,"#");

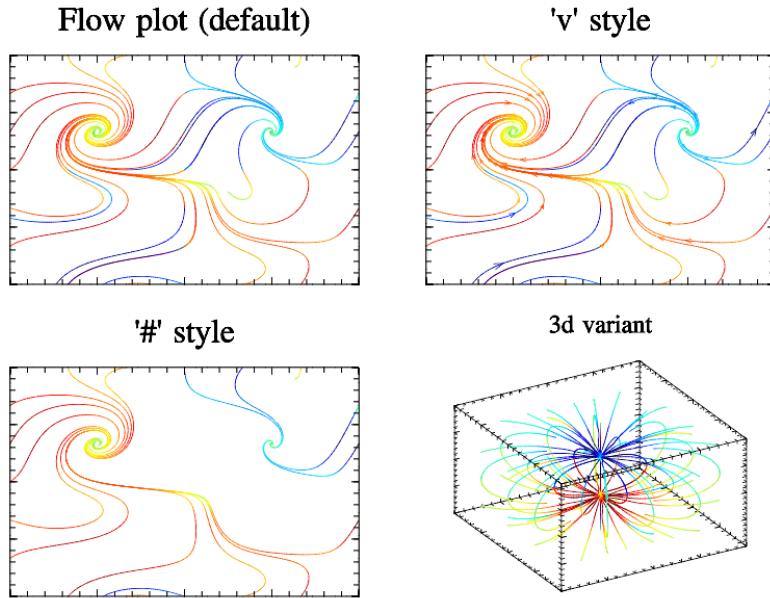
    mglData ex,ey,ez; mglS_prepare3v(&ex,&ey,&ez);
}

```

```

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

```



2.8.4 Pipe sample

Pipe is similar to Flow 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 109](#)). At this warm color corresponds to normal flow (like attractor), cold one corresponds to inverse flow (like source). The sample code is:

```

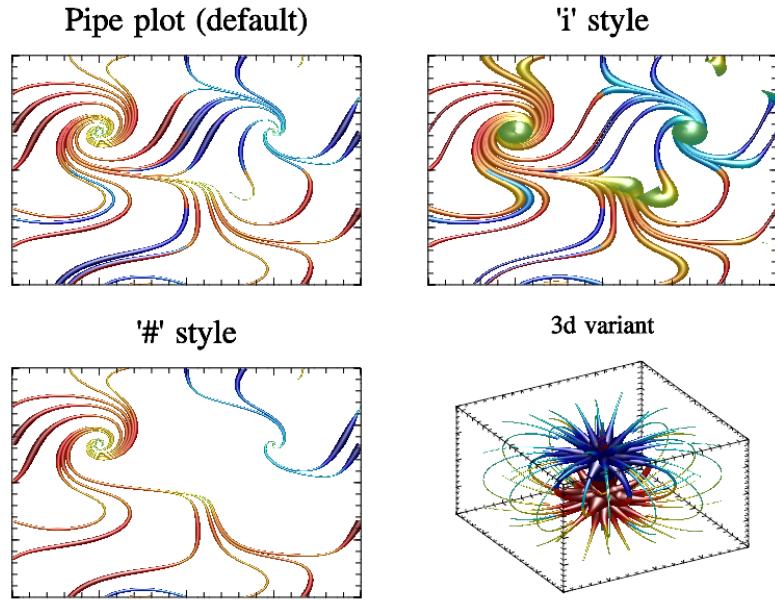
int sample(mglGraph *gr)
{
    mglData a,b;  mgls_prepare2v(&a,&b);
    gr->SubPlot(2,2,0,""); gr->Title("Pipe plot (default)");
    gr->Light(true);  gr->Box();  gr->Pipe(a,b);

    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; mgls_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;
}

```

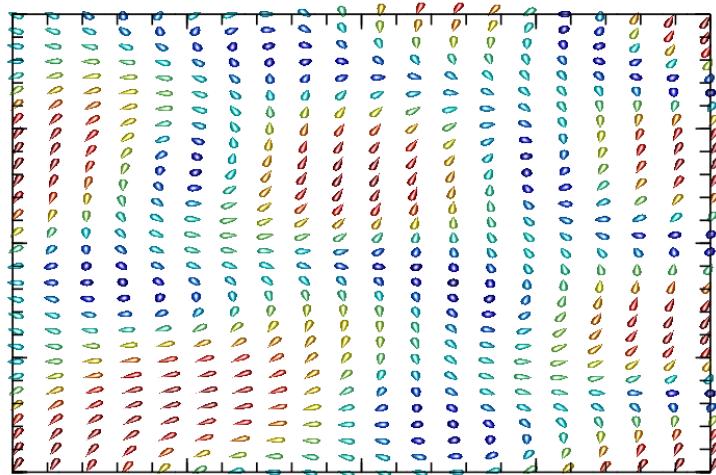


2.8.5 Dew sample

Dew is similar to Vect but use drops instead of arrows. The sample code is:

```
int sample(mglGraph *gr)
{
    mgldata a,b; mgl_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);
```

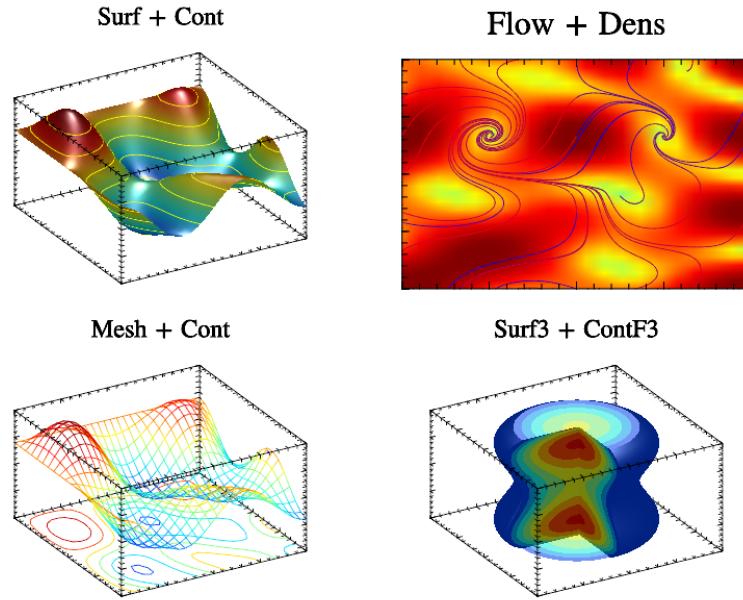
```

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

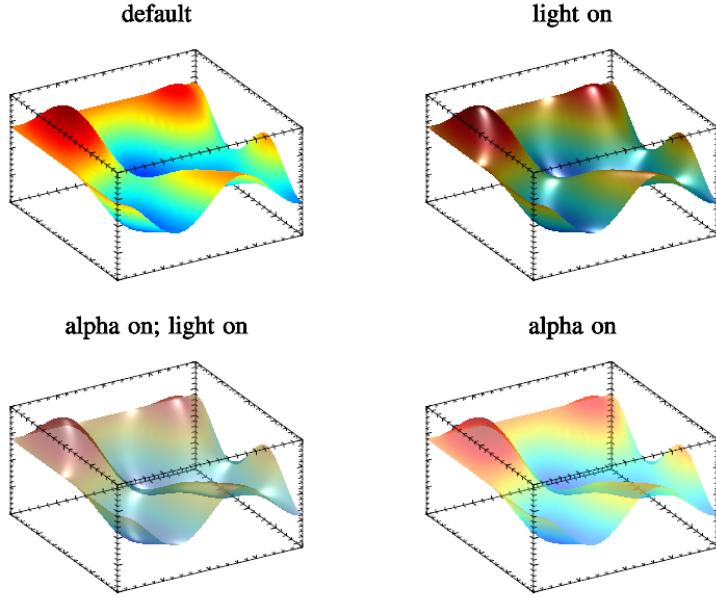
    gr->SubPlot(2,2,3); gr->Title("alpha on"); gr->Rotate(50,60);

```

```

    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()`. 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 113](#)). 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.

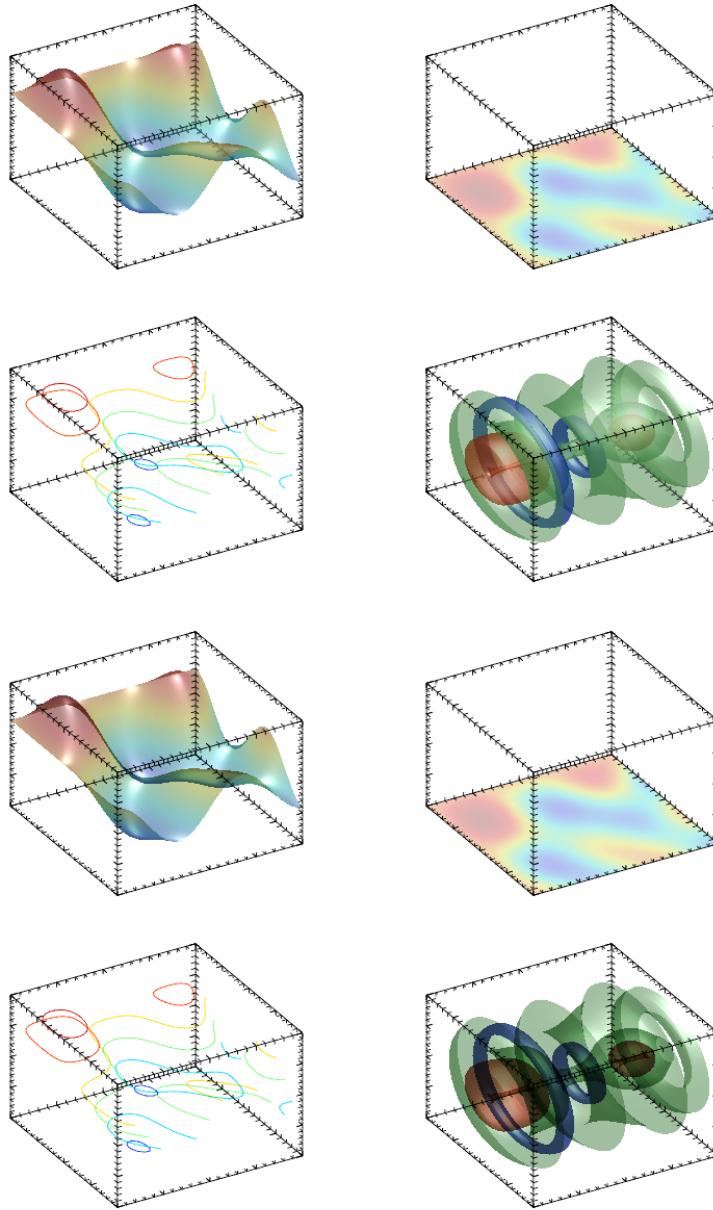
As example I shall show several plots for different types of transparency. The code is the same except the values of `SetTranspType` function:

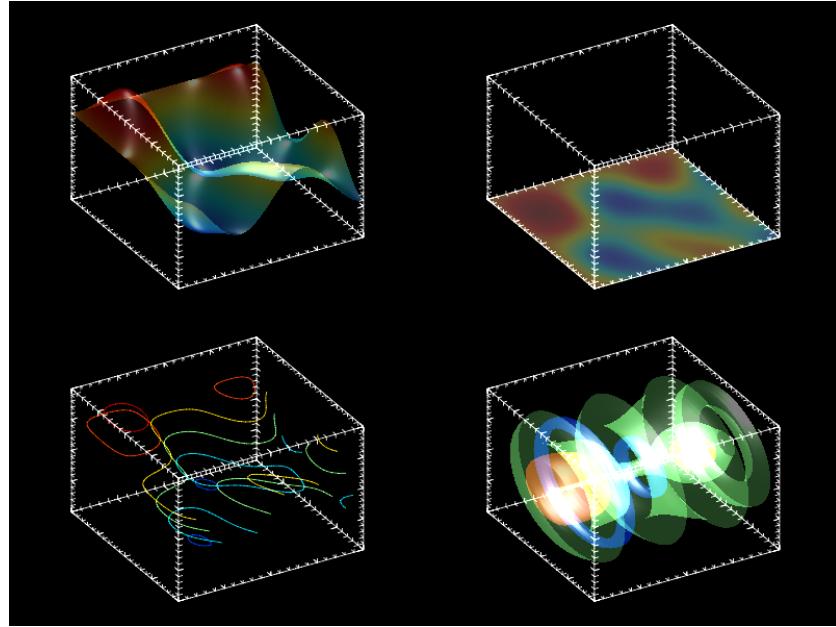
```

int sample(mglGraph *gr)
{
    gr->Alpha(true);  gr->Light(true);

```

```
mglData a; mglS_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;
}
```



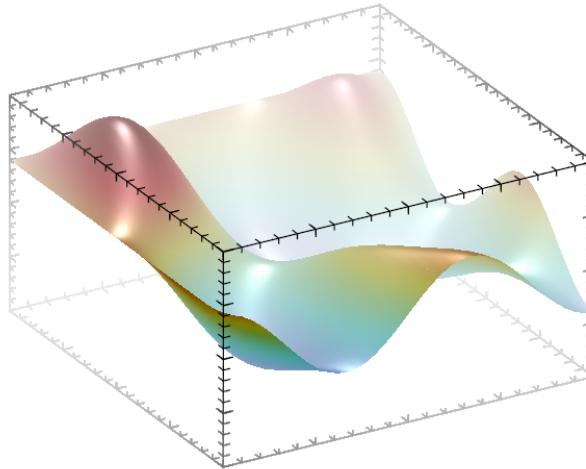


2.9.4 Adding fog

MathGL can add a fog to the image. Its switching on is rather simple – just use `Fog` 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);
    return 0;
}
```

Fog sample

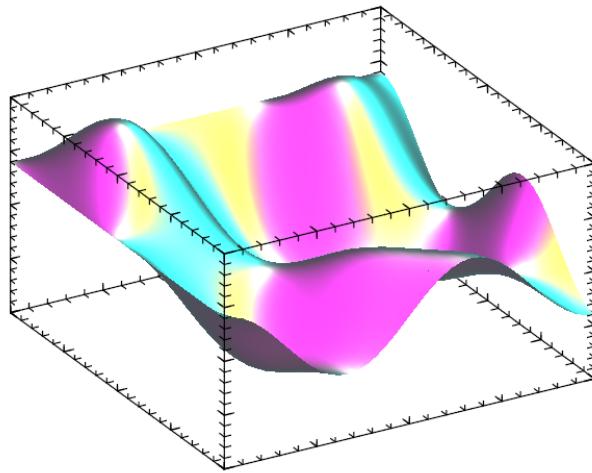


2.9.5 Several light sources

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:

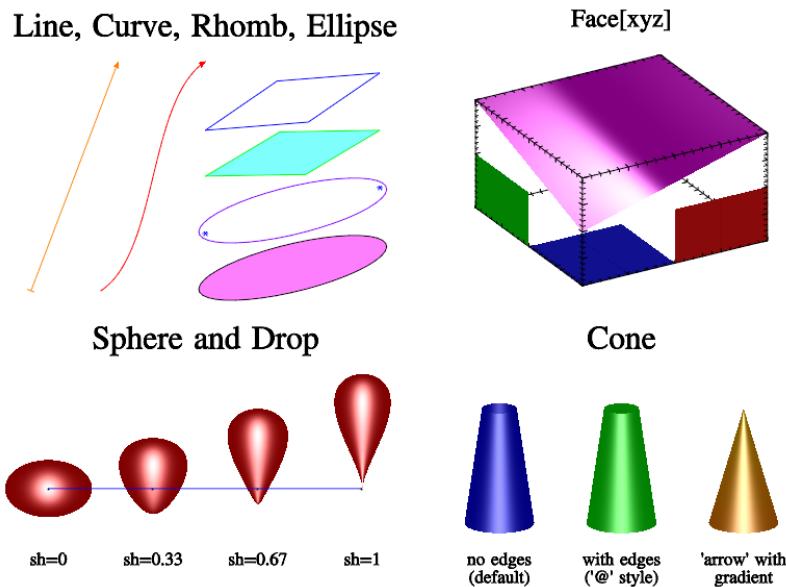
```
int sample(mglGraph *gr)
{
    mglData a;  mglS_prepare2d(&a);
    gr->Title("Several light sources");
    gr->Rotate(50,60);  gr->Light(true);
    gr->AddLight(1,mglPoint(0,1,0), 'c');
    gr->AddLight(2,mglPoint(1,0,0), 'y');
    gr->AddLight(3,mglPoint(0,-1,0), 'm');
    gr->Box();  gr->Surf(a,"h");
    return 0;
}
```

Several light sources



2.9.6 Using primitives

MathGL provide a set of functions for drawing primitives (see [Section 4.6 \[Primitives\], page 136](#)). 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`. The sample code is:

```
int sample(mglGraph *gr)
{
    gr->Alpha(true);  gr->Light(true);
```

```
gr->SubPlot(2,2,0,""); gr->Title("Methane, CH_4");
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, H_{2}O");
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();

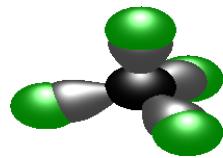
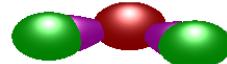
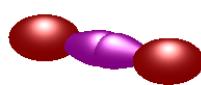
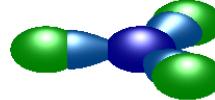
gr->SubPlot(2,2,2,""); gr->Title("Oxygen, O_2");
gr->StartGroup("Oxygen");
gr->Rotate(60,120);
gr->Drop(mglPoint(0,0.5,0),mglPoint(0,-0.3,0),0.3,"m",1,2);
gr->Sphere(mglPoint(0,0.5,0),0.25,"r");
gr->Drop(mglPoint(0,-0.5,0),mglPoint(0,0.3,0),0.3,"m",1,2);
gr->Sphere(mglPoint(0,-0.5,0),0.25,"r");
gr->EndGroup();

gr->SubPlot(2,2,3,""); gr->Title("Ammonia, NH_3");
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, CH₄Water, H₂OOxygen, O₂Ammonia, NH₃

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@#0");
    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** function:

```

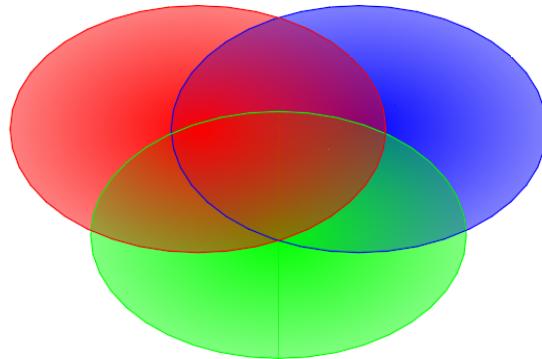
int sample(mglGraph *gr)
{
    gr->Title("Venn-like diagram"); gr->Alpha(true);
    gr->Circle(mglPoint(-0.3,0.3),0.7,"rr@");
    gr->Circle(mglPoint(0,-0.3),0.7,"gg@");
    gr->Circle(mglPoint( 0.3,0.3),0.7,"bb@");
    return 0;
}

```

```
}
```

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



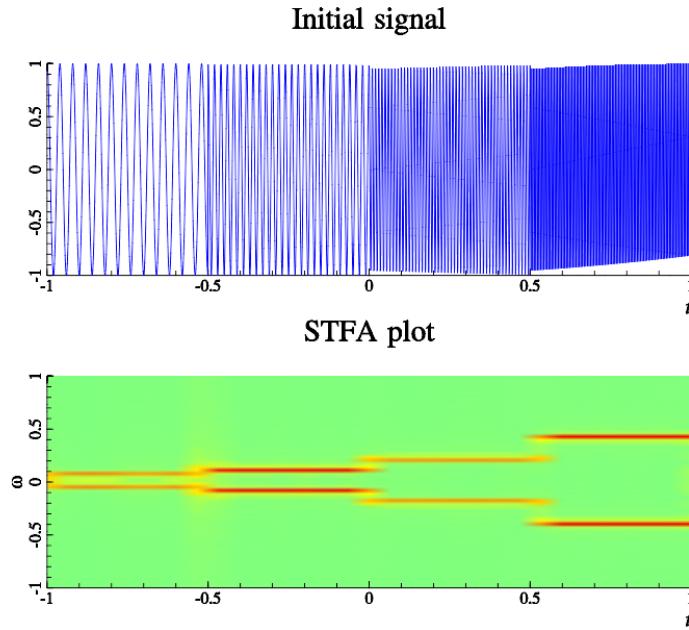
2.9.7 STFA sample

Short-time Fourier Analysis (STFA) 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;
}
```



2.9.8 Mapping visualization

Sometime ago I worked with mapping and have a question about its visualization. Let me remember you that mapping is some transformation rule for one set of number to another one. The 1d mapping is just an ordinary function – it takes a number and transforms it to another one. The 2d mapping (which I used) is a pair of functions which take 2 numbers and transform them to another 2 ones. Except general plots (like `SurfC`, `SurfA`) 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`. 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);

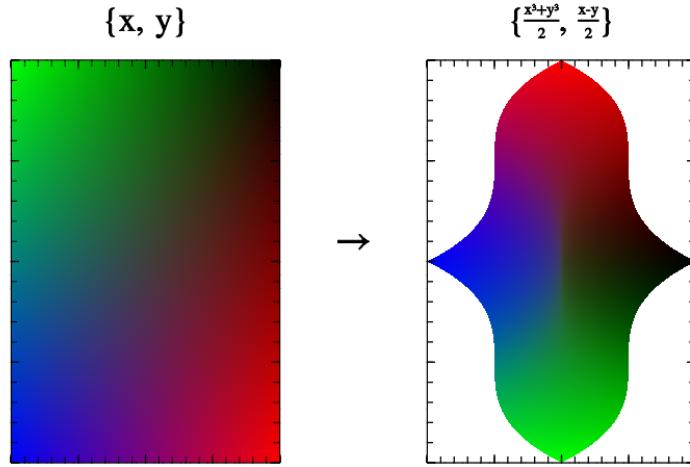
    gr->SubPlot(2, 1, 0);
    gr->Fill(a,"x"); gr->Fill(b,"y");
    gr->Puts(mglPoint(0, 1.1), "\\{x, y\\}", ":C", -2); gr->Box();
    gr->Map(a, b, "brgk");

    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;
}

```



2.9.9 Making histogram

Using the `Hist` 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` usage:

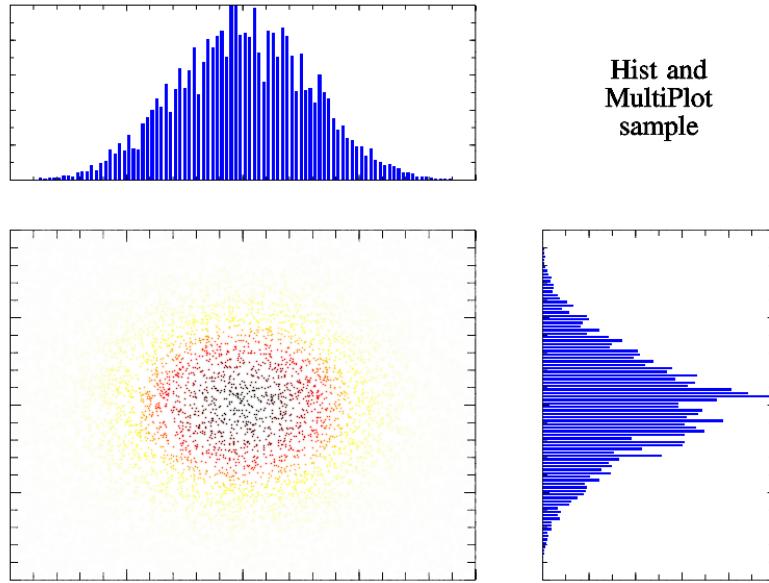
```

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 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 rnd(100), in(100); //data to be fitted and ideal data
gr->Fill(rnd,"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(rnd, ". ");
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)’

```
float ini[3] = {1,1,3};
mglData Ini(3,ini);
mglData res = gr->Fit(rnd, "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:

```

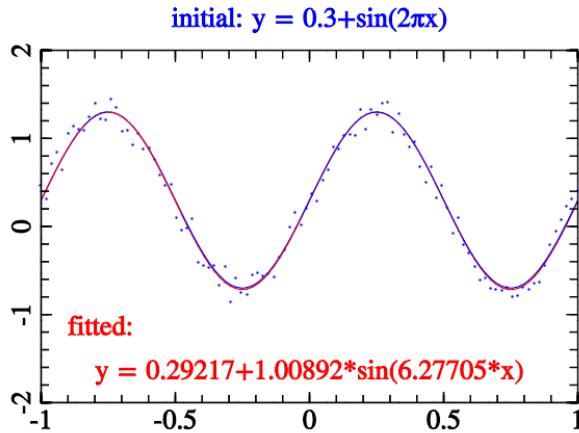
int sample(mglGraph *gr)
{
    mglData rnd(100), in(100);
    gr->Fill(rnd, "0.4*rnd+0.1+sin(2*pi*x)");
    gr->Fill(in, "0.3+sin(2*pi*x)");
    float ini[3] = {1,1,3};
    mglData Ini(3,ini);

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

    gr->Title("Fitting sample");
    gr->SetRange('y',-2,2); gr->Box(); gr->Plot(rnd, ". ");
    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



2.9.11 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 `mglRay()` for ray tracing, `mglPDE()` for PDE solving, `mglQ02d()` for beam tracing in 2D case (see [Section 6.11 \[Global functions\]](#), page [207](#)). 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 `mglRay()` function. Really ray tracing equation is Hamiltonian equation for 3D space. So, the function can be also used for finding a particle trajectory (i.e. solve Hamiltonian ODE) for 1D, 2D or 3D cases. The function have a set of arguments. First of all, it is Hamiltonian which defined the media (or the equation) you are planning to use. The Hamiltonian is defined by string which may depend on coordinates ‘x’, ‘y’, ‘z’, time ‘t’ (for particle dynamics) and momentums ‘p’= p_x , ‘q’= p_y , ‘v’= p_z . Next, you have to define the initial conditions for coordinates and momentums at ‘t’=0 and set the integrations step (default is 0.1) and its duration (default is 10). The Runge-Kutta method of 4-th order is used for integration.

```
const char *ham = "p^2+q^2-x-1+i*0.5*(y+x)*(y>-x)";
mglData r = mglRay(ham, mglPoint(-0.7, -1), mglPoint(0, 0.5), 0.02, 2);
```

This example calculate the reflection from linear layer (media with Hamiltonian ‘ $p^2+q^2-x-1=p_x^2 + p_y^2 - x - 1$ ’). This is parabolic curve. The resulting array have 7 columns which contain data for {x,y,z,p,q,v,t}.

The solution of PDE is a bit more complicated. As previous you have to specify the equation as pseudo-differential operator $\hat{H}(x, \nabla)$ which is called sometime as “Hamiltonian” (for example, in beam tracing). As previously, it is defined by string which may depend on coordinates ‘x’, ‘y’, ‘z’ (but not time!), momentums ‘p’=(d/dx)/ ik_0 , ‘q’=(d/dy)/ ik_0 and field amplitude ‘u’=| u |. The evolutionary coordinate is ‘z’ in all cases. So that, the equation look like $du/dz = ik_0 H(x, y, \hat{p}, \hat{q}, |u|)[u]$. Dependence on field amplitude ‘u’=| u | allows one to solve nonlinear problems too. For example, for nonlinear Shrodinger equation you may set `ham="p^2 + q^2 - u^2"`. Also you may specify imaginary part for wave absorption, like `ham = "p^2 + i*x*(x>0)"`, but only if dependence on variable ‘i’ is linear (i.e. $H = H_{re} + i * H_{im}$).

Next step is specifying the initial conditions at ‘z’=Min.z. The function need 2 arrays for real and for imaginary part. Note, that coordinates x,y,z are supposed to be in specified range [Min, Max]. So, the data arrays should have corresponding scales. Finally, you may set the integration step and parameter k0= k_0 . Also keep in mind, that internally the 2 times large box is used (for suppressing numerical reflection from boundaries) and the equation should well defined even in this extended range.

Final comment is concerning the possible form of pseudo-differential operator H . At this moment, simplified form of operator H is supported – all “mixed” terms (like ‘ $x*p$ ’-> $x*d/dx$) are excluded. For example, in 2D case this operator is effectively $H = f(p, z) + g(x, z, u)$. However commutable combinations (like ‘ $x*q$ ’-> $x*d/dy$) are allowed for 3D case.

So, for example let solve the equation for beam deflected from linear layer and absorbed later. The operator will have the form “`"p^2+q^2-x-1+i*0.5*(z+x)*(z>-x)"`” that correspond to equation $ik_0\partial_z u + \Delta u + x \cdot u + i(x + z)/2 \cdot u = 0$. 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);
```

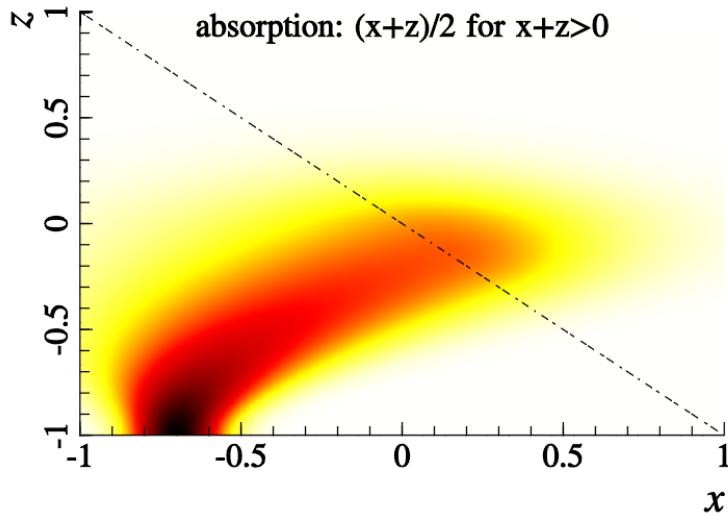
```

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->FPPlot("-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+z}{2}\cdot u = 0");
return 0;
}

```

PDE solver

$$\text{Equation: } ik_0\partial_zu + \Delta u + x\cdot u + i \frac{x+z}{2}\cdot u = 0$$



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 $1/K^2 = (|\ddot{r}|^2|\dot{r}|^2 - (\ddot{r}, \dot{r})^2)/|\dot{r}|^6$) 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
}

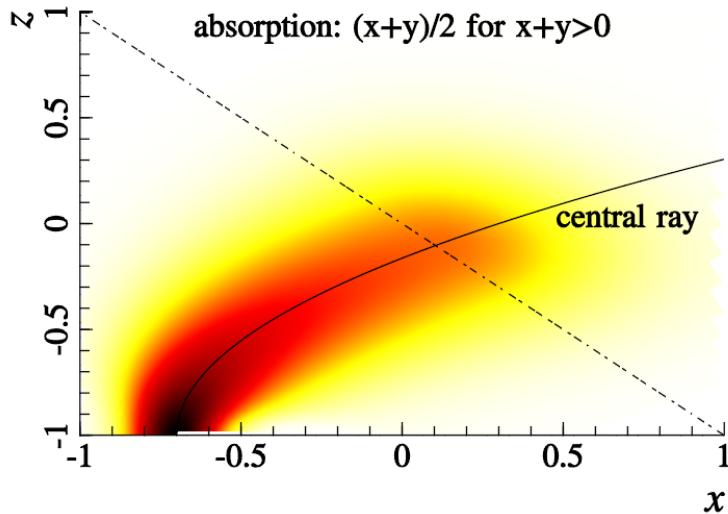
```

```

gr->Fill(re, "exp(-48*x^2)");
a = mglQ02d(ham, re, im, r, xx, yy, 1, 30);
gr->SetRange('c',0, 1);
gr->Dens(xx, yy, a, "wyrRk");
gr->FPplot("-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;
}

```

Beam and ray tracing



2.9.12 MGL parser using

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 7.3 \[mglParse class\]](#), page 214) 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!

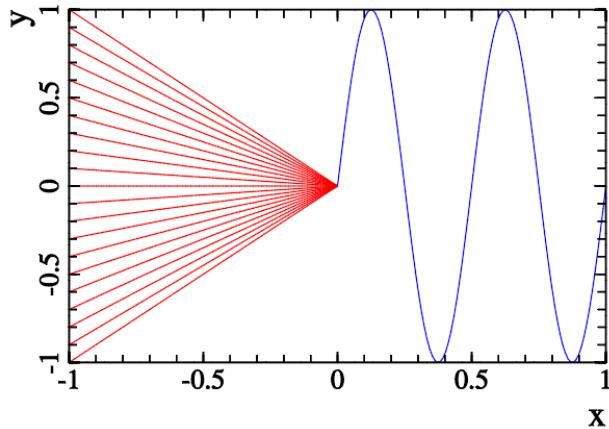
So, some simple example at the end. Here I define a data array, create variable, put data into it and plot it. The C++ code looks like this:

```
int sample(mglGraph *gr)
{
    gr->Title("MGL parser sample");
    float 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\nline 0 0 -1 $0 'r'\nnext");
    delete parser;
    return 0;
}
```

The code in C/Fortran looks practically the same:

```
int sample(HMGL gr)
{
    mgl_title(gr, "MGL parser sample")
    float 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_add_var(parser, "dat");
    mgl_data_set_float(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\nline 0 0 -1 $0 'r'\nnext");
    mgl_write_png(gr, "test.png", ""); // don't forgot to save picture
    return 0;
}
```

MGL parser sample



2.9.13 “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.

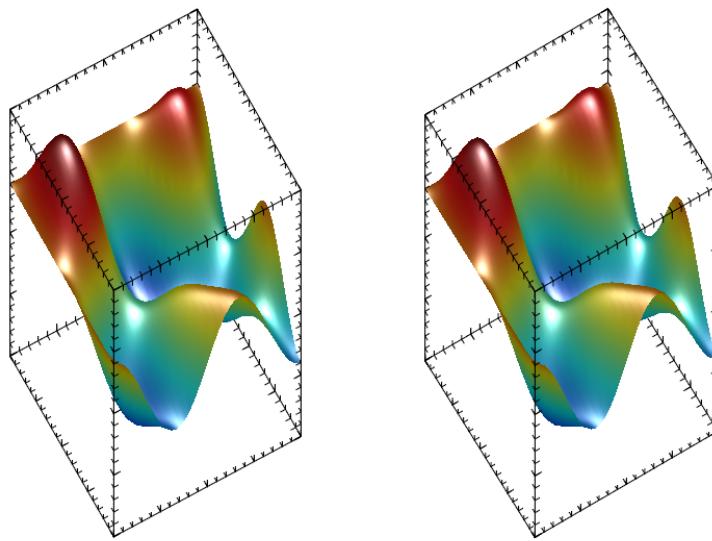
2.9.14 Stereo image

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;
}
```



2.10 FAQ

The plot does not appear

Check that points of the plot lie inside the bounding box and resize the bounding box using `Axis()` 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()`) 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()`, or as a parametrically specified surface by `Surf()`. See also,

Section 2.9 [Hints], page 84. 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 autoconf/automake script. 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);
```

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 will never be plotted. Second, special variables `CutMin`, `CutMax` or function `CutOff()` define the condition when the points should be omitted (see **Section 4.2.5 [Cutting], page 119**). Last, you may change the transparency of a part of the plot by the help of functions `SurfA()`, `Surf3A()` (see **Section 4.13 [Dual plotting], page 164**). In this 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, the recommended class `mglGraph` (header file `#include <mgl2/mgl.h>`) contains only `inline` functions and is acceptable for any compiler with the same binary files. However, if you plan to access low-level features (i.e. classes `mglBase`, `mglCanvas` and so on) then you have to recompile MathGL by yours compiler.

How I can build MathGL under Windows?

The simplest way is using the combination CMake+MinGW. Also you 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 configurator and make 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 autoconf/automake script was written mostly by D.Kulagin, and the export to IDTF was written mostly by M.Vidassov.

How can I display a bitmap on the figure?

You can import data into a `mglData` instance and display it by `Dens()` 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, `Fl_MathGL` 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::GetBits()`.

How can I create U3D file (make 3D in PDF)?

There are 2 steps: first you should create IDTF file, and later convert it to U3D. You can use [U3D tools](#) for converting IDTF file to U3D. It needs [libharu](#) 2.1.0 or later. For installation use `./bootstrap`, `./configure`, `make`, `sudo make install`. It provides `IDTFCConverter` program for converting text files `*.idtf` to binary files `*.u3d`. The latter can be included into PDF.

How I can change the font family?

First, you should download new font files from [here](#) or from [here](#). 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 `SetTickLen`. For example, use `gr->SetTickLen(-0.1);`.

How can I prevent text rotation?

Just use `SetRotatedText(false)`.

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

This standard GNU tool. There is special FAQ about its usage under Windows

– <http://www.mingw.org/wiki/FAQ>.

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 (float, 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 a box *Min x Max* (see [Section 4.3 \[Axis settings\]](#), page 122). 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 119](#)). 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 123](#)). 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 `Min.z`, `Max.z` 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 124](#)). 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 `SetTickTemp1()` for setting templates for tick labels (it supports TeX symbols). Also, there is a possibility to print arbitrary text as tick labels by help of `SetTicksVal()` function.

3.2 Color styles

Base colors are defined by one of symbol ‘`wkrgbcymhRGBCYMHlenupqLENUPQ`’. 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.

3.3 Line styles

The line style is defined by the string which may contain specifications for color (‘`wkrgbcymhRGBCYMHlenupqLENUPQ`’), dashing style (‘`-|;:ji=`’ or space), width (‘`123456789`’) and marks (‘`*o+xsd.^v<>`’ and ‘#’ modifier). If one of the type of information is omitted then default values used with next color from palette (see [Section 4.2.7 \[Palette and colors\], page 120](#)). 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`’.

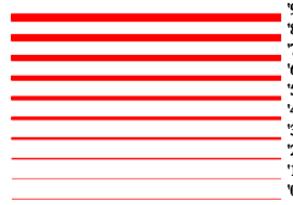
Dashing style has the following meaning: space – no line (usable for plotting only marks), ‘-’ – solid line (# ##### ##### ##### #####), ‘|’ – long dashed line (# ##### ##### -----), ‘;’ – dashed line (# ##### --- ##### ---), ‘=’ – small dashed line (# --- ##### --- ##### ---), ‘:’ – dotted line (# ---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

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, ‘_’ – 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 4.10 \[1D plotting\], page 144](#)).

‘.’	‘o’	‘#.’		Solid ‘.’		Style ‘AA’	Style ‘A’ or ‘A_’
‘+’	‘+’	‘#+’		Long Dash ‘ ’		Style ‘VV’	Style ‘V’ or ‘V_’
‘x’	‘x’	‘#x’		Dash ‘;’		Style ‘KK’	Style ‘K’ or ‘K_’
‘*’	‘*’	‘**’		Small dash ‘=’		Style ‘TT’	Style T or ‘T_’
‘#’	‘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’	‘#o’		Dots ‘?’		Style ‘OO’	Style ‘O’ or ‘O_’
‘w’	‘w’	‘#w’		None ‘’		Style ‘TT’	Style ‘T’ or ‘T_’
‘v’	‘v’	‘#v’				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



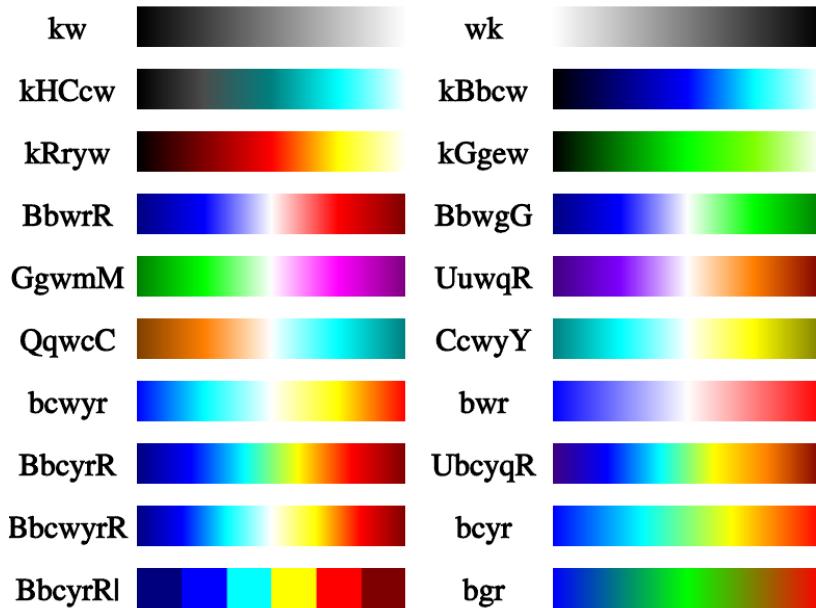
3.4 Color scheme

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 108](#)) 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 108](#)). The argument is the amplitude normalized between $C_{min} - C_{max}$ (see [Section 4.3 \[Axis settings\], page 122](#)). 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.

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.



When coloring by *coordinate* (used in [\[map\]](#), page 166), 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.

3.5 Font styles

Text style is specified by the string which may contain: color id characters ‘wkrgbcymhRGBCYMHW’ (see [Section 3.2 \[Color styles\], page 108](#)), 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’).

The font styles are: ‘**r**’ – roman (or regular) font, ‘**i**’ – italic style, ‘**b**’ – bold style. By default roman roman font is used. The align types are: ‘**L**’ – align left (default), ‘**C**’ – align center, ‘**R**’ – align right. Additional font effects are: ‘**w**’ – wired, ‘**o**’ – over-lined, ‘**u**’ – underlined.

Also a parsing of the LaTeX-like syntax is provided. There are commands for the font style changing inside the string (for example, use `\b` for bold font): `\a` or `\overline` – over-lined, `\b` or `\textbf` – bold, `\i` or `\textit` – italic, `\r` or `\textrm` – roman (disable bold and italic attributes), `\u` or `\underline` – underlined, `\w` or `\wire` – wired, `\big` – bigger size, `\smaller` – 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 108](#)). For example, words ‘`blue`’ and ‘`red`’ will be colored in the string ‘`#b{blue}` and `\color{red} text`’. The most of functions understand the newline symbol ‘`\n`’ and allows to print multi-line text. Finally, you can use arbitrary (if it was defined in font-face) UTF codes by command `\utf0x????`. For example, `\utf0x3b1` will produce α symbol.

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

In particular, the Greek letters are recognizable special symbols: α – `\alpha`, β – `\beta`, γ – `\gamma`, δ – `\delta`, ϵ – `\epsilon`, η – `\eta`, ι – `\iota`, χ – `\chi`, κ – `\kappa`, λ – `\lambda`, μ – `\mu`, ν – `\nu`, ω – `\omega`, ϕ – `\phi`, π – `\pi`, ψ – `\psi`, ρ – `\rho`, σ – `\sigma`, θ – `\theta`, τ – `\tau`, v – `\upsilon`, ξ – `\xi`, ζ – `\zeta`, ς – `\varsigma`, ε – `\varepsilon`, ϑ – `\vartheta`, φ – `\varphi`, A – `\Alpha`, B – `\Beta`, Γ – `\Gamma`, Δ – `\Delta`, E – `\Epsilon`, H – `\Eta`, I – `\Iota`, C – `\Chi`, K – `\Kappa`, Λ – `\Lambda`, M – `\Mu`, N – `\Nu`, O – `\O`, Ω – `\Omega`, Φ – `\Phi`, Π – `\Pi`, Ψ – `\Psi`, R – `\Rho`, Σ – `\Sigma`, Θ – `\Theta`, T – `\Tau, Υ – \Upsilon, Ξ – \Xi, Z – \Zeta.`

The small part of most common special TeX symbols are: \angle – `\angle`, \aleph – `\aleph`, \cdot – `\cdot`, \clubsuit – `\clubsuit`, \cup – `\cup`, \cap – `\cap`, \diamondsuit – `\diamondsuit`, \diamond – `\diamond`, \div – `\div`, \downarrow – `\downarrow`, \downarrow – `\downarrow`, \dag – `\dag`, \ddag – `\ddag`, \equiv – `\equiv`, \exists – `\exists`, \nexists – `\nexists`, \frown – `\frown`, \flat – `\flat`, \geq – `\geq`, \geq – `\geq`, \leq – `\leq`, \leq – `\leq`, \leftarrow – `\leftarrow`, \leftarrow – `\leftarrow`, \leftarrow – `\leftarrow`, \mp – `\mp`, ∇ – `\nabla`, \neq – `\neq`, \neq – `\neq`, \natural – `\natural`, \oint – `\oint`, \odot – `\odot`, \oplus – `\oplus`, ∂ – `\partial`, \parallel – `\parallel`, \perp – `\perp`, \pm – `\pm`, \propto – `\propto`, \prod – `\prod`, \Re – `\Re`, \rightarrow – `\rightarrow`, \rangle – `\rangle`, \spadesuit – `\spadesuit`, \sim – `\sim`, \smile – `\smile`, \subset – `\subset`, \supset – `\supset`, \sqrt – `\sqrt` or `\sqrt`, \S – `\S`, \sharp – `\sharp`, \sum – `\sum`, \times – `\times`, \rightarrow – `\rightarrow`, \uparrow – `\uparrow`, \wp – `\wp` and so on.

The font size can be defined explicitly (if `size>0`) or relatively to a base font size as `|size|*FontSize` (if `size<0`). The value `size=0` specifies that the string will not be printed. The base font size is measured in internal “MathGL” units. Special functions

`SetFontSizePT()`, `SetFontSizeCM()`, `SetFontSizeIN()` (see [Section 4.2.6 \[Font settings\]](#), page 119) 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 209). 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, ‘`pi`’ – number $\pi = 3.1415926\dots$

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 $\Gamma(x) = \int_0^\infty dt t^{x-1} \exp(-t)$, ‘`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(l,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.

The full list of options are:

<code>alpha val</code>	[MGL option]
Sets alpha value (transparency) of the plot. The value should be in range [0, 1]. See also [alphadef] , page 117.	
<code>ambient val</code>	[MGL option]
Sets brightness of ambient light for the plot. The value should be in range [0, 1]. See also [ambient] , page 118.	
<code>diffuse val</code>	[MGL option]
Sets brightness of diffuse light for the plot. The value should be in range [0, 1]. See also [diffuse] , page 118.	
<code>xrange val1 val2</code>	[MGL option]
Sets boundaries of x coordinate change for the plot. See also [xrange] , page 122.	
<code>yrange val1 val2</code>	[MGL option]
Sets boundaries of y coordinate change for the plot. See also [yrange] , page 122.	
<code>zrange val1 val2</code>	[MGL option]
Sets boundaries of z coordinate change for the plot. See also [zrange] , page 122.	
<code>cut val</code>	[MGL option]
Sets whether to cut or to project the plot points lying outside the bounding box. See also [cut] , page 119.	
<code>size val</code>	[MGL option]
Sets the size of text, marks and arrows. See also [font] , page 119, [marksize] , page 119, [arrowsize] , page 119.	
<code>meshnum val</code>	[MGL option]
Work like [meshnum] , page 119 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 4.10 \[1D plotting\], page 144](#) command.
See also [\[legend\], page 143](#).

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 116](#) and [Chapter 6 \[Data processing\], page 191](#)) and marked like [Method on `mglGraph`].

Finally, a special command language MGL (see [Chapter 7 \[MGL scripts\], page 211](#)) 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 7.3 \[mglParse class\], page 214](#)).

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 116](#)).
- `HMDT` — Pointer to class `mglData` (see [Chapter 6 \[Data processing\], page 191](#)).
- `HMPR` — Pointer to class `mglParse` (see [Section 7.3 \[mglParse class\], page 214](#)).

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 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 classes in:

- `mglGraph` – provide most plotting functions (see [Chapter 4 \[MathGL core\]](#), page 116).
- `mglData` – provide base data processing (see [Chapter 6 \[Data processing\]](#), page 191). 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 211).

To use Python classes just execute ‘`import mathgl`’. The simplest example will be:

```
import mathgl
a=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 191](#)) 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 107](#) and [Chapter 8 \[Other classes\], page 216](#).

4.1 Create and delete objects

<code>mglGraph (int kind=0, int width=600, int height=400)</code>	[Constructor on mglGraph]
<code>mglGraph (const mglGraph &gr)</code>	[Constructor on mglGraph]
<code>mglGraph (HMGL gr)</code>	[Constructor on mglGraph]
<code>HMGL mgl_create_graph (int width, int height)</code>	[C function]
<code>HMGL mgl_create_graph_gl ()</code>	[C function]
Creates the instance of class mglGraph with specified sizes <i>width</i> and <i>height</i> . Parameter <i>kind</i> may have following values: ‘0’ – use default plotter, ‘1’ – use OpenGL plotter.	
<code>~mglGraph ()</code>	[Destructor on mglGraph]
<code>HMGL mgl_delete_graph (HMGL gr)</code>	[C function]
Deletes the instance of class mglGraph .	
<code>HMGL Self ()</code>	[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.

<code>void DefaultPlotParam ()</code>	[Method on mglGraph]
<code>void mgl_set_def_param (HMGL gr)</code>	[C function]
Restore initial values for all of parameters.	
<code>void SetPlotId (const char *id)</code>	[Method on mglGraph]
<code>void mgl_set_plotid (HMGL gr, const char *id)</code>	[C function]
Sets name of plot for saving filename (in FLTK window for example).	

4.2.1 Transparency

There are several functions and variables for setup transparency. The general function is [\[alpha\], page 117](#) which switch on/off the transparency for overall plot. It influence

only for graphics which created after [alpha], page 117 call (with one exception, OpenGL). Function [alphadef], page 117 specify the default value of alpha-channel. Finally, function [transptype], page 117 set the kind of transparency. See Section 2.9.2 [Transparency and lighting], page 85, for sample code and picture.

<code>alpha [val=on]</code>	[MGL command]
<code>void Alpha (bool enable)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_alpha (HMGL gr, int enable)</code>	[C function]

Sets the transparency on/off and returns previous value of transparency. It is recommended to call this function before any plotting command. Default value is transparency off.

<code>alphadef val</code>	[MGL command]
<code>void SetAlphaDef (float val)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_alpha_default (HMGL gr, float alpha)</code>	[C function]

Sets default value of alpha channel (transparency) for all plotting functions. Initial value is 0.5.

<code>transptype val</code>	[MGL command]
<code>void SetTranspType (int type)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_transp_type (HMGL gr, int type)</code>	[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 86, for sample code and picture..

4.2.2 Lighting

There are several functions for setup lighting. The general function is `Light(bool)` which switch on/off the lighting for overall plot. It influence only for graphics which created after [light], page 117 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.

<code>light [val=on]</code>	[MGL command]
<code>bool Light (bool enable)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_light (HMGL gr, int enable)</code>	[C function]

Sets the using of light on/off for overall plot. Function returns previous value of lighting. Default value is lightning off.

<code>light num val</code>	[MGL command]
<code>void Light (int n, bool enable)</code>	[Method on <code>mglGraph</code>]

```

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', float [Method on mglGraph]
    bright=0.5, float ap=0)
void AddLight (int n, mglPoint r, mglPoint d, char [Method on mglGraph]
    c='w', float bright=0.5, float ap=0)
void mgl_add_light (HMGL gr, int n, float dx, float dy, float dz) [C function]
void mgl_add_light_ext (HMGL gr, int n, float dx, float dy, [C function]
    float dz, char c, float bright, float ap)
void mgl_add_light_loc (HMGL gr, int n, float rx, float ry, [C function]
    float rz, float dx, float dy, float dz, char c, float bright, float ap)
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.

void SetDifLight (bool enable) [ mglGraph]
void mgl_set_light_dif (HMGL gr, int enable) [ ]
Set on/off to use diffusive light (only for local light sources).

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

```

4.2.3 Fog

```

fog val [dz=0.25] [MGL command]
void Fog (float d, float dz=0.25) [Method on mglGraph]
void mgl_set_fog (HMGL gr, float d, float 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  $\sim 1 - \exp(-d^*z)$ . 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 2.9.4 \[Adding fog\], page 88, 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 ( float val) [Method on mglGraph]
void mgl_set_bar_width (HMGL gr, float val) [C function]
Sets relative width of rectangles in \[bars\], page 148, \[barh\], page 148, \[boxplot\], page 149, \[candle\], page 150. Default value is 0.7.

```

```

marksize val                                [MGL command]
void SetMarkSize (float val)                [Method on mglGraph]
void mgl_set_mark_size (HMGL gr, float val)      [C function]
    Sets size of marks for Section 4.10 \[1D plotting\], page 144. Default value is 0.02.

arrowsize val                                [MGL command]
void SetArrowSize (float val)                [Method on mglGraph]
void mgl_set_arrow_size (HMGL gr, float val)      [C function]
    Sets size of arrows for Section 4.10 \[1D plotting\], page 144, lines and curves (see
Section 4.6 \[Primitives\], page 136). Default value is 0.03.

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 154, \[fall\], page 155, \[grid\], page 142
    and also the number of hachures in \[vect\], page 168, \[dew\], page 168 and the number
    of cells in \[cloud\], page 161. By default (=0) it draws all lines/hachures/cells.

```

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 28, 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, float x1, float y1, float z1,           [C function]
                      float x2, float y2, float 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 110.

<code>rotatetext val</code>	[MGL command]
<code>void SetRotatedText (bool val)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_rotated_text (HMGL gr, int val)</code>	[C function]
Sets to use or not text rotation.	
<code>void SetFontDef (const char *fnt)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_font_def (HMGL gr, const char * val)</code>	[C function]
Sets the font specification (see Section 4.7 [Text printing] , page 139). Default is ‘rC’ – Roman font centering.	
<code>void SetFontSize (float val)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_font_size (HMGL gr, float val)</code>	[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.	
<code>void SetFontSizePT (float cm, int dpi=72)</code>	[Method on <code>mglGraph</code>]
Set FontSize by size in pt and picture DPI (default is 16 pt for dpi=72).	
<code>inline void SetFontSizeCM (float cm, int dpi=72)</code>	[Method on <code>mglGraph</code>]
Set FontSize by size in centimeters and picture DPI (default is 0.56 cm = 16 pt).	
<code>inline void SetFontSizeIN (float cm, int dpi=72)</code>	[Method on <code>mglGraph</code>]
Set FontSize by size in inch and picture DPI (default is 0.22 in = 16 pt).	
<code>void LoadFont (const char *name, const char *path="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_load_font (HMGL gr, const char *name, const char *path)</code>	[C function]
Load font typeface from <i>path/name</i> .	
<code>void CopyFont (mglGraph * from)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_copy_font (HMGL gr, HMGL gr_from)</code>	[C function]
Copy font data from another <code>mglGraph</code> object.	
<code>void RestoreFont ()</code>	[Method on <code>mglGraph</code>]
<code>void mgl_restore_font (HMGL gr)</code>	[C function]
Restore font data to default typeface.	

4.2.7 Palette and colors

<code>palette 'colors'</code>	[MGL command]
<code>void SetPalette (const char *colors)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_palette (HMGL gr, const char *colors)</code>	[C function]
Sets the palette as selected colors. Default value is "Hbgrcmhyhlnqeup" 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 4.10 [1D plotting] ,	

[page 144](#)) 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()` or `Axis()` functions).

<code>void SetDefScheme (const char *sch)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_def_sch (HMGL gr, const char *sch)</code>	[C function]
Sets the <code>sch</code> as default color scheme. Default value is "BbcyrR".	
<code>void mgl_set_color (char id, float r, float g, float b)</code>	[C function]
Sets RGB values for color with given <code>id</code> .	

4.2.8 Error handling

Normally user should set it to zero by `SetWarn(0)`; before plotting and check if `GetWarnCode()` 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.

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

<code>const char *Message ()</code>	[Method on <code>mglGraph</code>]
<code>const char *mgl_get_mess (HMGL gr)</code>	[C function]
Return messages about matters why some plot are not drawn. If returned string is empty then there are no messages.	

<code>int GetWarnCode ()</code>	[Method on <code>mglGraph</code>]
<code>int mgl_get_warn_code (HMGL gr)</code>	[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

```

mglWarnLegA
    Too many legend entries

mglWarnLeg
    No legend entries

mglWarnS1c
    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

```

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 119), after it transformation formulas are applied, and finally the data was normalized in bounding box.

4.3.1 Ranges (bounding box)

<code>xrange v1 v2</code>	[MGL command]
<code>yrange v1 v2</code>	[MGL command]
<code>zrange v1 v2</code>	[MGL command]
<code>crange v1 v2</code>	[MGL command]
<code>void SetRange (char dir, float v1, float v2)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_range_val (HMGL gr, char dir, float v1, float v2)</code>	[C function]
Sets the range for ‘x’-,‘y’-,‘z’- coordinate or coloring (‘c’). See also [ranges] , page 123.	
<code>xrange dat [add=off]</code>	[MGL command]
<code>yrange dat [add=off]</code>	[MGL command]
<code>zrange dat [add=off]</code>	[MGL command]
<code>crange dat [add=off]</code>	[MGL command]
<code>void SetRange (char dir, const mglData &dat, bool add=false)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_range_dat (HMGL gr, char dir, const HCDT a, int add)</code>	[C function]
Sets the range for ‘x’-,‘y’-,‘z’- coordinate or coloring (‘c’) as minimal and maximal values of data <code>dat</code> . Parameter <code>add=on</code> 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 (float x1, float x2, float y1, float y2,      [Method on mglGraph]
                float z1=0, float z2=0)
void mgl_set_ranges (HMGL gr, float x1, float x2, float y1,      [C function]
                     float y2, float z1, float 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 mglData &xx, const mglData &yy)          [Method on mglGraph]
void SetRanges (const mglData &xx, const mglData &yy, const mglData &zz) [Method on mglGraph]
void SetRanges (const mglData &xx, const mglData &yy, const mglData &zz, const mglData &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.
```

```

origin x0 y0 [z0=nan]                                         [MGL command]
void SetOrigin (mglPoint p0)                                     [Method on mglGraph]
void SetOrigin (float x0, float y0, float z0=NAN)             [Method on mglGraph]
void mgl_set_origin (HMGL gr, float x0, float y0, float z0)   [C function]
Sets center of axis cross section. If one of values is NAN then MathGL try to select
optimal axis position.

```

4.3.2 Curved coordinates

```

axis 'fx' 'fy' ['fz'=' 'fa'='']                               [MGL command]
void SetFunc (const char *EqX, const char *EqY,               [Method on mglGraph]
              const char *EqZ="", const char *EqA="")
void mgl_set_func (HMGL gr, const char *EqX, const char *EqY,    [C function]
                   const char *EqZ, const char *EqA)
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 corre-
sponding 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 112.

```

```

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 co-
ordinate. Paramater how define the coordinates: mglCartesian=0 –
Cartesian coordinates (no transformation); mglPolar=1 – Polar coordinates
 $x_n = x * \cos(y), y_n = x * \sin(y), z_n = z;$  mglSpherical=2 – Sperical coor-
dinates  $x_n = x * \sin(y) * \cos(z), y_n = x * \sin(y) * \sin(z), z_n = x * \cos(y);$ 

```

```

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  $x_n = x, y_n = \lg(y), z_n = z.$ 

```

<code>ternary val</code>	[MGL command]
<code>void Ternary (int tern)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_ternary (HMGL gr, int tern)</code>	[C function]

The function sets to draws Ternary or Quaternary plot. 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 $a+b+c+d=1$. Use `Ternary(0)` for returning to usual axis. See Section 2.2.6 [Ternary axis], page 24, for sample code and picture.

4.3.3 Ticks

<code>adjust ['dir'='xyzc']</code>	[MGL command]
<code>void Adjust (const char *dir="xyzc")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_adjust_ticks (HMGL gr, const char *dir)</code>	[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.

<code>xtick val [sub=0 org=nan]</code>	[MGL command]
<code>ytick val [sub=0 org=nan]</code>	[MGL command]
<code>ztick val [sub=0 org=nan]</code>	[MGL command]
<code>ctick val [sub=0 org=nan]</code>	[MGL command]
<code>void SetTicks (char dir, float d=0, int ns=0, float org=NAN)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_ticks_dir (HMGL gr, char dir, float d, int ns, float org)</code>	[C function]
<code>void mgl_set_ticks (HMGL gr, int nx, int ny, int nz)</code>	[C function]

Set the ticks step `d`, number of sub-ticks `ns` 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.

<code>xtick val1 'lbl1' [val2 'lbl2' ...]</code>	[MGL command]
<code>ytick val1 'lbl1' [val2 'lbl2' ...]</code>	[MGL command]
<code>ztick val1 'lbl1' [val2 'lbl2' ...]</code>	[MGL command]
<code>void SetTicksVal (char dir, const char *lbl, bool add=false)</code>	[Method on <code>mglGraph</code>]
<code>void SetTicksVal (char dir, const wchar_t *lbl, bool add=false)</code>	[Method on <code>mglGraph</code>]
<code>void SetTicksVal (char dir, const mglData &val, const char *lbl, bool add=false)</code>	[Method on <code>mglGraph</code>]
<code>void SetTicksVal (char dir, const mglData &val, const wchar_t *lbl, bool add=false)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_ticks_str (HMGL gr, char dir, const char *lbl, bool add)</code>	[C function]
<code>void mgl_set_ticks_wcs (HMGL gr, char dir, const wchar_t *lbl, bool add)</code>	[C function]
<code>void mgl_set_ticks_val (HMGL gr, char dir, HCDT val, const char *lbl, bool add)</code>	[C function]
<code>void mgl_set_ticks_valw (HMGL gr, char dir, HCDT val, const wchar_t *lbl, bool add)</code>	[C function]

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

<code>xtick 'templ'</code>	[MGL command]
<code>ytick 'templ'</code>	[MGL command]
<code>ztick 'templ'</code>	[MGL command]
<code>ctick 'templ'</code>	[MGL command]
<code>void SetTickTempl (char dir, const char *templ)</code>	[Method on <code>mglGraph</code>]
<code>void SetTickTempl (char dir, const wchar_t *templ)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_tick_templ (HMGL gr, const char *templ)</code>	[C function]
<code>void mgl_set_tick_templw (HMGL gr, const wchar_t *templ)</code>	[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'). Setting on template switch off automatic ticks tuning.

<code>ticktime 'dir' [dv 'tmpl']</code>	[MGL command]
<code>void SetTicksTime (char dir, float val, const char *tmpl)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_set_ticks_time (HMGL gr, float val, const char *tmpl)</code>	[C function]

Set time labels with step *val* and template *tmpl* for x-,y-,z-axis ticks or colorbar ticks. It may contain TeX symbols also. The format of template *tmpl* 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 *tmpl=""* then automatic tick step and/or template will be selected.

```
tuneticks val [pos=1.15] [MGL command]
void SetTuneTicks (int tune, float pos=1.15) [Method on mglGraph]
void mgl_tune_ticks (HMGL gr, int tune, float 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.

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.

ticklen val [stt=1] [MGL command]
void SetTickLen (float val, float stt=1) [Method on mglGraph]
void mgl_set_tick_len (HMGL gr, float val, float 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 sub 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 126, [\[multiplot\]](#), page 127 or [\[inplot\]](#), page 127 for specifying the place. Second one can be [\[title\]](#), page 128 for adding title for the subplot. After it a [\[rotate\]](#), page 128 and [\[aspect\]](#), page 128. And finally any other plotting functions may be called. Alternatively you can use [\[columnplot\]](#), page 127, [\[gridplot\]](#), page 128, [\[stickplot\]](#), page 128 or relative [\[inplot\]](#), page 127 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 15, 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 *stl="<>_^", float dx=0, float dy=0) [Method on mglGraph]
void mgl_subplot (HMGL gr, int nx, int ny, int m, const char *stl) [C function]
```

```
void mgl_subplot_d (HMGL gr, int nx, int ny, int m, const char      [C function]
                    *stl, float dx, float 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.

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*.

```
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=off]                                     [MGL command]
```

```
void InPlot (float x1, float x2, float y1, float y2,           [Method on mglGraph]
             bool rel=false)
```

```
void mgl_inplot (HMGL gr, float x1, float x2, float y1, float y2)  [C function]
```

```
void mgl_relplot (HMGL gr, float x1, float x2, float y1, float      [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 126](#) (or [\[inplot\]](#), [page 127](#) 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, float 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, float d)     [C function]
```

Puts further plotting in *ind*-th cell of column with *num* cells. The position is relative to previous [\[subplot\]](#), [page 126](#) (or [\[inplot\]](#), [page 127](#) 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, float 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, float d) [C function]
```

Puts further plotting in *ind*-th cell of *nx*ny* grid. The position is relative to previous [[subplot](#), page 126 (or [[inplot](#)], page 127 with *rel=false*). Parameter *d* set extra gap between cells.

```
stickplot num ind tet phi                               [MGL command]
void StickPlot (int num, int ind, float tet, float phi) [Method on mglGraph]
void mgl_stickplot (HMGL gr, int num, int ind, float tet, float phi) [C function]
```

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 126 (or [[inplot](#)], page 127 with *rel=false*).

```
title 'title' ['stl'=' size=-2]                         [MGL command]
void Title (const char *txt, const char *stl="", float size=-2) [Method on mglGraph]
```

```
void Title (const wchar_t *txt, const char *stl="", float size=-2) [Method on mglGraph]
```

```
void mgl_title (HMGL gr, const char *txt, const char *stl, float size) [C function]
```

```
void mgl_titlew (HMGL gr, const wchar_t *txt, const char *stl, float size) [C function]
```

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

- font style (see, [Section 3.5 \[Font styles\]](#), page 110);
- ‘#’ 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 tetz tetx [tety=0]                                [MGL command]
void Rotate (float TetX, float TetZ, float TetY=0)      [Method on mglGraph]
void mgl_rotate (HMGL gr, float TetX, float TetZ, float 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 (float Tet, float x, float y, float z)     [Method on mglGraph]
void mgl_rotate_vector (HMGL gr, float Tet, float x, float y, float z) [C function]
```

Rotates a further plotting around vector {x, y, z} on angle *Tet*.

```
aspect ax ay [az=1]                                     [MGL command]
void Aspect (float Ax, float Ay, float Az=1)           [Method on mglGraph]
void mgl_aspect (HMGL gr, float Ax, float Ay, float 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 128 function.

```

perspective val                                [MGL command]
void Perspective (float a)                  [Method on mglGraph]
void mgl_perspective (HMGL gr, float a)      [C function]
    Add (switch on) the perspective to plot. The parameter  $a / z_{eff} \in [0, 1]$ . By default ( $a=0$ ) the perspective is off.

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. Stack can keep up to 10 matrices.

void Pop ()                                 [Method on mglGraph]
void mgl_mat_pop (HMGL gr)                 [C function]
    Pop (restore last 'pushed') transformation matrix into stack.

void SetPlotFactor (float val)              [Method on mglGraph]
void mgl_set_plotfactor (HMGL gr, float val) [C function]
    Sets the factor of plot size. It is not recommended to set it lower than 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 2 functions `View()` and `Zoom()` which transform whole image. I.e. they act as secondary transformation matrix. They were introduced for rotating/zooming the whole plot by mouse.

```

void View (float TetX, float TetZ, float TetY=0)      [Method on mglGraph]
void mgl_view (HMGL gr, float TetX, float TetZ, float 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 128. Use Zoom(0,0,1,1) to return default view.

void Zoom (float x1, float y1, float x2,      [Method on mglGraph (C++, Python)]
           float y2)
void mgl_set_zoom (HMGL gr, float x1, float y1, float x2, float      [C function]
                   y2)

```

The function changes the scale of graphics that correspond to zoom in/out of the picture. After function call the current plot will be cleared and further the picture will contain plotting from its part $[x1, x2] * [y1, y2]$. Here picture coordinates $x1, x2, y1, y2$ changes from 0 to 1. Attention! this settings can not be overwritten by any other functions. 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)        [Method on mglGraph]
void mgl_set_size (HMGL gr, int width, int height) [C function]
    Sets size of picture in pixels. This function must be called before any other plotting because it completely remove picture contents.

```

```
quality val [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). If MGL_DRAW_LMEM=0x4 is set then direct bitmap drawing is used
(low memory usage).
```

```
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 IDTF files.

```
write ['fname='"]
[MGL command]
void WriteFrame (const char *fname="", const char *descr="") [Method on mglGraph]
void mgl_write_frame (HMGL gr, const char *fname, const char *descr) [C function]
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 class property.
```

```
void WritePNG (const char *fname, const char *descr="", int compr="", bool alpha=true) [Method on mglGraph]
void mgl_write_png (HMGL gr, const char *fname, const char *descr) [C function]
void mgl_write_png_solid (HMGL gr, const char *fname, const char *descr) [C function]
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
NO_PNG is 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 NO_JPEG is 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.

```
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.

```
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 165](#), [\[surf3a\]](#), [page 165](#) or [\[cloud\]](#), [page 161](#)). 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.

```
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).

```
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](#) 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](#) 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 *descr="") [Method on mglGraph]
```

```
void mgl_write_stl (HMGL gr, const char *fname, const char *descr) [C function]
```

Exports current frame to STL file using vector representation (see [STL 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 *descr="", bool colored=false) [Method on mglGraph]
```

```
void mgl_write_off (HMGL gr, const char *fname, const char *descr, bool colored) [C function]
```

Exports current frame to OFF file using vector representation (see [OFF 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 WriteIDTF (const char *fname, const char *descr="") [Method on mglGraph]
```

```
void mgl_write_idtf (HMGL gr, const char *fname, const char *descr) [C function]
```

Exports current frame to IDTF file. Later this file can be converted to U3D format. The vector representation is used. So, the output file may be too large for graphic of large data array (especially for surfaces). 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 nowait=false) [Method on mglGraph]
```

```
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 ExportMGLD (const char *fname, const char *descr="") [Method on mglGraph]
```

```
void mgl_export_mgld (HMGL gr, const char *fname, const char *descr) [C function]
```

Exports points and primitives in file using MGLD format. 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 MGLD format. 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 182](#)) they can provide a way to show animation. Also you can write several frames into animated GIF file.

```
int NewFrame ()                                [Method on mglGraph]
int 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 ResetFrames ()                                [Method on mglGraph]
int mgl_reset_frames (HMGL gr)                  [C function]
```

Reset frames counter (start it from zero).

```
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.

```
int 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 182](#)) or save in file (see also, [Section 4.5.1 \[Export to file\], page 130](#)).

```
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, float *x, float *y,      [C function]
                  float *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 126).
```

mglPoint CalcScr (mglPoint p)	[Method on mglGraph]
void mgl_calc_scr (HMGL gr, float x, float y, float 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 126).

```

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 (long xs, long ys)	[Method on mglGraph]
int mgl_get_obj_id (HMGL gr, long xs, long ys)	[C function]

Get the numeric id for most upper object at pixel {xs, ys} of the picture.

```

int GetSplId (long xs, long ys)                                     [Method on mglGraph]
int mgl_get_spl_id (HMGL gr, long xs, long 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.
```

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. 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_recv (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 Primitives

These functions draw some simple objects like line, point, sphere, drop, cone and so on. See [Section 2.9.6 \[Using primitives\], page 90](#), for sample code and picture.

```
clf                                              [MGL command]
void Clf ()                                     [Method on mglGraph]
void Clf (float r, float g, float b)           [Method on mglGraph]
void mgl_clf (HMGL gr)                          [C function]
void mgl_clf_rgb (HMGL gr, float r, float g, float b) [C function]
    Clear the picture and fill it by color specified color.
```

```
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, float x, float y, float z, const char
               *mark)                           [C function]
```

Draws a mark (point ‘.’ by default) at position *p*={*x*, *y*, *z*} with color *col*.

```

void Error (mglPoint p, mglPoint e, char *stl="")           [Method on mglGraph]
void mgl_error_box (HMGL gr, float px, float py, float pz, float
                     ex, float ey, float ez, char *stl)
    Draws a 3d error box at position p with sizes e 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",
            int num=2)                                         [Method on mglGraph]
void mgl_line (HMGL gr, float x1, float y1, float z1, float x2,
                float y2, float z2, char *stl, int num)          [C function]
    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
    straight line will be drawn in all coordinate system (independently on transformation
    formulas (see Section 4.3.2 \[Curved coordinates\], page 123). 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,
            mglPoint d2, const char *stl="B", int num=100)      [Method on mglGraph]
void mgl_curve (HMGL gr, float x1, float y1, float z1, float dx1,
                 float dy1, float dz1, float x2, float y2, float z2, float dx2, float
                 dy2, float dz2, const char *stl, int num)          [C function]
    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 coor-
    dinate system (independently on transformation formulas, see Section 4.3.2 \[Curved
    coordinates\], page 123). 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.

void Face (mglPoint p1, mglPoint p2, mglPoint p3,
            mglPoint p4, const char *stl="w")                   [Method on mglGraph]
void mgl_face (HMGL gr, float x1, float y1, float z1, float x2,
                float y2, float z2, float x3, float y3, float z3, float x4, float y4,
                float z4, const char *stl)                         [C function]
    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.

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 (float x0, float y0, float z0, float wy,        [Method on mglGraph]
            float wz, const char *stl="w", float d1=0, float d2=0)

```

```

void FaceY (float x0, float y0, float z0, float wx,           [Method on mglGraph]
            float wz, const char *stl="w", float d1=0, float d2=0)
void FaceZ (float x0, float y0, float z0, float wx,           [Method on mglGraph]
            float wy, const char *stl="w", float d1=0, float d2=0)
void mgl_facex (HMGL gr, float x0, float y0, float z0, float wy,    [C function]
                float wz, const char *stl, float d1, float d2)
void mgl_facey (HMGL gr, float x0, float y0, float z0, float wx,    [C function]
                float wz, const char *stl, float d1, float d2)
void mgl_facez (HMGL gr, float x0, float y0, float z0, float wx,    [C function]
                float wy, const char *stl, float d1, float 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).

sphere x0 y0 r ['col'='r']                                [MGL command]
sphere x0 y0 z0 r ['col'='r']                               [MGL command]
void Sphere (mglPoint p, float r, const char *stl="r")    [Method on mglGraph]
void mgl_sphere (HMGL gr, float x0, float y0, float z0, float 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, float r, const char     [Method on mglGraph]
           *col="r", float shift=1, float ap=1)
void mgl_drop (HMGL gr, float x0, float y0, float z0, float dx,   [C function]
               float dy, float dz, float r, const char *col, float shift, float 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'="" edge=off]      [MGL command]
void Cone (mglPoint p1, mglPoint p2, float r1, float       [Method on mglGraph]
           r2=-1, const char *stl="B", bool edge=false)
void mgl_cone (HMGL gr, float x1, float y1, float z1, float x2,   [C function]
               float y2, float z2, float r1, float r2, const char *stl, int draw_edge)
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.

circle x0 y0 r ['col'='r']                                [MGL command]
circle x0 y0 z0 r ['col'='r']                               [MGL command]
void Circle (mglPoint p, float r, const char *stl="r")    [Method on mglGraph]
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' sh=1 asp=1] [MGL command]
ellipse x1 y1 z1 x2 y2 z2 r ['col'='r' sh=1 asp=1] [MGL command]
void Ellipse (mglPoint p1, mglPoint p2, float r, const char *col="r") [Method on mglGraph]
```

```
void mgl_ellipse (HMGL gr, float x1, float y1, float z1, float x2, float y2, float z2, float r, const char *col) [C function]
```

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' sh=1 asp=1] [MGL command]
rhomb x1 y1 z1 x2 y2 z2 r ['col'='r' sh=1 asp=1] [MGL command]
void Rhomb (mglPoint p1, mglPoint p2, float r, const char *col="r") [Method on mglGraph]
```

```
void mgl_rhomb (HMGL gr, float x1, float y1, float z1, float x2, float y2, float z2, float r, const char *col) [C function]
```

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.

4.7 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 110](#)). 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 119](#).

The font parameters are described by string. This string may set the text color ‘`wkrgrbcymhRGBCYMHW`’ (see [Section 3.2 \[Color styles\], page 108](#)). Also, after delimiter symbol ‘`:`’, it can contain characters of font type (‘`rbiwou`’) and/or align (‘`LRC`’) 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. 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 subplot (for ‘a’) or picture (for ‘A’). If string contains symbol ‘@’ then box around text is drawn.

See [Section 2.2.7 \[Text features\]](#), page 26, for sample code and picture.

<code>text x y 'text' ['fnt'="" size=-1]</code>	[MGL command]
<code>text x y z 'text' ['fnt'="" size=-1]</code>	[MGL command]
<code>void Puts (mglPoint p, const char *text, const char *fnt=:C", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void Putsw (mglPoint p, const wchar_t *text, const char *fnt=:C", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void Puts (float x, float y, const char *text, const char *fnt=:AC", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void Putsw (float x, float y, const wchar_t *text, const char *fnt=:AC", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_puts (HMGL gr, float x, float y, float z, const char *text, const char *fnt, float size)</code>	[C function]
<code>void mgl_putsw (HMGL gr, float x, float y, float z, const wchar_t *text, const char *fnt, float size)</code>	[C function]

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 *FontSize*).

<code>text x y dx dy 'text' ['fnt':':L' size=-1]</code>	[MGL command]
<code>text x y z dx dy dz 'text' ['fnt':':L' size=-1]</code>	[MGL command]
<code>void Puts (mglPoint p, mglPoint d, const char *text, const char *fnt=:L", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void Putsw (mglPoint p, mglPoint d, const wchar_t *text, const char *fnt=:L", float size=-1)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_puts_dir (HMGL gr, float x, float y, float z, float dx, float dy, float dz, const char *text, const char *fnt, float size)</code>	[C function]
<code>void mgl_putsw_dir (HMGL gr, float x, float y, float z, float dx, float dy, float dz, const wchar_t *text, const char *fnt, float size)</code>	[C function]

The function plots the string *text* at position *p* along direction *d* with specified *size*. Parameter *fnt* set text style and text position: above (‘T’) or under (‘t’) the line.

<code>fgets x y 'fname' [n=0 'fnt'="" size=-1.4]</code>	[MGL command]
<code>fgets x y z 'fname' [n=0 'fnt'="" size=-1.4]</code>	[MGL command]

Draws unrotated *n*-th line of file *fname* at position {x,y,z} with specified *size*. By default parameters from [\[font\]](#), page 119 command are used.

<code>text ydat 'text' ['fnt'=""</code>	[MGL command]
<code>text xdat ydat 'text' ['fnt'=""</code>	[MGL command]
<code>text xdat ydat zdat 'text' ['fnt'=""</code>	[MGL command]
<code>void Text (const mglData &y, const char *text, const char *fnt="", const char *opt="")</code>	[Method on <code>mglGraph</code>]

```

void Text (const mglData &y, const wchar_t *text,           [Method on mglGraph]
           const char *fnt="", const char *opt="")
void Text (const mglData &x, const mglData &y, const      [Method on mglGraph]
           char *text, const char *fnt="", const char *opt="")
void Text (const mglData &x, const mglData &y, const      [Method on mglGraph]
           wchar_t *text, const char *fnt="", const char *opt="")
void Text (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &z, const char *text, const char *fnt="", const char
           *opt="")
void Text (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &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 *text,   [C function]
                   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 interval [Min.x, Max.x] (see [Section 4.3.1 \[Ranges \(bounding box\)\], page 122](#)). If array *z* is not specified then *z*[i] = Min.z is used. String *opt* contain command options (see [Section 3.7 \[Command options\], page 113](#)).

4.8 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 122](#).

```

axis ['dir='xyz' 'stl='"]                                [MGL command]
void Axis (const char *dir="xyz", const char *stl="")    [Method on mglGraph]
void mgl_axis (HMGL gr, const char *dir, const char *stl) [C function]

```

Draws axes with ticks (see [Section 4.3 \[Axis settings\], page 122](#)). Parameter *dir* may contain:

- ‘xyz’ for drawing axis in corresponding direction;
- ‘_’ for disabling tick labels;
- ‘AKDTVISO’ for drawing arrow at the end of axis;

- ‘a’ for forced adjusting of axis ticks.

Styles of ticks and axis can be overrided by using *stl* string. See [Section 2.2.2 \[Axis and ticks\]](#), page 17, for sample code and picture.

<code>colorbar ['sch'=""]</code>	[MGL command]
<code>void Colorbar (const char *sch="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_colorbar (HMGL gr, const char *sch)</code>	[C function]

Draws colorbar. Parameter *sch* may contain:

- color scheme (see [Section 3.4 \[Color scheme\]](#), page 109);
- ‘<>^_’ 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.

See [Section 2.2.4 \[Colorbars\]](#), page 22, for sample code and picture.

<code>colorbar vdat ['sch'=""]</code>	[MGL command]
<code>void Colorbar (const mglData &v, const char *sch="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_colorbar_val (HMGL gr, HCDT v, const char *sch)</code>	[C function]

The same as previous but with sharp colors *sch* (current palette if *sch*=“”) for values *v*. See [Section 2.6.13 \[ContD sample\]](#), page 65, for sample code and picture.

<code>colorbar 'sch' x y [w=1 h=1]</code>	[MGL command]
<code>void Colorbar (const char *sch, float x, float y,</code>	[Method on <code>mglGraph</code>]
<code>float w=1, float h=1)</code>	
<code>void mgl_colorbar_ext (HMGL gr, const char *sch, float x, float y,</code>	[C function]
<code>float w, float h)</code>	

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.

<code>colorbar vdat 'sch' x y [w=1 h=1]</code>	[MGL command]
<code>void Colorbar (const mglData &v, const char *sch,</code>	[Method on <code>mglGraph</code>]
<code>float x, float y, float w=1, float h=1)</code>	
<code>void mgl_colorbar_val_ext (HMGL gr, HCDT v, const char *sch,</code>	[C function]
<code>float x, float y, float w, float h)</code>	

The same as previous but with sharp colors *sch* (current palette if *sch*=“”) for values *v*. See [Section 2.6.13 \[ContD sample\]](#), page 65, for sample code and picture.

<code>grid ['dir'='xyz' 'pen'='B']</code>	[MGL command]
<code>void Grid (const char *dir="xyz", const char</code>	[Method on <code>mglGraph</code>]
<code>*pen="B")</code>	
<code>void mgl_axis_grid (HMGL gr, const char *dir, const char *pen)</code>	[C function]

Draws grid lines perpendicular to direction determined by string parameter *dir*. The step of grid lines is the same as tick step for [\[axis\]](#), page 141. The style of lines is determined by *pen* parameter (default value is dark blue solid line ‘B-’).

<code>box ['stl'='k' ticks=on]</code>	[MGL command]
<code>void Box (const char *col="", bool ticks=true)</code>	[Method on <code>mglGraph</code>]
<code>void mgl_box (HMGL gr, int ticks)</code>	[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 23, for sample code and
    picture.

xlabel 'text' [pos=1 shift=0]                                     [MGL command]
ylabel 'text' [pos=1 shift=0]                                     [MGL command]
zlabel 'text' [pos=1 shift=0]                                     [MGL command]
tlabel 'text' [pos=1 shift=0]                                     [MGL command]
void Label (char dir, const char *text, float pos=1,           [Method on mglGraph]
            float shift=0)
void Label (char dir, const wchar_t *text, float pos=1,         [Method on mglGraph]
            float shift=0)
void mgl_label (HMGL gr, char dir, const char *text)           [C function]
void mgl_label_ext (HMGL gr, char dir, const char *text, float   [C function]
                    pos, float shift)
void mgl_labelw_ext (HMGL gr, char dir, const wchar_t *text,      [C function]
                     float pos, float shift)
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. See Section 4.7 \[Text printing\], page 139.

```

4.9 Legend

These functions draw legend to the graph (useful for [Section 4.10 \[1D plotting\]](#), page 144). 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 144) and further plotting it. The position of the legend can be selected automatic or manually (even out of bounding box). Parameters *font* and *size* specify the font style and size (see [Section 4.2.6 \[Font settings\]](#), page 119). 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 *font* may contain:

- font style for legend text;
- ‘A’ for positioning in absolute coordinates;
- ‘#’ for drawing box around legend;
- colors for background (first one) and border (second one) of legend. Note, that last color is always used as color for legend text.

See [Section 2.2.8 \[Legend sample\]](#), page 27, for sample code and picture.

```

legend [pos=3 'font'='#' size=-0.8 llen=0.1]                   [MGL command]
void Legend (int pos=0x3, const char *font="#" , float          [Method on mglGraph]
             size=-0.8, float llen=0.1)

```

```
void mgl_legend (HMGL gr, int pos, const char *fnt, float size,           [C function]
                 float llen)
```

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). Parameter *fnt* can contain 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). If string *fnt* contain ‘#’ then border around the legend is drawn.

```
legend x y ['fnt'='#' size=-1 llen=0.1]                      [MGL command]
void Legend (float x, float y, const char *fnt="#" ,           [Method on mglGraph]
             float size=-0.8, float llen=0.1)
void mgl_legend_pos (HMGL gr, float x, float y, const char *fnt,   [C function]
                     float size, float llen)
```

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].

```
addlegend 'text' 'stl'                                     [MGL command]
void AddLegend (const char *text, const char *style)      [Method on mglGraph]
void AddLegend (const wchar_t *text, const char
                *style)                                [Method on mglGraph]
void mgl_add_legend (HMGL gr, const char *text, const char
                     *style)                               [C function]
void mgl_add_legendw (HMGL gr, const wchar_t *text, const char
                      *style)                             [C function]
```

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 108](#)).

```
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.
```

4.10 1D plotting

These functions perform plotting of 1D data. 1D means that data depended from only 1 parameter like parametric curve {x[i],y[i],z[i]}, i=1...n. By default (if absent) values of x[i] are equidistantly distributed in axis range, and z[i]=Min.z. 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 108](#)). By default (*pen=""*) solid line with color from palette is used (see [Section 4.2.7](#)

[Palette and colors], page 120). 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 113). See Section 2.5 [1D samples], page 36, for sample code and picture.

<code>plot ydat ['stl'="]</code>	[MGL command]
<code>plot xdat ydat ['stl'="]</code>	[MGL command]
<code>plot xdat ydat zdat ['stl'="]</code>	[MGL command]
<code>void Plot (const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Plot (const mglData &x, const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Plot (const mglData &x, const mglData &y, const mglData &z, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_plot (HMGL gr, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_plot_xy (HMGL gr, HCDT x, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_plot_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char *pen, const char *opt)</code>	[C function]

These functions draw continuous lines between points $\{x[i], y[i], z[i]\}$. See also [area], page 147, [step], page 145, [stem], page 147, [tube], page 153, [mark], page 151, [error], page 150, [belt], page 155, [tens], page 146, [tape], page 146. See Section 2.5.1 [Plot sample], page 37, for sample code and picture.

<code>radar adat ['stl'="]</code>	[MGL command]
<code>void Radar (const mglData &a, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_radar (HMGL gr, HCDT a, const char *pen, const char *opt)</code>	[C function]

This functions draws radar chart which is continuous lines between points located on an radial lines (like plot in Polar coordinates). Parameter *value* in options *opt* 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* contain ' #' symbol then "grid" (radial lines and circle for r) is drawn. See also [plot], page 145. See Section 2.5.2 [Radar sample], page 38, for sample code and picture.

<code>step ydat ['stl'="]</code>	[MGL command]
<code>step xdat ydat ['stl'="]</code>	[MGL command]
<code>step xdat ydat zdat ['stl'="]</code>	[MGL command]
<code>void Step (const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Step (const mglData &x, const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Step (const mglData &x, const mglData &y, const mglData &z, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_step (HMGL gr, HCDT y, const char *pen, const char *opt)</code>	[C function]

```
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 145, [\[stem\]](#), page 147, [\[tile\]](#), page 156, [\[boxs\]](#), page 155. See [Section 2.5.3 \[Step sample\]](#), page 39, 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 mglData &y, const mglData &c, const      [Method on mglGraph]
            char *pen="", const char *opt="")
void Tens (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &c, const char *pen="", const char *opt="")
void Tens (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const mglData &c, const char *pen="", const char
            *opt="")
void mgl_tens (HMGL gr, HCDT y, HCDT c, const char *pen, const [C function]
               char *opt)
void mgl_tens_xy (HMGL gr, HCDT x, HCDT y, HCDT c, const char      [C function]
                  *pen, const char *opt)
void mgl_tens_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT c, const [C function]
                   char *pen, const char *opt)
```

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 109) and style and/or width of line (see [Section 3.3 \[Line styles\]](#), page 108). See also [\[plot\]](#), page 145, [\[mesh\]](#), page 154, [\[fall\]](#), page 155. See [Section 2.5.4 \[Tens sample\]](#), page 40, 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 mglData &y, const char *pen="",      [Method on mglGraph]
            const char *opt="")
void Tape (const mglData &x, const mglData &y, const      [Method on mglGraph]
            char *pen="", const char *opt="")
void Tape (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const char *pen="", const char *opt="")
void mgl_tape (HMGL gr, HCDT y, const char *pen, const [C function]
               char *opt)
void mgl_tape_xy (HMGL gr, HCDT x, HCDT y, const char      [C function]
                  *pen, const char *opt)
void mgl_tape_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char      [C function]
                   *pen, const char *opt)
```

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 `SetBarWidth()`. See also [\[plot\]](#), page 145,

[flow], page 169, [barwidth], page 118. See Section 2.5.19 [Tape sample], page 53, for sample code and picture.

<code>area ydat ['stl'=""]</code>	[MGL command]
<code>area xdat ydat ['stl'=""]</code>	[MGL command]
<code>area xdat ydat zdat ['stl'=""]</code>	[MGL command]
<code>void Area (const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Area (const mglData &x, const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Area (const mglData &x, const mglData &y, const mglData &z, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_area (HMGL gr, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_area_xy (HMGL gr, HCDT x, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_area_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char *pen, const char *opt)</code>	[C function]

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 145, [bars], page 148, [stem], page 147, [region], page 147. See Section 2.5.5 [Area sample], page 41, for sample code and picture.

<code>region ydat1 ydat2 ['stl'=""]</code>	[MGL command]
<code>region xdat ydat1 ydat2 ['stl'=""]</code>	[MGL command]
<code>void Region (const mglData &y1, const mglData &y2, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Region (const mglData &x, const mglData &y1, const mglData &y2, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_region (HMGL gr, HCDT y1, HCDT y2, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_region_xy (HMGL gr, HCDT x, HCDT y1, HCDT y2, const char *pen, const char *opt)</code>	[C function]

These functions fill area between 2 curves. Dimensions of arrays y_1 and y_2 must be equal. Also you can use gradient filling if number of specified colors is equal to 2*number of curves. If pen contain symbol ‘i’ then only area with $y_1 < y < y_2$ will be filled else the area with $y_2 < y < y_1$ will be filled too. See also [area], page 147, [bars], page 148, [stem], page 147. See Section 2.5.6 [Region sample], page 42, for sample code and picture.

<code>stem ydat ['stl'=""]</code>	[MGL command]
<code>stem xdat ydat ['stl'=""]</code>	[MGL command]
<code>stem xdat ydat zdat ['stl'=""]</code>	[MGL command]
<code>void Stem (const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void Stem (const mglData &x, const mglData &y, const char *pen="", const char *opt="")</code>	[Method on <code>mglGraph</code>]

```

void Stem (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &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 147, [\[bars\]](#), page 148, [\[plot\]](#), page 145, [\[mark\]](#), page 151. See [Section 2.5.7 \[Stem sample\]](#), [page 43](#), 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 mglData &y, const char *pen="",      [Method on mglGraph]
            const char *opt="")
void Bars (const mglData &x, const mglData &y, const      [Method on mglGraph]
            char *pen="", const char *opt="")
void Bars (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &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. See also [\[barh\]](#), page 148, [\[cones\]](#), page 149, [\[area\]](#), page 147, [\[stem\]](#), page 147, [\[chart\]](#), page 149, [\[barwidth\]](#), page 118. See [Section 2.5.8 \[Bars sample\]](#), [page 44](#), for sample code and picture.

```

barh vdat ['stl="']                                [MGL command]
barh ydat vdat ['stl="']                            [MGL command]
void Barh (const mglData &v, const char *pen="",      [Method on mglGraph]
            const char *opt="")
void Barh (const mglData &y, const mglData &v, const      [Method on mglGraph]
            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. See also [bars], page 148, [barwidth], page 118. See Section 2.5.9 [Barh sample], page 45, for sample code and picture.

<code>cones ydat ['stl'="]</code>	[MGL command]
<code>cones xdat ydat ['stl'="]</code>	[MGL command]
<code>cones xdat ydat zdat ['stl'="]</code>	[MGL command]
<code>void Cones (const mglData &y, const char *pen="", const char *opt="")</code>	[Method on mglGraph]
<code>void Cones (const mglData &x, const mglData &y, const char *pen="", const char *opt="")</code>	[Method on mglGraph]
<code>void Cones (const mglData &x, const mglData &y, const mglData &z, const char *pen="", const char *opt="")</code>	[Method on mglGraph]
<code>void mgl_cones (HMGL gr, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_cones_xy (HMGL gr, HCDT x, HCDT y, const char *pen, const char *opt)</code>	[C function]
<code>void mgl_cones_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, const char *pen, const char *opt)</code>	[C function]

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. See also [bars], page 148, [barwidth], page 118. See Section 2.5.10 [Cones sample], page 46, for sample code and picture.

<code>chart adat ['col'="]</code>	[MGL command]
<code>void Chart (const mglData &a, const char *col="", const char *opt="")</code>	[Method on mglGraph]
<code>void mgl_chart (HMGL gr, HCDT a, const char *col, const char *opt)</code>	[C function]

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 120). 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 47, for sample code and picture.

<code>boxplot adat ['stl'="]</code>	[MGL command]
<code>boxplot xdat adat ['stl'="]</code>	[MGL command]
<code>void BoxPlot (const mglData &a, const char *pen="", const char *opt="")</code>	[Method on mglGraph]
<code>void BoxPlot (const mglData &x, const mglData &a, const char *pen="", const char *opt="")</code>	[Method on mglGraph]

```
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. See also [plot], page 145, [error], page 150, [bars], page 148, [barwidth], page 118. See Section 2.5.12 [BoxPlot sample], page 48, 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 mglData &v1, const char *pen="",      [Method on mglGraph]
             const char *opt="")
void Candle (const mglData &v1, const mglData &v2,      [Method on mglGraph]
             const char *pen="", const char *opt="")
void Candle (const mglData &v1, const mglData &y1,      [Method on mglGraph]
             const mglData &y2, const char *pen="", const char *opt="")
void Candle (const mglData &v1, const mglData &v2,      [Method on mglGraph]
             const mglData &y1, const mglData &y2, const char *pen="", const
             char *opt="")
void Candle (const mglData &x, const mglData &v1,      [Method on mglGraph]
             const mglData &v2, const mglData &y1, const mglData &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 y2,      [C function]
                     const char *pen, const char *opt)
void mgl_candle_xyv (HMGL gr, HCDT x, HCDT v1, HCDT v2, HCDT y1,      [C function]
                     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. "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 145, [bars], page 148, [barwidth], page 118. See Section 2.5.13 [Candle sample], page 49, 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 mglData &y, const mglData &ey,      [Method on mglGraph]
            const char *pen="", const char *opt="")
void Error (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &ey, const char *pen="", const char *opt="")
```

```

void Error (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &ex, const mglData &ey, const char *pen="", const char
            *opt="")
void mgl_error (HMGL gr, HCDT y, HCDT ey, const char *pen, const [C function]
                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 145, [\[mark\]](#), page 151. See [Section 2.5.14](#) [\[Error sample\]](#), page 49, 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 mglData &y, const mglData &r, const [Method on mglGraph]
            char *pen="", const char *opt="")
void Mark (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &r, const char *pen="", const char *opt="")
void Mark (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const mglData &r, const char *pen="", const char
            *opt="")
void mgl_mark_y (HMGL gr, HCDT y, HCDT r, const char *pen, const [C function]
                 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, const [C function]
                   char *pen, const char *opt)

```

These functions draw marks with size $r[i]*[\text{marksiz}]$, page 119 at points $\{x[i], y[i], z[i]\}$. If you need to draw markers of the same size then you can use [\[plot\]](#), page 145 function with empty line style ‘.’. For markers with size in axis range use [\[error\]](#), page 150 with style ‘@’. See also [\[plot\]](#), page 145, [\[textmark\]](#), page 151, [\[error\]](#), page 150, [\[stem\]](#), page 147. See [Section 2.5.15](#) [\[Mark sample\]](#), page 50, 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 mglData &y, const char *txt, [Method on mglGraph]
               const char *fnt="", const char *opt="")
void TextMark (const mglData &y, const wchar_t *txt, [Method on mglGraph]
               const char *fnt="", const char *opt="")
void TextMark (const mglData &y, const mglData &r, [Method on mglGraph]
               const char *txt, const char *fnt="", const char *opt="")

```

```

void TextMark (const mglData &y, const mglData &r,      [Method on mglGraph]
               const wchar_t *txt, const char *fnt="", const char *opt="")
void TextMark (const mglData &x, const mglData &y,      [Method on mglGraph]
               const mglData &r, const char *txt, const char *fnt="", const char
               *opt="")
void TextMark (const mglData &x, const mglData &y,      [Method on mglGraph]
               const mglData &r, const wchar_t *txt, const char *fnt="", const char
               *opt="")
void TextMark (const mglData &x, const mglData &y,      [Method on mglGraph]
               const mglData &z, const mglData &r, const char *txt, const char
               *fnt="", const char *opt="")
void TextMark (const mglData &x, const mglData &y,      [Method on mglGraph]
               const mglData &z, const mglData &r, const wchar_t *txt, const char
               *fnt="", const char *opt="")
void mgl_textmark (HMGL gr, HCDT y, const char *txt, const char      [C function]
                  *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_xy (HMGL gr, HCDT x, HCDT y, HCDT r, const      [C function]
                      char *txt, const char *fnt, const char *opt)
void mgl_textmarkw_xy (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 r,      [C function]
                        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 145,
[mark], page 151, [stem], page 147. See Section 2.5.16 [TextMark sample], page 51,
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 mglData &y, const char *txt, const      [Method on mglGraph]
            char *fnt="", const char *opt="")
void Label (const mglData &y, const wchar_t *txt,           [Method on mglGraph]
            const char *fnt="", const char *opt="")
void Label (const mglData &x, const mglData &y, const      [Method on mglGraph]
            char *txt, const char *fnt="", const char *opt="")
void Label (const mglData &x, const mglData &y, const      [Method on mglGraph]
            wchar_t *txt, const char *fnt="", const char *opt="")

```

```

void Label (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const char *txt, const char *fnt="", const char *opt="")
void Label (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &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 *txt, [C function]
                    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. See also [plot], page 145, [mark], page 151, [textmark], page 151. See Section 2.5.17 [Label sample], page 52, 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 mglData &y, const mglData &r, const [Method on mglGraph] char *pen="", const char *opt="")	
void Tube (const mglData &y, float r, const char [Method on mglGraph] *pen="", const char *opt="")	
void Tube (const mglData &x, const mglData &y, const [Method on mglGraph] mglData &r, const char *pen="", const char *opt="")	
void Tube (const mglData &x, const mglData &y, float [Method on mglGraph] r, const char *pen="", const char *opt="")	
void Tube (const mglData &x, const mglData &y, const [Method on mglGraph] mglData &z, const mglData &r, const char *pen="", const char *opt="")	
void Tube (const mglData &x, const mglData &y, const [Method on mglGraph] mglData &z, float r, const char *pen="", const char *opt="")	
void mgl_tube_r (HMGL gr, HCDT y, HCDT r, const char *pen, const [C function] char *opt)	
void mgl_tube (HMGL gr, HCDT y, float 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, float 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, float 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 145. See [Section 2.5.18 \[Tube sample\]](#), page 53, for sample code and picture.

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

```
void Torus (const mglData &r, const mglData &z, const      [Method on mglGraph]
            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’). See also [\[plot\]](#), page 145, [\[axial\]](#), page 159. See [Section 2.5.20 \[Torus sample\]](#), page 54, for sample code and picture.

4.11 2D plotting

These functions perform plotting of 2D data. 2D means that data depend from 2 independent parameters like matrix $f(x_i, y_j), i = 1\dots n, j = 1\dots m$. 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 109) for plot. String *opt* contain command options (see [Section 3.7 \[Command options\]](#), page 113). See [Section 2.6 \[2D samples\]](#), page 55, for sample code and picture.

```
surf zdat ['sch'='']                                     [MGL command]
```

```
surf xdat ydat zdat ['sch'='']                           [MGL command]
```

```
void Surf (const mglData &z, const char *sch="",      [Method on mglGraph]
           const char *opt="")
```

```
void Surf (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &z, const char *sch="", const char *opt="")
```

```
void mgl_surf (HMGL gr, HCDT z, const char *sch, const char *opt)  [C function]
```

```
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. See also [\[mesh\]](#), page 154, [\[dens\]](#), page 156, [\[belt\]](#), page 155, [\[tile\]](#), page 156, [\[boxs\]](#), page 155, [\[surf\]](#), page 164, [\[surfa\]](#), page 165. See [Section 2.6.1 \[Surf sample\]](#), page 56, for sample code and picture.

```
mesh zdat ['sch'='']                                     [MGL command]
```

```
mesh xdat ydat zdat ['sch'='']                           [MGL command]
```

```
void Mesh (const mglData &z, const char *sch="",      [Method on mglGraph]
           const char *opt="")
```

```
void Mesh (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &z, const char *sch="", const char *opt="")
```

```
void mgl_mesh (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
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 154, [\[fall\]](#), page 155, [\[meshnum\]](#), page 119, [\[cont\]](#), page 156, [\[tens\]](#), page 146. See [Section 2.6.4 \[Mesh sample\]](#), page 58, for sample code and picture.

```
fall zdat ['sch']='' [MGL command]
```

```
fall xdat ydat zdat ['sch']='' [MGL command]
```

```
void Fall (const mglData &z, const char *sch="", [Method on mglGraph]
            const char *opt="")
```

```
void Fall (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const char *sch="", const char *opt="")
```

```
void mgl_fall (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
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 155, [\[mesh\]](#), page 154, [\[tens\]](#), page 146, [\[meshnum\]](#), page 119. See [Section 2.6.5 \[Fall sample\]](#), page 59, for sample code and picture.

```
belt zdat ['sch']='' [MGL command]
```

```
belt xdat ydat zdat ['sch']='' [MGL command]
```

```
void Belt (const mglData &z, const char *sch="", [Method on mglGraph]
            const char *opt="")
```

```
void Belt (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const char *sch="", const char *opt="")
```

```
void mgl_belt (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
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 145). If sch contain ‘x’ then belts are drawn along x-direction else (by default) belts are drawn along y-direction. See also [\[fall\]](#), page 155, [\[surf\]](#), page 154, [\[plot\]](#), page 145, [\[meshnum\]](#), page 119. See [Section 2.6.6 \[Belt sample\]](#), page 60, for sample code and picture.

```
boxs zdat ['sch']='' [MGL command]
```

```
boxs xdat ydat zdat ['sch']='' [MGL command]
```

```
void Boxs (const mglData &z, const char *sch="", [Method on mglGraph]
            const char *opt="")
```

```
void Boxs (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const char *sch="", const char *opt="")
```

```
void mgl_boxs (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
void mgl_boxs_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]
```

The function draws vertical boxes for surface specified parametrically $\{x[i,j], y[i,j], z[i,j]\}$. See also [\[surf\]](#), page 154, [\[dens\]](#), page 156, [\[tile\]](#), page 156, [\[step\]](#), page 145. See [Section 2.6.7 \[Boxes sample\]](#), page 60, for sample code and picture.

```
tile zdat ['sch']='' [MGL command]
```

```
tile xdat ydat zdat ['sch']='' [MGL command]
```

```
void Tile (const mglData &z, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void Tile (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void mgl_tile (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
void mgl_tile_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]
```

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 145. See also [\[surf\]](#), page 154, [\[boxs\]](#), page 155, [\[step\]](#), page 145, [\[tiles\]](#), page 166. See [Section 2.6.8 \[Tile sample\]](#), page 61, for sample code and picture.

```
dens zdat ['sch']='' [MGL command]
```

```
dens xdat ydat zdat ['sch']='' [MGL command]
```

```
void Dens (const mglData &z, const char *sch="", const char *opt="", float zVal=NAN) [Method on mglGraph]
```

```
void Dens (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt="", float zVal=NAN) [Method on mglGraph]
```

```
void mgl_dens (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
```

```
void mgl_dens_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]
```

The function draws density plot for surface specified parametrically $\{x[i,j], y[i,j], z[i,j]\}$ at $z = \text{Min.z}$. If string sch have symbol '#' then grid lines are drawn. See also [\[surf\]](#), page 154, [\[cont\]](#), page 156, [\[conf\]](#), page 157, [\[boxs\]](#), page 155, [\[tile\]](#), page 156, [dens\[xyz\]](#). See [Section 2.6.10 \[Dens sample\]](#), page 63, for sample code and picture.

```
cont vdat zdat ['sch']='' [MGL command]
```

```
cont vdat xdat ydat zdat ['sch']='' [MGL command]
```

```
void Cont (const mglData &v, const mglData &z, const char *sch, const char *opt="") [Method on mglGraph]
```

```
void Cont (const mglData &v, const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void mgl_cont__val (HMGL gr, HCDT v, HCDT z, const char *sch, const char *opt) [C function]
```

```
void mgl_cont_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]
```

The function draws contour lines for surface specified parametrically $\{x[i,j], y[i,j], z[i,j]\}$ at $z=v[k]$ or at $z = \text{Min.z}$ 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 156, [contf], page 157, [contd], page 157, [axial], page 159, cont [xyz]. See Section 2.6.11 [Cont sample], page 63, for sample code and picture.

```
cont zdat ['sch']=''                                [MGL command]
cont xdat ydat zdat ['sch']=''                    [MGL command]
void Cont (const mglData &z, const char *sch="",      [Method on mglGraph]
           const char *opt="")
void Cont (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &z, const char *sch="", const char *opt="")
void mgl_cont (HMGL gr, HCDT z, const char *sch, const char *opt)    [C function]
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 mglData &v, const mglData &z, const      [Method on mglGraph]
            char *sch="", const char *opt="")
void ContF (const mglData &v, const mglData &x, const      [Method on mglGraph]
            mglData &y, const mglData &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 $z = \text{Min.z}$ if sch contain symbol '_'. Contours are plotted for $z[i,j]=v[k]$ where $v[k]$ are values of data array v (must be $v.\text{nx}>2$). See also [dens], page 156, [cont], page 156, [contd], page 157, contf [xyz]. See Section 2.6.12 [ContF sample], page 64, for sample code and picture.

```
contf zdat ['sch']=''                                [MGL command]
contf xdat ydat zdat ['sch']=''                    [MGL command]
void ContF (const mglData &z, const char *sch="",      [Method on mglGraph]
            const char *opt="")
void ContF (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &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 mglData &v, const mglData &z, const      [Method on mglGraph]
            char *sch="", const char *opt="")
void ContD (const mglData &v, const mglData &x, const      [Method on mglGraph]
            mglData &y, const mglData &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 $z = \text{Min.z}$ 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 $v.\text{nx}>2$). String sch sets the contour colors: the color of k -th contour is determined by character $sch[k\%strlen(sch)]$. See also [[dens](#)], page 156, [[cont](#)], page 156, [[conf](#)], page 157. See [Section 2.6.13 \[ContD sample\]](#), page 65, for sample code and picture.

```

contd zdat ['sch=']                                [MGL command]
contd xdat ydat zdat ['sch=']                      [MGL command]
void ContD (const mglData &z, const char *sch="",      [Method on mglGraph]
            const char *opt="")
void ContD (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &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 mglData &v, const mglData &z, const      [Method on mglGraph]
            char *sch="", const char *opt="")
void ContV (const mglData &v, const mglData &x, const      [Method on mglGraph]
            mglData &y, const mglData &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 at contour lines for surface specified parametrically $\{x[i,j], y[i,j], z[i,j]\}$ at $z=v[k]$ or at $z = \text{Min.z}$ 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 156, [[conf](#)], page 157. See [Section 2.6.14 \[ContV sample\]](#), page 66, for sample code and picture.

```

contv zdat ['sch=']                                [MGL command]
contv xdat ydat zdat ['sch=']                      [MGL command]

```

```

void ContV (const mglData &z, const char *sch="",           [Method on mglGraph]
            const char *opt="")
void ContV (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt= "") [Method on mglGraph]
void mgl_contv (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
void mgl_contv_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]

```

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 mglData &v, const mglData &z, const char *sch="", const char *opt= "") [Method on mglGraph]
void Axial (const mglData &v, const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt= "") [Method on mglGraph]
void mgl_axial_val (HMGL gr, HCDT v, HCDT z, const char *sch, const char *opt) [C function]
void mgl_axial_xy_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]

```

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 contain symbols 'x' or 'z' then rotation axis will be set to specified direction (default is 'y'). See also [cont], page 156, [contf], page 157, [torus], page 154, [surf3], page 160. See Section 2.6.15 [Axial sample], page 67, for sample code and picture.

```

axial zdat ['sch'='] [MGL command]
axial xdat ydat zdat ['sch'='] [MGL command]
void Axial (const mglData &z, const char *sch="", const char *opt="", int num=3) [Method on mglGraph]
void Axial (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt="", int num=3) [Method on mglGraph]
void mgl_axial (HMGL gr, HCDT z, const char *sch, const char *opt) [C function]
void mgl_axial_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char *sch, const char *opt) [C function]

```

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 mglData &z, const char *sch="", const char *opt= "") [Method on mglGraph]
void Grid (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt= "") [Method on mglGraph]

```

```
void mgl_grid (HMGL gr, HCDT z, const char *sch, const char *opt)      [C function]
void mgl_grid_xy (HMGL gr, HCDT x, HCDT y, HCDT z, const char
                  *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 $z = \text{Min.z}$. See also [\[dens\]](#), page 156, [\[cont\]](#), page 156, [\[conf\]](#), page 157, [\[meshnum\]](#), page 119.

4.12 3D plotting

These functions perform plotting of 3D data. 3D means that data depend from 3 independent parameters like matrix $f(x_i, y_j, z_k), i = 1\dots n, j = 1\dots m, k = 1\dots l$. 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 $x.\text{nx}=a.\text{nx} \&& y.\text{nx}=a.\text{ny} \&& z.\text{nz}=a.\text{nz}$ or $x.\text{nx}=y.\text{nx}=z.\text{nx}=a.\text{nx} \&& x.\text{ny}=y.\text{ny}=z.\text{ny}=a.\text{ny} \&& x.\text{nz}=y.\text{nz}=z.\text{nz}=a.\text{nz}$. 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 109) for plot. String opt contain command options (see [Section 3.7 \[Command options\]](#), page 113). See [Section 2.7 \[3D samples\]](#), page 69, for sample code and picture.

```
surf3 adat val ['sch'=""]                                     [MGL command]
surf3 xdat ydat zdat adat val ['sch'=""]                     [MGL command]
void Surf3 (float val, const mglData &a, const char
            *sch="", const char *opt="")
void Surf3 (float val, const mglData &x, const
            mglData &y, const mglData &z, const mglData &a, const char *sch="",
            const char *opt="")
void mgl_surf3_val (HMGL gr, float val, HCDT a, const char *sch,
                    const char *opt)                                [C function]
void mgl_surf3_xyz_val (HMGL gr, float val, HCDT x, HCDT y, 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. 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 161, [\[dens3\]](#), page 161, [\[surf3c\]](#), page 164, [\[surf3a\]](#), page 165, [\[axial\]](#), page 159. See [Section 2.7.1 \[Surf3 sample\]](#), page 69, for sample code and picture.

```
surf3 adat ['sch'=""]                                     [MGL command]
surf3 xdat ydat zdat adat ['sch'=""]                     [MGL command]
void Surf3 (const mglData &a, const char *sch="",       [Method on mglGraph]
            const char *opt="")
void Surf3 (const mglData &x, const mglData &y, const
            mglData &z, const mglData &a, const char *sch="",
            const char *opt="")                                [Method on mglGraph]
void mgl_surf3 (HMGL gr, HCDT a, const char *sch, const char
                *opt)                                [C function]
```

```
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 mglData &a, const char *sch="",      [Method on mglGraph]
            const char *opt="")
```

```
void Cloud (const mglData &x, const mglData &y, const   [Method on mglGraph]
            mglData &z, const mglData &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 119](#). If string *sch* contain symbol ‘.’ then lower quality plot will produced with much low memory usage. If string *sch* contain symbol ‘!’ then transparency will be inversed, i.e. higher become transparent and lower become not transparent. See also [\[surf3\], page 160](#), [\[meshnum\], page 119](#). See [Section 2.7.4 \[Cloud sample\], page 71](#), 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 mglData &a, const char *sch="",      [Method on mglGraph]
            float sVal=-1, const char *opt="")
```

```
void Dens3 (const mglData &x, const mglData &y, const   [Method on mglGraph]
            mglData &z, const mglData &a, const char *sch="", float sVal=-1,
            const char *opt="")
```

```
void mgl_dens3 (HMGL gr, HCDT a, const char *sch, float 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, float 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 161](#), [\[contf3\], page 162](#), [\[dens\], page 156](#), [\[grid3\], page 163](#). See [Section 2.7.5 \[Dens3 sample\], page 72](#), 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 mglData &v, const mglData &a, const   [Method on mglGraph]
            char *sch="", float sVal=-1, const char *opt="")
```

```

void Cont3 (const mglData &v, const mglData &x, const [Method on mglGraph]
            mglData &y, const mglData &z, const mglData &a, const char *sch="",
            float sVal=-1, const char *opt="")
void mgl_cont3_val (HMGL gr, HCDT v, HCDT a, const char *sch,           [C function]
                    float sVal, const char *opt)
void mgl_cont3_xyz_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z,       [C function]
                        HCDT a, const char *sch, float 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 stl have symbol ‘#’ then grid lines are drawn. If string stl have symbol ‘t’ or ‘T’ then contour labels will be drawn below (or above) the contours. See also [dens3], page 161, [contf3], page 162, [cont], page 156, [grid3], page 163. See Section 2.7.6 [Cont3 sample], page 73, 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 mglData &a, const char *sch="",           [Method on mglGraph]
            float sVal=-1, const char *opt="", const char *opt="")
void Cont3 (const mglData &x, const mglData &y, const [Method on mglGraph]
            mglData &z, const mglData &a, const char *sch="", float sVal=-1,
            const char *opt="")
void mgl_cont3 (HMGL gr, HCDT a, const char *sch, float 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, float 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 mglData &v, const mglData &a,           [Method on mglGraph]
              const char *sch="", float sVal=-1, const char *opt="")
void Contf3 (const mglData &v, const mglData &x,           [Method on mglGraph]
              const mglData &y, const mglData &z, const mglData &a, const char
              *sch="", float sVal=-1, const char *opt="")
void mgl_contf3_val (HMGL gr, HCDT v, HCDT a, const char *sch,     [C function]
                     float sVal, const char *opt)
void mgl_contf3_xyz_val (HMGL gr, HCDT v, HCDT x, HCDT y, HCDT z,   [C function]
                         HCDT a, const char *sch, float 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 stl have symbol ‘#’ then grid lines are drawn. If string stl have symbol ‘t’ or ‘T’ then contour labels will be drawn below (or above) the contours. See also [dens3], page 161, [cont3], page 161, [contf], page 157, [grid3], page 163. See Section 2.7.6 [Cont3 sample], page 73, for sample code and picture.

```

conf3 adat ['sch'=' sval=-1] [MGL command]
conf3 xdat ydat zdat adat ['sch'=' sval=-1] [MGL command]
void Contf3 (const mglData &a, const char *sch="", [Method on mglGraph]
              float sVal=-1, const char *opt="", const char *opt="")
void Contf3 (const mglData &x, const mglData &y, [Method on mglGraph]
              const mglData &z, const mglData &a, const char *sch="", float
              sVal=-1, const char *opt="")
void mgl_contf3 (HMGL gr, HCDT a, const char *sch, float 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, float 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 mglData &a, const char *sch="", [Method on mglGraph]
             float sVal=-1, const char *opt="")
void Grid3 (const mglData &x, const mglData &y, const [Method on mglGraph]
             mglData &z, const mglData &a, const char *sch="", float sVal=-1,
             const char *opt="")
void mgl_grid3 (HMGL gr, HCDT a, const char *sch, float 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, float 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 161, [[contf3](#)], page 162, [[dens3](#)], page 161, [[grid2](#)], page 159, [[meshnum](#)], page 119.

```

beam tr g1 g2 adat rval ['sch'=' flag=0 num=3] [MGL command]
void Beam (const mglData &tr, const mglData &g1, [Method on mglGraph]
            const mglData &g2, const mglData &a, float r, const char *stl="", int
            flag=0, int num=3)
void Beam (float val, const mglData &tr, const mglData [Method on mglGraph]
            &g1, const mglData &g2, const mglData &a, float r, const char *stl="", int
            flag=0)
void mgl_beam (HMGL gr, HCDT tr, HCDT g1, HCDT g2, HCDT a, float r, [C function]
               const char *stl, int flag, int num)
void mgl_beam_val (HMGL gr, float val, HCDT tr, HCDT g1, HCDT g2, [C function]
                   HCDT a, float 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 $nx>2$. The y-size of data arrays $tr, g1, g2$ and z-size of the data array a must be equal. See also [[surf3](#)], page 160.

4.13 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 109](#)) for plot. String opt contain command options (see [Section 3.7 \[Command options\], page 113](#)).

```
surfC zdat cdat ['sch'=""]                                     [MGL command]
surfC xdat ydat zdat cdat ['sch'=""]                           [MGL command]
void SurfC (const mglData &z, const mglData &c, const      [Method on mglGraph]
            char *sch="", const char *opt="")
void SurfC (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const mglData &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, const      [C function]
                   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. All dimensions of arrays z and c must be equal. Surface is plotted for each z slice of the data. See also [\[surf\]](#), page 154, [\[surfa\]](#), page 165, [\[surf3c\]](#), page 164. See [Section 2.6.2 \[SurfC sample\], page 57](#), for sample code and picture.

```
surf3c adat cdat val ['sch'=""]                               [MGL command]
surf3c xdat ydat zdat adat cdat val ['sch'=""]             [MGL command]
void Surf3C (float val, const mglData &a, const           [Method on mglGraph]
              mglData &c, const char *sch="", const char *opt="")
void Surf3C (float val, const mglData &x, const           [Method on mglGraph]
              mglData &y, const mglData &z, const mglData &a, const mglData &c,
              const char *sch="", const char *opt="")
void mgl_surf3c_val (HMGL gr, float val, HCDT a, HCDT c, const      [C function]
                     char *sch, const char *opt)
void mgl_surf3c_xyz_val (HMGL gr, float 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 160 function but the color of isosurface depends on values of array c . If string contain '#' then wire plot is produced. See also [\[surf3\]](#), page 160, [\[surfC\]](#), page 164, [\[surf3a\]](#), page 165. See [Section 2.7.2 \[Surf3C sample\], page 70](#), for sample code and picture.

```
surf3c adat cdat ['sch'=""]                               [MGL command]
surf3c xdat ydat zdat adat cdat ['sch'=""]             [MGL command]
void Surf3C (const mglData &a, const mglData &c,           [Method on mglGraph]
              const char *sch="", const char *opt="")
```

```

void Surf3C (const mglData &x, const mglData &y,           [Method on mglGraph]
              const mglData &z, const mglData &a, const mglData &c, const char
              *sch="", const char *opt="")
void mgl_surf3c (HMGL gr, HCDT a, HCDT c, const char *sch, const   [C function]
                  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 mglData &z, const mglData &c, const   [Method on mglGraph]
            char *sch="", const char *opt="")
void SurfA (const mglData &x, const mglData &y, const   [Method on mglGraph]
            mglData &z, const mglData &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, const   [C function]
                   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. All dimensions of arrays *z* and *c* must be equal. Surface is plotted for each *z* slice of the data. See also [\[surf\]](#), page 154, [\[surfcl\]](#), page 164, [\[surf3a\]](#), page 165. See [Section 2.6.3](#) [[SurfA sample](#)], page 58, for sample code and picture.

```

surf3a adat cdat val ['sch']=''                          [MGL command]
surf3a xdat ydat zdat adat cdat val ['sch']=''        [MGL command]
void Surf3A (float val, const mglData &a, const         [Method on mglGraph]
              mglData &c, const char *sch="", const char *opt="")
void Surf3A (float val, const mglData &x, const         [Method on mglGraph]
              mglData &y, const mglData &z, const mglData &a, const mglData &c,
              const char *sch="", const char *opt="")
void mgl_surf3a_val (HMGL gr, float val, HCDT a, HCDT c, const   [C function]
                     char *sch, const char *opt)
void mgl_surf3a_xyz_val (HMGL gr, float 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 160 function but the color of isosurface depends on values of array *c*. If string contain '#' then wire plot is produced. See also [\[surf3\]](#), page 160, [\[surfcl\]](#), page 164, [\[surf3a\]](#), page 165. See [Section 2.7.3](#) [[Surf3A sample](#)], page 71, for sample code and picture.

```

surf3a adat cdat ['sch']=''                            [MGL command]
surf3a xdat ydat zdat adat cdat ['sch']=''          [MGL command]
void Surf3A (const mglData &a, const mglData &c,      [Method on mglGraph]
              const char *sch="", const char *opt="")

```

```

void Surf3A (const mglData &x, const mglData &y,           [Method on mglGraph]
              const mglData &z, const mglData &a, const mglData &c, const char
              *sch="", const char *opt="")
void mgl_surf3a (HMGL gr, HCDT a, HCDT c, const char *sch, const   [C function]
                  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).

```

tiles zdat rdat ['sch=']                                [MGL command]
tiles xdat ydat zdat rdat ['sch=']                      [MGL command]
void TileS (const mglData &z, const mglData &c, const   [Method on mglGraph]
             char *sch="", const char *opt="")
void TileS (const mglData &x, const mglData &y, const   [Method on mglGraph]
             mglData &z, const mglData &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, const   [C function]
                   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 156 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 165, [\[tile\]](#), page 156. See [Section 2.6.9 \[TileS sample\]](#), page 62, for sample code and picture.

```

map udat vdat ['sch=']                                [MGL command]
map xdat ydat udat vdat ['sch=']                      [MGL command]
void Map (const mglData &ax, const mglData &ay, const   [Method on mglGraph]
          char *sch="", const char *opt="")
void Map (const mglData &x, const mglData &y, const   [Method on mglGraph]
          mglData &ax, const mglData &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, const   [C function]
                  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 2.9.8 \[Mapping visualization\]](#), page 94, for sample code and picture.

```

stfa re im dn ['sch=']                                [MGL command]
stfa xdat ydat re im dn ['sch=']                      [MGL command]

```

```

void STFA (const mglData &re, const mglData &im, int      [Method on mglGraph]
           dn, const char *sch="", const char *opt="")
void STFA (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &re, const mglData &im, int dn, const char *sch="", const
           char *opt="")
void mgl_stfa (HMGL gr, HCDT re, HCDT im, int dn, const char *sch,    [C function]
               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 $z=Min.z$. For example in 1D case, result is density plot of data $res[i, j] = |\sum_d nexp(I * j * d) * (re[i * dn + d] + I * im[i * dn + d])|/dn$ with size {int(nx/dn), dn, ny}. At this array re , im parametrically depend on coordinates x , y . The size of re and im must be the same. The minor dimensions of arrays x , y , re should be equal. Arrays x , y can be vectors (not matrix as re). See [Section 2.9.7 \[STFA sample\]](#), page 93, for sample code and picture.

4.14 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 109) for plot. String opt contain command options (see [Section 3.7 \[Command options\]](#), page 113).

```

traj xdat ydat udat vdat ['sch=']                                [MGL command]
traj xdat ydat zdat udat vdat wdat ['sch=']                         [MGL command]
void Traj (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &ax, const mglData &ay, const char *sch="", const char
           *opt="")
void Traj (const mglData &x, const mglData &y, const      [Method on mglGraph]
           mglData &z, const mglData &ax, const mglData &ay, const mglData &az,
           const char *sch="", const char *opt="")
void mgl_traj_xyz (HMGL gr, HCDTx, HCDTy, HCDTz, HCDTax, HCDTay,    [C function]
                  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 108). By default ($pen=""$) color from palette is used (see [Section 4.2.7 \[Palette and colors\]](#), page 120). 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 168. See [Section 2.8.2 \[Traj sample\]](#), page 80, for sample code and picture.

```

vect udat vdat ['sch'=""] [MGL command]
vect xdat ydat udat vdat ['sch'=""] [MGL command]
void Vect (const mglData &ax, const mglData &ay, [Method on mglGraph]
           const char *sch="", const char *opt="")
void Vect (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &ax, const mglData &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=Min.z. The length and color of arrows are proportional to $\sqrt{ax^2+ay^2}$. The number of arrows depend on [meshnum], page 119. The apperance 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 169, [dew], page 168. See Section 2.8.1 [Vect sample], page 79, for sample code and picture.

```

vect udat wdat ['sch'=""] [MGL command]
vect xdat ydat zdat udat vdat wdat ['sch'=""] [MGL command]
void Vect (const mglData &ax, const mglData &ay, [Method on mglGraph]
           const mglData &az, const char *sch="", const char *opt="")
void Vect (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &z, const mglData &ax, const mglData &ay, const mglData &az,
           const char *sch="", const char *opt="")
void mgl_vect_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const char [C function]
                  *sch, const char *opt)
void mgl_vect_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, HCDT [C function]
                   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 $\sqrt{ax^2+ay^2+az^2}$.

```

dew udat vdat ['sch'=""] [MGL command]
dew xdat ydat udat vdat ['sch'=""] [MGL command]
void Dew (const mglData &ax, const mglData &ay, const [Method on mglGraph]
           char *sch="", const char *opt="")
void Dew (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &ax, const mglData &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, const char *sch, const char *opt) [C function]
```

The function draws dew-drops for plane vector field {ax, ay} depending parametrically on coordinates x, y at level $z = \text{Min.z}$. 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 119. See also [vect], page 168. See Section 2.8.5 [Dew sample], page 83, for sample code and picture.

```
flow udat vdat ['sch']='' [MGL command]
```

```
flow xdat ydat udat vdat ['sch']='' [MGL command]
```

```
void Flow (const mglData &ax, const mglData &ay, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void Flow (const mglData &x, const mglData &y, const mglData &ax, const mglData &ay, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void mgl_flow_2d (HMGL gr, HCDT ax, HCDT ay, const char *sch, const char *opt) [C function]
```

```
void mgl_flow_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay, const char *sch, const char *opt) [C function]
```

The function draws flow threads for the plane vector field {ax, ay} parametrically depending on coordinates x, y at level $z = \text{Min.z}$. 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 171, [vect], page 168, [tape], page 146, [barwidth], page 118. See Section 2.8.3 [Flow sample], page 81, 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 mglData &ax, const mglData &ay, const mglData &az, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void Flow (const mglData &x, const mglData &y, const mglData &z, const mglData &ax, const mglData &ay, const mglData &az, const char *sch="", const char *opt="") [Method on mglGraph]
```

```
void mgl_flow_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const char *sch, const char *opt) [C function]
```

```
void mgl_flow_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, HCDT ay, HCDT az, const char *sch, const char *opt) [C function]
```

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 mglData &ax, const      [Method on mglGraph]
            mglData &ay, const char *sch="", const char *opt="")
void FlowP (mglPoint p0, const mglData &x, const      [Method on mglGraph]
            mglData &y, const mglData &ax, const mglData &ay, const char
            *sch="", const char *opt="")
void mgl_flowp_2d (HMGL gr, float x0, float y0, float z0, HCDT      [C function]
                   ax, HCDT ay, const char *sch, const char *opt)
void mgl_flowp_xy (HMGL gr, float x0, float y0, float z0, HCDT x,   [C function]
                   HCDT y, HCDT ax, HCDT ay, const char *sch, const char *opt)
The same as first one ([flow], page 169) 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 mglData &ax, const      [Method on mglGraph]
            mglData &ay, const mglData &az, const char *sch="", const char
            *opt="")
void FlowP (mglPoint p0, const mglData &x, const      [Method on mglGraph]
            mglData &y, const mglData &z, const mglData &ax, const mglData &ay,
            const mglData &az, const char *sch="", const char *opt="")
void mgl_flowp_3d (HMGL gr, float x0, float y0, float z0, HCDT      [C function]
                   ax, HCDT ay, HCDT az, const char *sch, const char *opt)
void mgl_flowp_xyz (HMGL gr, float x0, float y0, float 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 mglData &phi, const char *sch="",           [Method on mglGraph]
            const char *opt="")
void Grad (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &phi, const char *sch="", const char *opt="")
void Grad (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const mglData &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). If *value*<0 then lines start only from edges of axis range. Warm color corresponds to normal flow (like minimum). Cold one corresponds to

inverse flow (like maximum). If string *sch* contain symbol ‘v’ then arrows are drawn on flow threads. See also [dens], page 156, [cont], page 156, [flow], page 169.

```
pipe udat vdat ['sch'=" r0=0.05] [MGL command]
pipe xdat ydat udat vdat ['sch'=" r0=0.05] [MGL command]
void Pipe (const mglData &ax, const mglData &ay, [Method on mglGraph]
           const char *sch="", float r0=0.05, const char *opt="")
void Pipe (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &ax, const mglData &ay, const char *sch="", float r0=0.05,
           const char *opt="")
void mgl_pipe_2d (HMGL gr, HCDT ax, HCDT ay, const char *sch, [C function]
                  float r0, const char *opt)
void mgl_pipe_xy (HMGL gr, HCDT x, HCDT y, HCDT ax, HCDT ay, [C function]
                  const char *sch, float 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 = Min.z. 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 169, [vect], page 168. See Section 2.8.4 [Pipe sample], page 82, 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 mglData &ax, const mglData &ay, [Method on mglGraph]
           const mglData &az, const char *sch="", float r0=0.05, const char
           *opt="")
void Pipe (const mglData &x, const mglData &y, const [Method on mglGraph]
           mglData &z, const mglData &ax, const mglData &ay, const mglData &az,
           const char *sch="", float r0=0.05, const char *opt="")
void mgl_pipe_3d (HMGL gr, HCDT ax, HCDT ay, HCDT az, const char [C function]
                  *sch, float r0, const char *opt)
void mgl_pipe_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT ax, HCDT
                   ay, HCDT az, const char *sch, float 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 $\sqrt{ax^2 + ay^2 + az^2}$.

4.15 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 109.

<code>densx dat ['sch'=' sval=nan]</code>	[MGL command]
<code>densy dat ['sch'=' sval=nan]</code>	[MGL command]
<code>densz dat ['sch'=' sval=nan]</code>	[MGL command]
<code>void DensX (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void DensY (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void DensZ (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_dens_x (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]
<code>void mgl_dens_y (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]
<code>void mgl_dens_z (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]

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. For example, code like

```
gr->DensX(c.Sum("x"), "BbcyrR", -1);
gr->DensY(c.Sum("y"), "", 1);
gr->DensZ(c.Sum("z"), "", -1);
```

will produce the following picture. See also [ContXYZ], page 172, [ContFXYZ], page 173, [dens], page 156, Section 4.17 [Data manipulation], page 179. See Section 2.7.8 [Dens projection sample], page 74, for sample code and picture.

<code>contx dat ['sch'=' sval=nan]</code>	[MGL command]
<code>conty dat ['sch'=' sval=nan]</code>	[MGL command]
<code>contz dat ['sch'=' sval=nan]</code>	[MGL command]
<code>void ContX (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void ContY (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void ContZ (const mglData &a, const char *stl="", float sVal=NAN, const char *opt="")</code>	[Method on <code>mglGraph</code>]
<code>void mgl_cont_x (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]
<code>void mgl_cont_y (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]
<code>void mgl_cont_z (HMGL gr, HCDT a, const char *stl, float sVal, const char *opt)</code>	[C function]

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. For example, code like

```
gr->ContX(c.Sum("x"), "BbcyrR", -1);
gr->ContY(c.Sum("y"), 0, 1);
```

```
gr->ContZ(c.Sum("z"),0,-1);
```

will produce the following picture. See also [ContFXYZ], page 173, [DensXYZ], page 171, [cont], page 156, Section 4.17 [Data manipulation], page 179. See Section 2.7.9 [Cont projection sample], page 75, for sample code and picture.

```
void ContX (const mglData &v, const mglData &a, const      [Method on mglGraph]
            char *stl="", float sVal=NAN, const char *opt="")
void ContY (const mglData &v, const mglData &a, const      [Method on mglGraph]
            char *stl="", float sVal=NAN, const char *opt="")
void ContZ (const mglData &v, const mglData &a, const      [Method on mglGraph]
            char *stl="", float sVal=NAN, const char *opt="")
void mgl_cont_x_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                     float sVal, const char *opt)
void mgl_cont_y_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                     float sVal, const char *opt)
void mgl_cont_z_val (HMGL gr, HCDT v, HCDT a, const char *stl,      [C function]
                     float sVal, const char *opt)
```

The same as previous with manual contour levels.

confx dat ['sch'=' sval=nan]	[MGL command]
confy dat ['sch'=' sval=nan]	[MGL command]
confz dat ['sch'=' sval=nan]	[MGL command]
void ContFX (const mglData &a, const char *stl="",	[Method on mglGraph]
float sVal=NAN, const char *opt="")	
void ContFY (const mglData &a, const char *stl="",	[Method on mglGraph]
float sVal=NAN, const char *opt="")	
void ContFZ (const mglData &a, const char *stl="",	[Method on mglGraph]
float sVal=NAN, const char *opt="")	
void mgl_contf_x (HMGL gr, HCDT a, const char *stl, float sVal,	[C function]
const char *opt)	
void mgl_contf_y (HMGL gr, HCDT a, const char *stl, float sVal,	[C function]
const char *opt)	
void mgl_contf_z (HMGL gr, HCDT a, const char *stl, float 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. For example, code like

```
gr->ContFX(c.Sum("x"),"BbcyrR",-1);
gr->ContFY(c.Sum("y"),0,1);
gr->ContFZ(c.Sum("z"),0,-1);
```

will produce the following picture. See also [ContFXYZ], page 173, [DensXYZ], page 171, [cont], page 156, Section 4.17 [Data manipulation], page 179. See Section 2.7.10 [ContF projection sample], page 76, for sample code and picture.

```

void ContFX (const mglData &v, const mglData &a,           [Method on mglGraph]
             const char *stl="", float sVal=NAN, const char *opt="")
void ContFY (const mglData &v, const mglData &a,           [Method on mglGraph]
             const char *stl="", float sVal=NAN, const char *opt="")
void ContFZ (const mglData &v, const mglData &a,           [Method on mglGraph]
             const char *stl="", float sVal=NAN, const char *opt="")
void mgl_contf_x_val (HMGL gr, HCDT v, HCDT a, const char *stl,   [C function]
                      float sVal, const char *opt)
void mgl_contf_y_val (HMGL gr, HCDT v, HCDT a, const char *stl,   [C function]
                      float sVal, const char *opt)
void mgl_contf_z_val (HMGL gr, HCDT v, HCDT a, const char *stl,   [C function]
                      float 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=Min.z where 'x' variable is changed in
xrange. You do not need to create the data arrays to plot it. See also [plot], page 145.

```

```

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 *eqY,   [C function]
                    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. See also
[plot], page 145.

```

```

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. See also [surf],
page 154.

```

```

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 *eqY,   [C function]
                    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. See also [surf], page 154.

```

```

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 mglData &id, const mglData &x, [Method on mglGraph]
              const mglData &y, const char *sch="", const char *opt="")
void TriPlot (const mglData &id, const mglData &x, [Method on mglGraph]
              const mglData &y, const mglData &z, const mglData &c, const char
              *sch="", const char *opt="")
void TriPlot (const mglData &id, const mglData &x, [Method on mglGraph]
              const mglData &y, const mglData &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 176, [crust], page 177, [quadplot], page 176. See Section 2.7.11 [TriPlot and QuadPlot], page 76, 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 mglData &id, const mglData &x, [Method on mglGraph]
               const mglData &y, const mglData &z, const mglData &c, const char
               *sch="", const char *opt="")
void TriCont (const mglData &id, const mglData &x, [Method on mglGraph]
               const mglData &y, const mglData &z, const char *sch="", const char
               *opt="")
void TriContV (const mglData &v, const mglData &id, [Method on mglGraph]
                const mglData &x, const mglData &y, const mglData &z, const mglData
                &c, const char *sch="", const char *opt="")
void TriContV (const mglData &v, const mglData &id, [Method on mglGraph]
                const mglData &x, const mglData &y, const mglData &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 y, [C function]
                        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 $z = \text{Min.z}$ 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 . 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 175, [cont], page 156.

```
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 mglData &id, const mglData &x,      [Method on mglGraph]
               const mglData &y, const char *sch="", const char *opt="")
void QuadPlot (const mglData &id, const mglData &x,      [Method on mglGraph]
               const mglData &y, const mglData &z, const mglData &c, const char
               *sch="", const char *opt="")
void QuadPlot (const mglData &id, const mglData &x,      [Method on mglGraph]
               const mglData &y, const mglData &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 z,      [C function]
                        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 175. See Section 2.7.11 [TriPlot and QuadPlot], page 76, for sample code and picture.

```
dots xdat ydat zdat ['sch'=""]                                [MGL command]
dots xdat ydat zdat adat ['sch'=""]                           [MGL command]
void Dots (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const char *sch="", const char *opt="")
void Dots (const mglData &x, const mglData &y, const      [Method on mglGraph]
            mglData &z, const mglData &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      [C function]
                  char *sch, const char *opt)
```

The function draws the arbitrary placed points $\{x[i], y[i], z[i]\}$. String sch sets the color scheme. If array a is specified then it define the transparency of dots. Arrays

x, y, z , a must have equal sizes. See also [crust], page 177, [mark], page 151, [plot], page 145. See Section 2.7.12 [Dots sample], page 77, for sample code and picture.

```
crust xdat ydat zdat ['sch='] [MGL command]
void Crust (const mglData &x, const mglData &y, const mglData &z, const char *sch="", const char *opt="")
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 176, [triplet], page 175.

4.16 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 4.2.8 [Error handling], page 121).

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 selected as maximal value of fit size and the value of $mglFitPnts$. Note, that this functions use GSL library and do something only if MathGL was compiled with GSL support. See Section 2.9.10 [Nonlinear fitting sample], page 96, 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 mglData &a, const mglData &s, [Method on mglGraph]
               const char *func, const char *var, const char *opt="")
mglData FitS (const mglData &a, const mglData &s, [Method on mglGraph]
               const char *func, const char *var, mglData &ini, const char *opt="")
mglData FitS (const mglData &x, const mglData &a, [Method on mglGraph]
               const mglData &s, const char *func, const char *var, const char
               *opt="")
mglData FitS (const mglData &x, const mglData &a, [Method on mglGraph]
               const mglData &s, const char *func, const char *var, mglData &ini,
               const char *opt="")
mglData FitS (const mglData &x, const mglData &y, [Method on mglGraph]
               const mglData &a, const mglData &s, const char *func, const char
               *var, const char *opt="")
```

```

mglData FitS (const mglData &x, const mglData &y,           [Method on mglGraph]
             const mglData &a, const mglData &s, const char *func, const char
             *var, mglData &ini, const char *opt="")
mglData FitS (const mglData &x, const mglData &y,           [Method on mglGraph]
             const mglData &z, const mglData &a, const mglData &s, const char
             *func, const char *var, const char *opt="")
mglData FitS (const mglData &x, const mglData &y,           [Method on mglGraph]
             const mglData &z, const mglData &a, const mglData &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, const
                  char *var, HMDT ini, const char *opt) [C function]
HMDT mgl_fit_xys (HMGL gr, HCDT x, HCDT a, HCDT s, const char
                  *func, const char *var, HMDT ini, const char *opt) [C function]
HMDT mgl_fit_xyzs (HMGL gr, HCDT x, HCDT y, HCDT a, HCDT s, const
                  char *func, const char *var, HMDT ini, const char *opt) [C function]
HMDT mgl_fit_xyzas (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, 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 sdat 'func' 'var' [ini=0] [MGL command]
fit res xdat adat sdat 'func' 'var' [ini=0] [MGL command]
fit res xdat ydat adat sdat 'func' 'var' [ini=0] [MGL command]
fit res xdat ydat zdat adat sdat 'func' 'var' [ini=0] [MGL command]
mglData Fit (const mglData &a, const char *func,
              const char *var, const char *opt="") [Method on mglGraph]
mglData Fit (const mglData &a, const char *func,           [Method on mglGraph]
              const char *var, mglData &ini, const char *opt="")
mglData Fit (const mglData &x, const mglData &a,           [Method on mglGraph]
              const char *func, const char *var, const char *opt="")
mglData Fit (const mglData &x, const mglData &a,           [Method on mglGraph]
              const char *func, const char *var, mglData &ini, const char *opt="")
mglData Fit (const mglData &x, const mglData &y,           [Method on mglGraph]
              const mglData &a, const char *func, const char *var, mglData &ini,
              const char *opt="")
mglData Fit (const mglData &x, const mglData &y,           [Method on mglGraph]
              const mglData &z, const mglData &a, const char *func, const char
              *var, const char *opt="")
mglData Fit (const mglData &x, const mglData &y,           [Method on mglGraph]
              const mglData &z, const mglData &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
                  *var, HMDT ini, const char *opt) [C function]

```

```

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, const   [C function]
                  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 mglData &a, const char *func,                 [Method on mglGraph]
               const char *var, const char *opt="")
mglData Fit2 (mglData &fit, const mglData &a, const             [Method on mglGraph]
               char *func, const char *var, mglData &ini, const char *opt="")
mglData Fit3 (mglData &fit, const mglData &a, const             [Method on mglGraph]
               char *func, const char *var, const char *opt="")
mglData Fit3 (mglData &fit, const mglData &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 interval [Min, Max].
```

putsfit x y ['pre'=' ' 'fnt'=' ' size=-1] [MGL command]
void PutsFit (mglPoint p, const char *prefix="", [Method on mglGraph]
 const char *font="", float size=-1)
void mgl_puts_fit (HMGL gr, float x, float y, float z, const [C function]
 char *prefix, const char *font, float 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 4.7 [Text printing], page 139.

const char *GetFit () [Method on mglGraph]
const char * mgl_get_fit (HMGL gr) [C function]
Get last fitted formula with found coefficients (as numbers).

4.17 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 mglData &x, const mglData &a,    [Method on mglGraph]
              const char *opt="")
mglData Hist (const mglData &x, const mglData &y,    [Method on mglGraph]
              const mglData &a, const char *opt="")
mglData Hist (const mglData &x, const mglData &y,    [Method on mglGraph]
              const mglData &z, const mglData &a, const char *opt="")

```

```
HMDT mgl_hist_x (HMGL gr, HCDT x, HCDT a, const char *opt) [C function]
HMDT mgl_hist_xy (HMGL gr, HCDT x, HCDT y, HCDT a, const char
    *opt) [C function]
HMDT mgl_hist_xyz (HMGL gr, HCDT x, HCDT y, HCDT z, HCDT a, const
    char *opt) [C function]
```

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 selected as maximal value of *res* size and the value of *mglFitPnts*.

```
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
    *opt="")
void Fill (mglData &u, const char *eq, const mglData
    &v, const char *opt="")
void Fill (mglData &u, const char *eq, const mglData
    &v, const mglData &w, const char *opt="")
void mgl_data_fill_eq (HMGL gr, HMDT u, const char *eq, HCDTv,
    HCDTw, const char *opt) [C function]
```

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. 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).

```
pde RES 'ham' ini_re ini_im [dz=0.1 k0=100] [MGL command]
mglData PDE (const char *ham, const mglData &ini_re, [Method on mglGraph]
    const mglData &ini_im, float dz=0.1, float k0=100, const char
    *opt="")
HMDT mgl_pde_solve (HMGL gr, const char *ham, HCDT ini_re, HCDT
    ini_im, float dz, float k0, const char *opt) [C function]
```

Solves equation $\frac{du}{dz} = i*k0*ham(p,q,x,y,z,|u|)[u]$, where $p=-i/k0*d/dx$, $q=-i/k0*d/dy$ are pseudo-differential operators. Parameters *ini_re*, *ini_im* specify real and imaginary part of initial field distribution. Parameters *Min*, *Max* set the bounding box for the solution. Note, that really this ranges are increased by factor 3/2 for purpose of reducing reflection from boundaries. Parameter *dz* set the step along evolutionary coordinate *z*. At this moment, simplified form of function *ham* is supported – all “mixed” terms (like ‘*x***p*->*x***d*/*dx*) are excluded. For example, in 2D case this function is effectively *ham* = *f*(*p*, *z*) + *g*(*x*, *z*, *u*). However commutable combinations (like ‘*x***q*->*x***d*/*dy*) are allowed. Here variable ‘*u*’ is used for field amplitude $|u|$. This allow one solve nonlinear problems – for example, for nonlinear Shrodinger equation you may set *ham* = “*p*² + *q*² - *u*²”. You may specify imaginary part for wave absorption, like *ham* = “*p*² + *i***x**(*x*>0)”, but only

if dependence on variable ‘i’ is linear (i.e. $ham = hre + i * him$). See [Section 2.9.11 \[PDE solving hints\]](#), page [97](#), for sample code and picture.

4.18 IDTF functions

These functions provide IDTF specific features. In all other cases they do nothing.

`void VertexColor (bool enable)` [Method on `mglGraph`]
Enables smooth color change.

`void Compression (bool enable)` [Method on `mglGraph`]
Gives smaller files, but quality degrades.

5 Widget classes

There are set of “window” classes for making a window with MathGL graphics: `mglWindow` and `mglGLUT` for whole window, `F1_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.

For callback drawing you can use: NULL pointer if you’ll update plot manually, global function of type `int draw(HMGL gr, void *p)` or `int draw(mglGraph *gr)`, or instance of class derived from `mglDraw` class. This class is defined in `#include <mgl2/window.h>` and have only 2 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.

5.1 `mglWindow` class

This class is derived from `mglGraph` class (see [Chapter 4 \[MathGL core\]](#), page 116). It provide methods for handling window with MathGL graphics.

<code>mglWindow (const char *title="MathGL")</code>	[Constructor on <code>mglWindow</code>]
<code>mglWindow (int (*draw)(HMGL gr, void *p), const</code>	[Constructor on <code>mglWindow</code>]
<code>char *title="MathGL", void *par=NULL, int kind=0, void (*reload)(HMGL</code>	
<code>gr, void *p)=0)</code>	
<code>mglWindow (int (*draw)(mglGraph *gr), const char</code>	[Constructor on <code>mglWindow</code>]
<code>*title="MathGL", int kind=0)</code>	
<code>mglWindow (mglDraw *draw, const char</code>	[Constructor on <code>mglWindow</code>]
<code>*title="MathGL", int kind=0)</code>	
<code>HMGL mgl_create_graph_qt (int (*draw)(HMGL gr, void *p), const</code>	[C function]
<code>char *title, void *par, void (*reload)(HMGL gr, void *p))</code>	
<code>HMGL mgl_create_graph_fltk (int (*draw)(HMGL gr, void *p),</code>	[C function]
<code>const char *title, void *par, void (*reload)(HMGL gr, void *p))</code>	
<code>HMGL mgl_create_graph_glut (int (*draw)(HMGL gr, void *p),</code>	[C function]
<code>const char *title, void *par, void (*reload)(HMGL gr, void *p))</code>	

Creates a 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. Parameter *kind* may have following values: '0' – use FLTK window, '1' – use Qt window.

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.

```

int RunThr ()                                     [Method on mglWindow]
int mgl_fltk_thr ()                         [C function]
    Run main loop for event handling in separate thread. Note, right now it work for
    FLTK windows only.

int Run ()                                       [Method on mglWindow]
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 SetClickFunc (void (*func)(HMGL gr, void *p))      [Method on mglWindow]
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 ToggleAlpha ()                                [Method on mglWindow]
int 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 mglWindow]
int 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 mglWindow]
int 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 mglWindow]
int 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 mglWindow]
int mgl_wnd_toggle_no (HMGL gr)               [C function]
    Switch off all zooming and rotation and restore initial state.
```

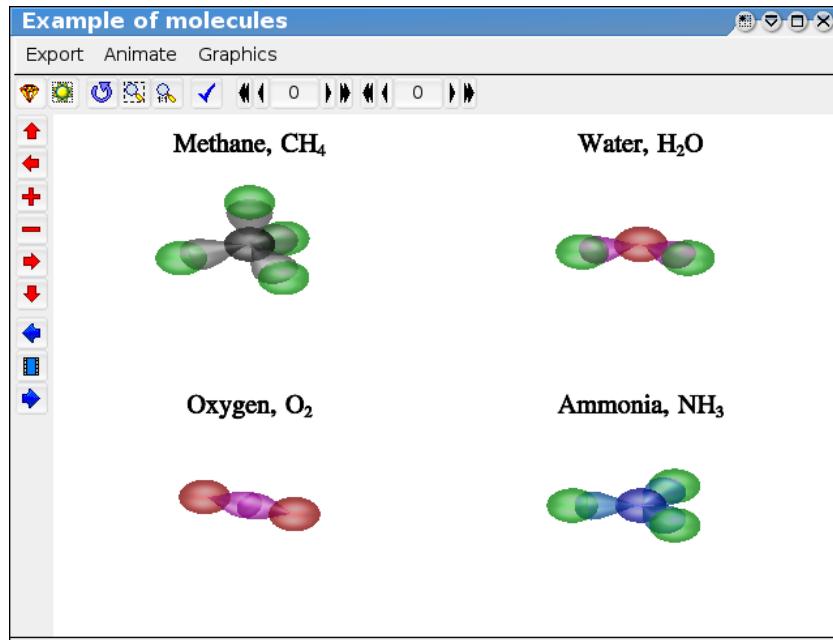
```

void Update ()                                  [Method on mglWindow]
int 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.
```

<code>void Reload ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_reload (HMGL gr)</code>	[C function]
Reload user data and update picture. This function also update number of frames which drawing function can create.	
<code>void Adjust ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_adjust (HMGL gr)</code>	[C function]
Adjust size of bitmap to window size.	
<code>void NextFrame ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_next_frame (HMGL gr)</code>	[C function]
Show next frame if one.	
<code>void PrevFrame ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_prev_frame (HMGL gr)</code>	[C function]
Show previous frame if one.	
<code>void Animation ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_animation (HMGL gr)</code>	[C function]
Run/stop slideshow (animation) of frames.	
<code>void SetDelay (float dt)</code>	[Method on <code>mglWindow</code>]
<code>int mgl_wnd_set_delay (HMGL gr, int dt)</code>	[C function]
Sets delay for animation in seconds. Default value is 1 sec.	
<code>void Setup (bool clfupd=true, bool showpos=false)</code>	[Method on <code>mglWindow</code>]
<code>int mgl_setup_window (HMGL gr, bool clfupd, bool showpos)</code>	[C function]
Enable/disable flags for:	
• clearing plot before Update();	
• showing the last mouse click position in the widget.	
<code>mglPoint LastMousePos ()</code>	[Method on <code>mglWindow</code>]
<code>int mgl_get_last_mouse_pos (HMGL gr, float *x, float *y, float *z)</code>	[C function]
Gets last position of mouse click.	

5.2 Fl_MathGL class

Class is FLTK widget which display MathGL graphics (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 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 (float t, float 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 (float X1, float Y1, float X2, float Y2) [Method on Fl_MathGL]
Set zoom in/out region
```

```
void get_zoom (float *X1, float *Y1, float *X2, float *Y2) [Method on Fl_MathGL]
Get zoom in/out region
```

```
void set_popup (const Fl_Menu_Item *pmenu, Fl_Widget *w, void *v) [Method on Fl_MathGL]
Set popup menu pointer
```

```

void set_graph (mglCanvas *gr) [Method on Fl_MathGL]
void set_graph (mglGraph *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 setGraph() call.

HMGL get_graph () [Method on Fl_MathGL]
    Get pointer to grapher.

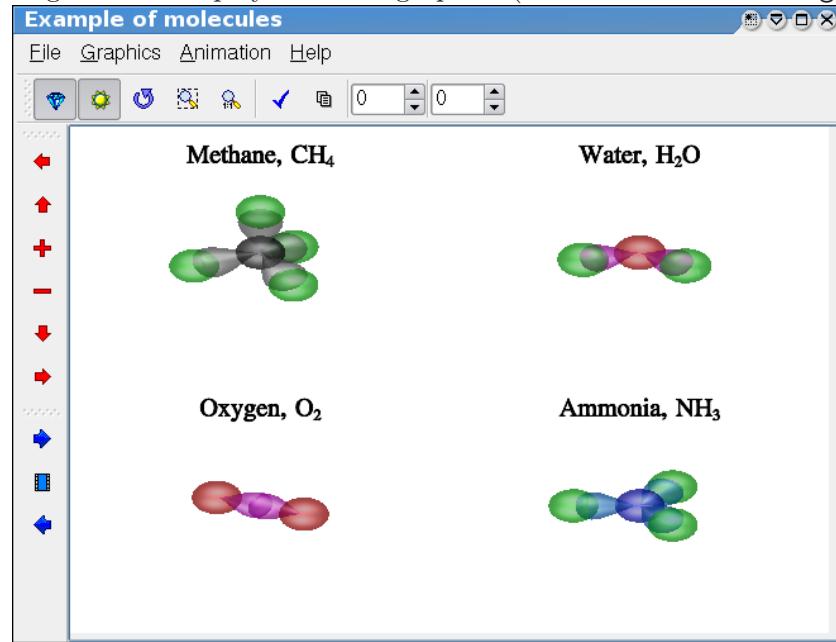
Fl_Valuator * tet_val [Fl_MathGL option of Fl_MathGL]
    Pointer to external tet-angle validator.

Fl_Valuator * phi_val [Fl_MathGL option of Fl_MathGL]
    Pointer to external phi-angle validator.

```

5.3 QMathGL class

Class is Qt widget which display MathGL graphics (defined in `#include <mgl2/qt.h>`).



```

void setDraw (mglDraw *dr) [Method on QMathGL]
    Sets drawing functions from a class inherited from mglDraw.

void setDraw (int (*draw)(mglBase *gr, void *p), void *par=NULL) [Method on QMathGL]
void setDraw (int (*draw)(mglGraph *gr, void *p)) [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.

```

<code>void setGraph (mglCanvas *gr)</code>	[Method on QMathGL]
<code>void setGraph (mglGraph *gr)</code>	[Method on QMathGL]
Set pointer to external grapher (instead of built-in <code>mglCanvas</code>). Note that QMathGL will automatically delete this object at destruction or at new <code>setGraph()</code> call.	
<code>HMGL getGraph ()</code>	[Method on QMathGL]
Get pointer to grapher.	
<code>void setPopup (QMenu *p)</code>	[Method on QMathGL]
Set popup menu pointer.	
<code>void setSize (int w, int h)</code>	[Method on QMathGL]
Set widget/picture sizes	
<code>double getRatio ()</code>	[Method on QMathGL]
Return aspect ratio of the picture.	
<code>int getPer ()</code>	[Method on QMathGL]
Get perspective value in percents.	
<code>int getPhi ()</code>	[Method on QMathGL]
Get Phi-angle value in degrees.	
<code>int getTet ()</code>	[Method on QMathGL]
Get Theta-angle value in degrees.	
<code>bool getAlpha ()</code>	[Method on QMathGL]
Get transparency state.	
<code>bool getLight ()</code>	[Method on QMathGL]
Get lightning state.	
<code>bool getZoom ()</code>	[Method on QMathGL]
Get mouse zooming state.	
<code>bool getRotate ()</code>	[Method on QMathGL]
Get mouse rotation state.	
<code>void refresh ()</code>	[Slot on QMathGL]
Redraw saved bitmap without executing drawing function.	
<code>void update ()</code>	[Slot on QMathGL]
Update picture by executing drawing function.	
<code>void copy ()</code>	[Slot on QMathGL]
Copy graphics to clipboard.	
<code>void copyClickCoor ()</code>	[Slot on QMathGL]
Copy coordinates of click (as text).	
<code>void print ()</code>	[Slot on QMathGL]
Print current picture.	

<code>void stop ()</code>	[Slot on QMathGL]
Send signal to stop drawing.	
<code>void adjust ()</code>	[Slot on QMathGL]
Adjust image size to fit whole widget.	
<code>void nextSlide ()</code>	[Slot on QMathGL]
Show next slide.	
<code>void prevSlide ()</code>	[Slot on QMathGL]
Show previous slide.	
<code>void animation (bool st=true)</code>	[Slot on QMathGL]
Start/stop animation.	
<code>void setPer (int val)</code>	[Slot on QMathGL]
Set perspective value.	
<code>void setPhi (int val)</code>	[Slot on QMathGL]
Set Phi-angle value.	
<code>void setTet (int val)</code>	[Slot on QMathGL]
Set Theta-angle value.	
<code>void setAlpha (bool val)</code>	[Slot on QMathGL]
Switch on/off transparency.	
<code>void setLight (bool val)</code>	[Slot on QMathGL]
Switch on/off lightning.	
<code>void setGrid (bool val)</code>	[Slot on QMathGL]
Switch on/off drawing of grid for absolute coordinates.	
<code>void setZoom (bool val)</code>	[Slot on QMathGL]
Switch on/off mouse zooming.	
<code>void setRotate (bool val)</code>	[Slot on QMathGL]
Switch on/off mouse rotation.	
<code>void zoomIn ()</code>	[Slot on QMathGL]
Zoom in graphics.	
<code>void zoomOut ()</code>	[Slot on QMathGL]
Zoom out graphics.	
<code>void shiftLeft ()</code>	[Slot on QMathGL]
Shift graphics to left direction.	
<code>void shiftRight ()</code>	[Slot on QMathGL]
Shift graphics to right direction.	
<code>void shiftUp ()</code>	[Slot on QMathGL]
Shift graphics to up direction.	

void shiftDown ()	[Slot on QMathGL]
Shift graphics to down direction.	
void restore ()	[Slot on QMathGL]
Restore zoom and rotation to default values.	
void exportPNG (QString fname="")	[Slot on QMathGL]
Export current picture to PNG file.	
void exportPNGs (QString fname="")	[Slot on QMathGL]
Export current picture to PNG file (no transparency).	
void exportJPG (QString fname="")	[Slot on QMathGL]
Export current picture to JPEG file.	
void exportBPS (QString fname="")	[Slot on QMathGL]
Export current picture to bitmap EPS file.	
void exportEPS (QString fname="")	[Slot on QMathGL]
Export current picture to vector EPS file.	
void exportSVG (QString fname="")	[Slot on QMathGL]
Export current picture to SVG file.	
void exportGIF (QString fname="")	[Slot on QMathGL]
Export current picture to GIF file.	
void exportTEX (QString fname="")	[Slot on QMathGL]
Export current picture to LaTeX/Tikz file.	
void exportTGA (QString fname="")	[Slot on QMathGL]
Export current picture to TGA file.	
void exportXYZ (QString fname="")	[Slot on QMathGL]
Export current picture to XYZ/XYZL/XYZF file.	
void exportOBJ (QString fname="")	[Slot on QMathGL]
Export current picture to OBJ/MTL file.	
void exportSTL (QString fname="")	[Slot on QMathGL]
Export current picture to STL file.	
void exportOFF (QString fname="")	[Slot on QMathGL]
Export current picture to OFF file.	
void exportIDTF (QString fname="")	[Slot on QMathGL]
Export current picture to IDTF file.	
void setMGLFont (QString path)	[Slot on QMathGL]
Restore (path="") or load font for graphics.	
void about ()	[Slot on QMathGL]
Show about information.	

<code>void aboutQt ()</code>	[Slot on QMathGL]
Show information about Qt version.	
<code>void phiChanged (int val)</code>	[Signal on QMathGL]
Phi angle changed (by mouse or by toolbar).	
<code>void tetChanged (int val)</code>	[Signal on QMathGL]
Tet angle changed (by mouse or by toolbar).	
<code>void perChanged (int val)</code>	[Signal on QMathGL]
Perspective changed (by mouse or by toolbar).	
<code>void alphaChanged (bool val)</code>	[Signal on QMathGL]
Transparency changed (by toolbar).	
<code>void lightChanged (bool val)</code>	[Signal on QMathGL]
Lighting changed (by toolbar).	
<code>void gridChanged (bool val)</code>	[Signal on QMathGL]
Grid drawing changed (by toolbar).	
<code>void zoomChanged (bool val)</code>	[Signal on QMathGL]
Zooming changed (by toolbar).	
<code>void rotateChanged (bool val)</code>	[Signal on QMathGL]
Rotation changed (by toolbar).	
<code>void mouseClicked (float x, float y, float z)</code>	[Signal on QMathGL]
Mouse click take place at position {x,y,z}.	
<code>void frameChanged (int val)</code>	[Signal on QMathGL]
Need another frame to show.	
<code>void showWarn (QString warn)</code>	[Signal on QMathGL]
Need to show warning.	
<code>void posChanged (QString pos)</code>	[Signal on QMathGL]
Position of mouse click is changed.	
<code>void objChanged (int id)</code>	[Signal on QMathGL]
Object id is changed (due to mouse click).	
<code>void refreshData ()</code>	[Signal on QMathGL]
Data can be changed (drawing is finished).	
<code>QString appName</code>	[QMathGL option of QMathGL]
Application name for message boxes.	
<code>bool autoResize</code>	[QMathGL option of QMathGL]
Allow auto resizing (default is false).	

6 Data processing

This chapter describe class `mglData` for working with data array. This class is defined in `#include <mgl2/data.h>`. The class has functions for easy and safe allocation, resizing, loading and saving, modifying of data arrays. Also it can numerically differentiate and integrate data, interpolate, fill data by formula and so on. Class supports data with dimensions up to 3 (like function of 3 variables – x,y,z). The internal representation of numbers is float. Float type was chosen because it has smaller size in memory and usually it has enough precision in plotting purposes. You can change it by selecting option `--enable-double` at the library configuring (see [Section 1.3 \[Installation\], page 2](#)). Data arrays are denoted by Small Caps (like DAT) if it can be (re-)created by MGL commands.

6.1 Public variables

<code>float * a</code>	[Variable of <code>mglData</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>int nx</code>	[Variable of <code>mglData</code>]
Number of points in 1st dimensions ('x' dimension).	
<code>int ny</code>	[Variable of <code>mglData</code>]
Number of points in 2nd dimensions ('y' dimension).	
<code>int nz</code>	[Variable of <code>mglData</code>]
Number of points in 3d dimensions ('z' dimension).	
<code>char * id</code>	[Variable of <code>mglData</code>]
Column (or slice if nz>1) names – one character per column.	
<code>bool link</code>	[Variable of <code>mglData</code>]
Flag to use external data, i.e. don't delete it.	
<code>float GetVal (long i)</code>	[Method on <code>mglData</code>]
<code>void SetVal (float val, long i)</code>	[Method on <code>mglData</code>]
Gets or sets the value in by "flat" index <i>i</i> without border checking. Index <i>i</i> should be in range [0, nx*ny*nz-1].	
<code>long GetNx ()</code>	[Method on <code>mglData</code>]
<code>long GetNy ()</code>	[Method on <code>mglData</code>]
<code>long GetNz ()</code>	[Method on <code>mglData</code>]
<code>long mgl_data_get_nx (HCDT dat)</code>	[C function]
<code>long mgl_data_get_ny (HCDT dat)</code>	[C function]
<code>long mgl_data_get_nz (HCDT dat)</code>	[C function]
Gets the x-, y-, z-size of the data.	
<code>float mgl_data_get_value (HCDT dat, int i, int j, int k)</code>	[C function]
<code>float * mgl_data_value (HMMDT dat, int i, int j, int k)</code>	[C function]
<code>void mgl_data_set_value (HMMDT dat, float v, int i, int j, int k)</code>	[C function]
Gets or sets the value in specified cell of the data with border checking.	

```
const float * mgl_data_data (HCDT dat) [C function]
    Returns pointer to internal data array.
```

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]
HMDT mgl_create_data () [C function]
HMDT mgl_create_data_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 195 function.

copy DAT dat2 ['eq'=''] [MGL command]
copy DAT val [MGL command]
mglData (const mglData &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]
    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 corre-
    sponding formula similarly to \[fill\], page 195 function.
```

```
read DAT 'fname' [MGL command]
mglData (const char *fname) [Constructor on mglData]
HMDT mgl_create_data_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]
    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 mgl_data_create (HMDT 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 mgl_data_rearrange (HMDT 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 will be 1.

transpose dat ['dim']='yxz'] [MGL command]
void Transpose (const char *dim="yx") [Method on mglData]
void mgl_data_transpose (const char *dim) [C function]
    Transposes (shift order of) dimensions of the data. New order of dimensions is spec-
    ified 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 mgl_data_extend (HMDT 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 si-
    multaneously for 1d data by using parameter n2. Data to new slices is copy from
    existed one. For example, for  $n1>0$  new array will be  $a_{ij}^{new} = a_i^{old}$  where  $j=0...n1$ .
    Correspondingly, for  $n1<0$  new array will be  $a_{ij}^{new} = a_j^{old}$  where  $i=0...|n1|$ .

squeeze dat rx [ry=1 rz=1 sm=false] [MGL command]
void Squeeze (int rx, int ry=1, int rz=1, bool [Method on mglData]
    smooth=false)
void mgl_data_squeeze (HMDT 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 mgl_data_crop (HMDT 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 mgl_data_insert (HMDT 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 mgl_data_delete (HMDT 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 (long 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!
```

6.4 Data filling

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

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

```
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 float *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]
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_float2 (HMDT dat, const float **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 float 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_float3 (HMDT dat, const float ***A, int N1, int N2) [C function]
void mgl_data_set_double3 (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, N3, i.e. from array defined as float a[N1][N2][N3];.
```

```

void Set (gsl_vector *v) [Method on mglData]
void mgl_data_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 mgl_data_set_matrix (HMDT dat, gsl_matrix *m) [C function]
    Allocates memory and copies the data from the gsl_matrix * structure.

void Set (const mglData &from) [Method on mglData]
void Set (HCDT from) [Method on mglData]
void mgl_data_set (HMDT dat, HCDT from) [C function]
    Copies the data from mglData (or mglDataA) instance from.

void Set (const std::vector<int> &d) [Method on mglData]
void Set (const std::vector<float> &d) [Method on mglData]
void Set (const std::vector<double> &d) [Method on mglData]
    Allocates memory and copies the data from the std::vector<T> array.

void Set (const char *str, int NX, int NY=1, int NZ=1) [Method on mglData]
void mgl_data_set_values (const char *str, int NX, int NY, [C function]
                           int NZ)
    Allocates memory and scanf the data from the string.

void Link (const mglData &from) [Method on mglData]
void Link (const float *A, int NX, int NY=1, int [Method on mglData]
           NZ=1)
void mgl_data_link (HMDT dat, const float *A, int NX, int NY, [C function]
                     int NZ)
    Links external data array, i.e. don't delete it at exit.

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

fill dat v1 v2 ['dir='x] [MGL command]
void Fill (float v1, float v2, char dir='x') [Method on mglData]
void mgl_data_fill (HMDT dat, float v1, float v2, char dir) [C function]
    Equidistantly fills the data values to range [v1, v2] in direction dir={‘x’,‘y’,‘z’}.

fill dat 'eq' [MGL command]
fill dat 'eq' vdat [MGL command]
fill dat 'eq' vdat wdat [MGL command]
void Fill (HMGL gr, const char *eq, const char *opt="") [Method on mglData]
void Fill (HMGL gr, const char *eq, const mglData &vdat,
           const char *opt="")
void Fill (HMGL gr, const char *eq, const mglData &vdat, [Method on mglData]
           const mglData &wdat, const char *opt="")

```

```
void mgl_data_fill_eq (HMGL gr, HMDT dat, const char *eq, const [C function]
                      HMDT *vdat, const HMDT *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 mglData &v) [Method on mglData]
```

```
void Modify (const char *eq, const mglData &v, const [Method on mglData]
             mglData &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)
```

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 $\geq dim$.

```
fillsample dat 'how' [MGL command]
```

```
void FillSample (const char *how) [Method on mglData]
```

```
void mgl_data_fill_sample (HMDT a, const char *how) [mglData]
```

Fills data by ‘x’ or ‘k’ samples for Hankel (‘h’) or Fourier (‘f’) transform.

```
put dat val [i=: j=: k=:] [MGL command]
```

```
void Put (float val, int i=-1, int j=-1, int k=-1) [Method on mglData]
```

```
void mgl_data_put_val (HMDT a, float val, int i, int j, int k) [mglData]
```

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 $i=0\dots(nx-1)$, $j=0\dots(nz-1)$.

```
put dat vdat [i=: j=: k=:] [MGL command]
```

```
void Put (const mglData &v, int i=-1, int j=-1, int [Method on mglData]
          k=-1)
```

```
void mgl_data_put_dat (HMDT a, HCDT v, int i, int j, int k) [mglData]
```

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 $i=0\dots(nx-1)$, $j=0\dots(nz-1)$ and condition $v.nx>=nx$ is true.

```
idset dat 'ids' [MGL command]
```

```
void SetColumnId (const char *ids) [Method on mglData]
```

```
void mgl_data_set_id (const char *ids) [mglData]
```

Sets the symbol *ids* for data columns. The string should contain one symbol ‘a’…‘z’ per column. These ids are used in [\[column\]](#), [page 199](#) function.

6.5 File I/O

read DAT 'fname' [MGL command]
void Read (const char *fname) [Method on `mglData`]
void mgl_data_read (HMDT 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]
void Read (const char *fname, int mx, int my=1, int mz=1) [Method on `mglData`]
void mgl_data_read_dim (HMDT dat, const char *fname, int mx, int my, int mz) [C function]
 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]
void ReadMat (const char *fname, int dim=2) [Method on `mglData`]
void mgl_data_read_mat (HMDT 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, float from, float to, float step=1.f, bool as_slice=false) [Method on `mglData`]
 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`]
 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*.

save dat 'fname' [MGL command]
void Save (const char *fname, int ns=-1) const [Method on `mglData`]
void mgl_data_save (HCDT dat, const char *fname, int ns) [C function]
 Saves the whole data array (for *ns=-1*) or only *ns*-th slice to text file.

readhdf DAT 'fname' 'dname' [MGL command]
void ReadHDF (const char *fname, const char *dname) [Method on `mglData`]
void mgl_data_read_hdf (HMDT 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' 'lname'                                [MGL command]
void SaveHDF (const char *fname, const char *dname,      [Method on mglData]
              bool rewrite=false) const
void mgl_data_save_hdf (HCDT dat, const char *fname, const      [C function]
                         char *dname, int rewrite)
  Saves data array named dname to HDF5 file. This function does nothing if HDF5
  was disabled during library compilation.

datas 'fname'                                              [MGL command]
void DatasHDF (const char *fname, char *buf, long size)    [Method on mglData]
  const
void 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,        [Method on mglData]
             float v1=0, float v2=1)
void mgl_data_import (HMDT dat, const char *fname, const char      [C function]
                      *scheme, float v1, float v2)
  Reads data from bitmap file (now support only PNG format). The RGB values of
  bitmap pixels are transformed to float values in range [v1, v2] using color scheme
  scheme (see Section 3.4 \[Color scheme\], page 109).

export dat 'fname' 'sch' [v1=0 v2=0]                  [MGL command]
void Export (const char *fname, const char *scheme,        [Method on mglData]
             float v1=0, float v2=0, int ns=-1) const
void mgl_data_export (HMDT dat, const char *fname, const char      [C function]
                      *scheme, float v1, float v2, int ns) const
  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 109). 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=-1 zz=-1]                [MGL command]
mglData SubData (float xx, float yy=-1, float zz=-1)    [Method on mglData]
  const
HMDT mgl_data_subdata (HCDT dat, float xx, float yy, float 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).

```

```

subdata RES dat xdat [ydat=: zdat=:]                                [MGL command]
mglData SubData (const mglData &xx, const mglData
    &yy, const mglData &zz) const                                     [Method on mglData]

HMDT mgl_data_subdata_ext (HCDT dat, HCDT xx, HCDT yy, HCDT zz)   [C function]
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).

column RES dat 'eq'                                                 [MGL command]
mglData Column (const char *eq) const                               [Method on mglData]
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 196
function or read from files. In MGL version this command usually is used as inline
one dat('eq').

resize RES dat mx [my=1 mz=1]                                         [MGL command]
mglData Resize (int mx, int my=1, int mz=1, float                  [Method on mglData]
    x1=0, float x2=1, float y1=0, float y2=1, float z1=0, float 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]
    float x1, float x2, float y1, float y2, float z1, float 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].

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 mglData &idat, bool                         [Method on mglData]
    norm=true) const
mglData Evaluate (const mglData &idat, const mglData                  [Method on mglData]
    &jdat, bool norm=true) const
mglData Evaluate (const mglData &idat, const mglData                  [Method on mglData]
    &jdat, const mglData &kdat, bool norm=true) const
HMDT mgl_data_evaluate (HCDT dat, HCDT idat, HCDT jdat, HCDT
    kdat, int norm)                                                 [C function]
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.

hist RES dat num v1 v2 [nsub=0]                                       [MGL command]
hist RES dat wdat num v1 v2 [nsub=0]                                     [MGL command]

```

```
mglData Hist (int n, float v1=0, float v2=1, int nsub=0) const [Method on mglData]
mglData Hist (const mglData &w, int n, float v1=0, float v2=1, int nsub=0) const [Method on mglData]
HMDT mgl_data_hist (HCDT dat, int n, float v1, float v2, int nsub) [C function]
HMDT mgl_data_hist_w (HCDT dat, HCDT w, int n, float v1, float 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). See also Section 4.17 \[Data manipulation\], page 179
```

```
momentum RES dat 'how' ['dir'='z'] [MGL command]
mglData Momentum (char dir, const char *how) const [Method on mglData]
HMDT mgl_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 var='z' and so on. Coordinates 'x', 'y', 'z' are data indexes normalized in range [0,1].
```

```
sum RES dat 'dir' [MGL command]
mglData Sum (const char *dir) const [Method on mglData]
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).
```

```
max RES dat 'dir' [MGL command]
mglData Max (const char *dir) const [Method on mglData]
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).
```

```
min RES dat 'dir' [MGL command]
mglData Min (const char *dir) const [Method on mglData]
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).
```

```
combine RES adat bdat [MGL command]
mglData Combine (const mglData &a) const [Method on mglData]
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).
```

```
trace RES dat [MGL command]
mglData Trace () const [Method on mglData]
Gets array of diagonal elements a[i,i] (for 2D case) or a[i,i,i] (for 3D case) where i=0...nx-1. Function return copy of itself for 1D case. Data array must have dimensions ny,nz >= nx or ny,nz = 1.
```

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.

<code>cumsum dat 'dir'</code>	[MGL command]
<code>void CumSum (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_cumsum (HMDT dat, const char *dir)</code>	[C function]

Cumulative summation of the data in given direction or directions.

<code>integrate dat 'dir'</code>	[MGL command]
<code>void Integral (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_integral (HMDT dat, const char *dir)</code>	[C function]

Integrates (like cumulative summation) the data in given direction or directions.

<code>diff dat 'dir'</code>	[MGL command]
<code>void Diff (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_diff (HMDT dat, const char *dir)</code>	[C function]

Differentiates the data in given direction or directions.

<code>diff dat xdat ydat [zdat=0]</code>	[MGL command]
<code>void Diff (const mglData &x, const mglData &y)</code>	[Method on <code>mglData</code>]
<code>void Diff (const mglData &x, const mglData &y, const mglData &z)</code>	[Method on <code>mglData</code>]

<code>void mgl_data_diff_par (HMDT dat, HCDT x, HCDTy, HCDTz)</code>	[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)`.

<code>diff2 dat 'dir'</code>	[MGL command]
<code>void Diff2 (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_diff2 (const char *dir)</code>	[C function]

Double-differentiates (like Laplace operator) the data in given direction.

<code>sinfft dat 'dir'</code>	[MGL command]
<code>void SinFFT (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_sinfft (HMDT dat, const char *dir)</code>	[C function]

Do Sine transform of the data in given direction or directions. The Sine transform is $\sum a_i \sin(ki)$.

<code>cosfft dat 'dir'</code>	[MGL command]
<code>void CosFFT (const char *dir)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_cosfft (HMDT dat, const char *dir)</code>	[C function]

Do Cosine transform of the data in given direction or directions. The Cosine transform is $\sum a_i \cos(ki)$.

hankel *dat* '*dir*' [MGL command]
void *Hankel* (**const char** **dir*) [Method on *mglData*]
void *mgl_data_hankel* (*HMDT dat*, **const char** **dir*) [C function]

Do Hankel transform of the data in given direction or directions. The Hankel transform is $\sum a_i J_0(ki)$.

swap *dat* '*dir*' [MGL command]
void *Swap* (**const char** **dir*) [Method on *mglData*]
void *mgl_data_swap* (*HMDT 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 *mgl_data_roll* (*HMDT 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 *mgl_data_mirror* (*HMDT 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*->*n-i*. Note, that the similar effect in graphics you can reach by using options (see [Section 3.7 \[Command options\]](#), page 113), for example, **surf** *dat*; **xrange** 1 -1.

sew *dat* [*'dir'*=*'xyz'* *da*=2*pi] [MGL command]
void *Sew* (**const char** **dir*, **float** *da*=2*M_PI) [Method on *mglData*]
void *mgl_data_sew* (*HMDT dat*, **const char** **dir*, **float** *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", **float** *delta*=0) [Method on *mglData*]
void *mgl_data_smooth* (*HMDT dat*, **const char** **dir*, **float** *delta*) [C function]

Smooths the data on specified direction or directions. String *dirs* specifies the dimensions which will be smoothed. It may contain characters: 'x' for 1st dimension, 'y' for 2nd dimension, 'z' for 3d dimension. If string *dir* contain: '0' then does nothing, '3' – linear averaging over 3 points, '5' – linear averaging over 5 points. 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 (float v1=0, float 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  $\geq dim$ .

norms1 dat v1 v2 ['dir='z' keep=on sym=off]                  [MGL command]
void NormS1 (float v1=0, float v2=1, char dir='z',           [Method on mglData]
              bool keep_en=true, bool sym=false)
void mgl_data_norm_slice (HMDT dat, float v1, float 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 linear interpolation by [\[subdata\], page 198](#) command, or spline interpolation by [\[evaluate\], page 199](#) command. Also you can use [\[resize\], page 199](#) for obtaining a data array with new sizes.

However, there are much special faster functions in other modes (C/C++/Fortran/Python/...).■

```

float Spline (float x, float y=0, float z=0) const          [Method on mglData]
float mgl_data_spline (HCDT dat, float x, float y, float 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].

float Spline1 (float x, float y=0, float z=0) const         [Method on mglData]
Interpolates data by cubic spline to the given point x, y, z which assumed to be
normalized in range [0, 1].

float Linear (float x, float y=0, float z=0) const          [Method on mglData]
float mgl_data_linear (HCDT dat, float x, float y, float 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].

float Linear1 (float x, float y=0, float z=0) const         [Method on mglData]
Interpolates data by linear function to the given point x, y, z which assumed to be
normalized in range [0, 1].

```

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 <i>dat</i>	[MGL command]
const char * PrintInfo () const	[Method on <code>mglData</code>]
void PrintInfo (FILE *fp) const	[Method on <code>mglData</code>]
const char * mgl_data_info (HCDT dat)	[C function]
Gets or prints to file <i>fp</i> or as message (in MGL) information about the data (sizes, maximum/minimum, momentums and so on).	
info ' <i>txt</i> '	[MGL command]
Prints string <i>txt</i> as message.	
(dat) .nx	[MGL suffix]
(dat) .ny	[MGL suffix]
(dat) .nz	[MGL suffix]
long GetNx ()	[Method on <code>mglData</code>]
long GetNy ()	[Method on <code>mglData</code>]
long GetNz ()	[Method on <code>mglData</code>]
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]
float Maximal () const	[Method on <code>mglData</code>]
float mgl_data_max (HCDT dat)	[C function]
Gets maximal value of the data.	
(dat) .min	[MGL suffix]
float Minimal () const	[Method on <code>mglData</code>]
float mgl_data_min (HMDT dat) const	[C function]
Gets minimal value of the data.	
float Minimal (int & <i>i</i> , int & <i>j</i> , int & <i>k</i>) const	[Method on <code>mglData</code>]
float mgl_data_min_int (HCDT dat, int * <i>i</i> , int * <i>j</i> , int * <i>k</i>)	[C function]
Gets position of minimum to variables <i>i</i> , <i>j</i> , <i>k</i> and returns the minimal value.	
float Maximal (int & <i>i</i> , int & <i>j</i> , int & <i>k</i>) const	[Method on <code>mglData</code>]
float mgl_data_max_int (HCDT dat, int * <i>i</i> , int * <i>j</i> , int * <i>k</i>)	[C function]
Gets position of maximum to variables <i>i</i> , <i>j</i> , <i>k</i> and returns the maximal value.	
float Minimal (float & <i>x</i> , float & <i>y</i> , float & <i>z</i>) const	[Method on <code>mglData</code>]
float mgl_data_min_real (HCDT dat, float * <i>x</i> , float * <i>y</i> , float * <i>z</i>)	[C function]
Gets approximated (interpolated) position of minimum to variables <i>x</i> , <i>y</i> , <i>z</i> and returns the minimal value.	
(dat) .mx	[MGL suffix]
(dat) .my	[MGL suffix]
(dat) .mz	[MGL suffix]

```

float Maximal (float &x, float &y, float &z) const      [Method on mglData]
float mgl_data_max_real (HCDT dat, float *x, float *y, float *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]

float Momentum (char dir, float &a, float &w) const      [Method on mglData]
float Momentum (char dir, float &m, float &w, float
                 &s, float &k) const      [Method on mglData]
float mgl_data_momentum_val (HCDT dat, char dir, float *a,           [C function]
                               float *w, float *s, float *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,  $k = \sum (\xi_i - a)^4 dat_i / 3Iw^4$ ) 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]
float Find (const char *cond, int &i, int &j, int &k)      [Method on mglData]
    const
float mgl_data_first (HCDT dat, const char *cond, int *i, int *j,     [C function]
                      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]
float Last (const char *cond, int &i, int &j, int &k)      [Method on mglData]
    const
float mgl_data_last (HCDT dat, const char *cond, int *i, int *j,     [C function]
                      int *k)
    Find position (before specified in  $i, j, k$ ) of last nonzero value of formula  $cond$ . Func-
    tion return the data value at found position.

```

```

int Find (const char *cond, char dir, int i=0, int j=0,           [Method on mglData]
          int k=0) const
float 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 mglData]
float 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 mglData &d)                               [Method on mglData]
Copies data from other variable.
```

```

copy dat val                                                 [MGL command]
void operator= (float val)                                     [Method on float]
Set all data values equal to val.
```

```

multo dat dat2                                              [MGL command]
multo dat val                                               [MGL command]
void operator*= (const mglData &d)                            [Method on mglData]
void operator*= (float d)                                     [Method on mglData]
void mgl_data_mul_dat (HMDT dat, HCDT d)                   [C function]
void mgl_data_mul_num (HMDT dat, float d)                  [C function]
Multiplies data element by the other one or by value.
```

```

divto dat dat2                                              [MGL command]
divto dat val                                               [MGL command]
void operator/= (const mglData &d)                            [Method on mglData]
void operator/= (float d)                                     [Method on mglData]
void mgl_data_div_dat (HMDT dat, HCDT d)                   [C function]
void mgl_data_div_num (HMDT dat, float d)                  [C function]
Divides each data element by the other one or by value.
```

```

addto dat dat2                                              [MGL command]
addto dat val                                               [MGL command]
void operator+= (const mglData &d)                            [Method on mglData]
void operator+= (float d)                                     [Method on mglData]
void mgl_data_add_dat (HMDT dat, HCDT d)                   [C function]
void mgl_data_add_num (HMDT dat, float d)                  [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 mglData &d)</code>	[Method on <code>mglData</code>]
<code>void operator-= (float d)</code>	[Method on <code>mglData</code>]
<code>void mgl_data_sub_dat (HMDT dat, HCDT d)</code>	[C function]
<code>void mgl_data_sub_num (HMDT dat, float d)</code>	[C function]
Subtracts from each data element the other one or the value.	
<code>mglData operator+ (const mglData &a, const mglData &b)</code>	[Library Function]
<code>mglData operator+ (float a, const mglData &b)</code>	[Library Function]
<code>mglData operator+ (const mglData &a, float b)</code>	[Library Function]
Adds the other data or the number.	
<code>mglData operator- (const mglData &a, const mglData &b)</code>	[Library Function]
<code>mglData operator- (float a, const mglData &b)</code>	[Library Function]
<code>mglData operator- (const mglData &a, float b)</code>	[Library Function]
Subtracts the other data or the number.	
<code>mglData operator* (const mglData &a, const mglData &b)</code>	[Library Function]
<code>mglData operator* (float a, const mglData &b)</code>	[Library Function]
<code>mglData operator* (const mglData &a, float b)</code>	[Library Function]
Multiplies by the other data or the number.	
<code>mglData operator/ (const mglData &a, const mglData &b)</code>	[Library Function]
<code>mglData operator/ (const mglData &a, float 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 mglData &real, const mglData &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 <code>real</code> , <code>imag</code> on specified direction. The order of transformations is specified in string <code>type</code> : 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 mglData &ampl, const mglData &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 <code>ampl</code> and phase <code>phase</code> of complex numbers.	

```

fourier reDat imDat 'dir'                                [MGL command]
void mglFourier const mglData &re, const mglData &im,
    const char *dir)                                     [Global function]

void mgl_data_fourier HCDT re, HCDT im, const char *dir)      [C function]
    Does Fourier transform of complex data  $re+i*im$  in directions  $dir$ . Result is placed
    back into  $re$  and  $im$  data arrays.

stfad RES real imag dn ['dir='x']                         [MGL command]
mglData mglSTFA (const mglData &real, const mglData &imag,
    int dn, char dir='x')                                 [Global function]

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

pde RES 'ham' ini_re ini_im [dz=0.1 k0=100]             [MGL command]
mglData mglPDE (HMGL gr, const char *ham, const mglData
    &ini_re, const mglData &ini_im, float dz=0.1, float k0=100, const
    char *opt="")
HMDT mgl_pde_solve (HMGL gr, const char *ham, HCDT ini_re, HCDT
    ini_im, float dz, float k0, const char *opt)           [C function]
    Solves equation  $du/dz = i*k0*ham(p,q,x,y,z,|u|)[u]$ , where  $p=-i/k0*d/dx$ ,
 $q=-i/k0*d/dy$  are pseudo-differential operators. Parameters  $ini\_re$ ,  $ini\_im$  specify
real and imaginary part of initial field distribution. Parameters  $Min$ ,  $Max$  set
the bounding box for the solution. Note, that really this ranges are increased by
factor  $3/2$  for purpose of reducing reflection from boundaries. Parameter  $dz$  set the
step along evolutionary coordinate  $z$ . At this moment, simplified form of function
 $ham$  is supported – all “mixed” terms (like ‘ $x*p$ ’-> $x*d/dx$ ) are excluded. For
example, in 2D case this function is effectively  $ham = f(p, z) + g(x, z, u)$ . However
commutable combinations (like ‘ $x*q$ ’-> $x*d/dy$ ) are allowed. Here variable ‘ $u$ ’ is used
for field amplitude  $|u|$ . This allow one solve nonlinear problems – for example,
for nonlinear Shrodinger equation you may set  $ham="p^2 + q^2 - u^2"$ . You may
specify imaginary part for wave absorption, like  $ham = "p^2 + i*x*(x>0)"$ , but only
if dependence on variable ‘ $i$ ’ is linear (i.e.  $ham = hre + i * him$ ). See Section 2.9.11
[PDE solving hints], page 97, 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 p0,
    float dt=0.1, float tmax=10)                           [Global function]

HMDT mgl_ray_trace (const char *ham, float x0, float y0, float
    z0, float px, float py, float pz, float dt, float tmax) [C function]
    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  $\{7 * int(tmax/dt+1)\}$ .

```

```

qo2d RES 'ham' ini_re ini_im ray [r=1 k0=100 xx yy] [MGL command]
mglData mglQ02d (const char *ham, const mglData &ini_re, [Global function]
                  const mglData &ini_im, const mglData &ray, float r=1, float k0=100,
                  mglData *xx=0, mglData *yy=0)
mglData mglQ02d (const char *ham, const mglData &ini_re, [Global function]
                  const mglData &ini_im, const mglData &ray, mglData &xx, mglData &yy,
                  float r=1, float k0=100)
HMDT mgl_qo2d_solve (const char *ham, HCDT ini_re, HCDT ini_im, [C function]
                      HCDT ray, float r, float 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 maked. You may use, for example, the array created by mglRay\(\)
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 mglPDE\(\). See Section 2.9.11 \[PDE solving hints\], page 97, for sample code
and picture.

```

```

jacobian RES xdat ydat [zdat] [MGL command]
mglData mglJacobian (const mglData &x, const mglData &y) [Global function]
mglData mglJacobian (const mglData &x, const mglData &y,
                      const mglData &z) [Global function]
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  $\det \frac{\partial r}{\partial \xi_\beta}$  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 [zdat] [MGL command]
mglData mglTriangulation (const mglData &x, const [Global function]
                           mglData &y)
mglData mglTriangulation (const mglData &x, const [Global function]
                           mglData &y, const mglData &z)
HMDT mgl_triangulation_2d (HCDT x, HCDT y) [C function]
HMDT mgl_triangulation_3d (HCDT x, HCDT y, HCDT z) [C function]
Computes triangulation for arbitrary placed points with coordinates {x,y,z} (i.e. finds
triangles which connect points). The sizes of 1st dimension must be equal for all
arrays x.nx=y.nx=z.nx. Resulting array can be used in \[triplot\], page 175 or \[tricont\],
page 175 functions for visualization of reconstructed surface.

```

6.12 Evaluate expression

MathGL have a special class `mglExpr` for evaluating of formula specified by the string. This class is defined in `#include <mgl2/mgl.h>`. 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 112](#).

<code>mglExpr (const char *expr)</code>	[Constructor on <code>mglExpr</code>]
<code>HMEX mgl_create_expr (const char *expr)</code>	[C function]
Parses the formula <code>expr</code> and creates formula-tree. Constructor recursively parses the formula and creates a tree-like structure containing functions and operators for fast further evaluating by <code>Calc()</code> or <code>CalcD()</code> functions.	
<code>~mglExpr ()</code>	[Destructor on <code>mglExpr</code>]
<code>HMEX mgl_delete_expr (HMEX ex)</code>	[C function]
Deletes the instance of class <code>mglExpr</code> .	
<code>float Eval (float x, float y, float z)</code>	[Method on <code>mglExpr</code>]
<code>float mgl_expr_eval (HMEX ex, float x, float y, float z)</code>	[C function]
Evaluates the formula for ' <code>x</code> ', ' <code>r</code> '= <code>x</code> , ' <code>y</code> ', ' <code>n</code> '= <code>y</code> , ' <code>z</code> ', ' <code>t</code> '= <code>z</code> , ' <code>a</code> ', ' <code>u</code> '= <code>u</code> .	
<code>float Eval (float var[26])</code>	[Method on <code>mglExpr</code>]
<code>float mgl_expr_eval_v (HMEX ex, float *var)</code>	[C function]
Evaluates the formula for variables in array <code>var[0,...,'z'-'a']</code> .	
<code>float Diff (char dir, float x, float y, float z)</code>	[Method on <code>mglExpr</code>]
<code>float mgl_expr_diff (HMEX ex, char dir, float x, float y, float z)</code>	[C function]
Evaluates the formula derivation respect to <code>dir</code> for ' <code>x</code> ', ' <code>r</code> '= <code>x</code> , ' <code>y</code> ', ' <code>n</code> '= <code>y</code> , ' <code>z</code> ', ' <code>t</code> '= <code>z</code> , ' <code>a</code> ', ' <code>u</code> '= <code>u</code> .	
<code>float Diff (char dir, float var[26])</code>	[Method on <code>mglExpr</code>]
<code>float mgl_expr_diff_v (HMEX ex, char dir, float *var)</code>	[C function]
Evaluates the formula derivation respect to <code>dir</code> for variables in array <code>var[0,...,'z'-'a']</code> .	

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 113](#)). Symbol `:` starts new command (like new line character) if it is not placed inside a string or inside brackets.

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

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

- The string is any symbols between ordinary marks `''`. Long strings can be concatenated from several lines by `\`` symbol. I.e. the string `'a +'`
` b'` will give string `'a + b'` (here `
` is newline).
- Usually variable have a name which is arbitrary combination of symbols (except spaces and `''`) started from a letter and with length less than 64. A temporary array can be used as variable:
 - sub-arrays (like in [\[subdata\], page 198](#) 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 196](#) 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 198](#)) 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`, `pi=3.1415926...`, `on=1`, `off=0`, `:=1` are treated as number if they were not redefined by user. Variables with suffixes are treated as numbers (see [Section 6.9 \[Data information\], page 203](#)). Names defined by [\[define\]](#), page 212 command are treated as number. Also results of formulas with sizes $1 \times 1 \times 1$ 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 192, [\[var\]](#), page 195, [\[list\]](#), page 194, [\[copy\]](#), page 192, [\[read\]](#), page 197, [\[hist\]](#), page 199, [\[sum\]](#), page 200 and so on (see sections [Section 6.2 \[Data constructor\], page 192](#), [Section 6.4 \[Data filling\], page 194](#) and [Section 6.6 \[Make another data\], page 198](#)).

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 116](#) and [Chapter 6 \[Data processing\], page 191](#).

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. Here *N* is digit (0...9) or alpha (a...z).

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).

defpal \$N smth	[MGL command]
Sets <i>N</i> -th script argument to palette character at position evaluated from <i>smth</i> . Here <i>N</i> is digit (0...9) or alpha (a...z).	
call 'fname' [ARG1 ARG2 ... ARG9]	[MGL command]
Executes function <i>fname</i> (or script if function is not found). Optional arguments will be passed to functions. See also [func], page 213.	
func 'fname' [narg=0]	[MGL command]
Define the function <i>fname</i> and number of required arguments. The arguments will be placed in script parameters \$1, \$2, ... \$9. Note, you should stop script execution before function definition(s) by command [stop], page 213. See also [return], page 213.	
return	[MGL command]
Return from the function. See also [func], page 213.	
if dat 'cond'	[MGL command]
Starts block which will be executed if <i>dat</i> satisfy to <i>cond</i> .	
if val	[MGL command]
Starts block which will be executed if <i>val</i> is nonzero.	
elseif dat 'cond'	[MGL command]
Starts block which will be executed if previous if or elseif is false and <i>dat</i> satisfy to <i>cond</i> .	
elseif val	[MGL command]
Starts block which will be executed if previous if or elseif is false and <i>val</i> 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 \$ <i>N</i> -th argument changing from <i>v1</i> to <i>v2</i> with the step <i>dv</i> . Here <i>N</i> is digit (0...9) or alpha (a...z).	
for \$N dat	[MGL command]
Starts cycle with \$ <i>N</i> -th argument changing for <i>dat</i> values. Here <i>N</i> is digit (0...9) or alpha (a...z).	
next	[MGL command]
Finishes for cycle.	
once val	[MGL command]
The code between once on and once off will be executed only once. Useful for large data manipulation in programs like UDAV.	
stop	[MGL command]
Terminate execution.	

7.3 mglParse class

Class for parsing and executing MGL script. This class is defined in `#include <mgl2/mgl.h>`.

Class `mglParse` is the interpreter for MGL scripts. 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. These functions can be useful for displaying values of variables (arrays) in some external program (in window, for example). Function `AllowSetSize()` allows one to prevent changing the size of the picture inside the script (forbids the MGL command `setsiz`e).

<code>mglParse (bool setsiz=false)</code>	[Constructor on <code>mglParse</code>]
<code>mglParse (HMPR pr)</code>	[Constructor on <code>mglParse</code>]
<code>mglParse (mglParse &pr)</code>	[Constructor on <code>mglParse</code>]
<code>HMPR mgl_create_parser ()</code>	[C function]
Constructor initializes all values with zero and set <code>AllowSetSize</code> value.	
<code>~mglParse ()</code>	[Destructor on <code>mglParse</code>]
<code>void mgl_delete_parser (HMPR p)</code>	[C function]
Destructor delete parser	
<code>HMPR Self ()</code>	[Method on <code>mglParse</code>]
Returns the pointer to internal object of type <code>HMPR</code> .	
<code>void Execute (mglGraph *gr, const char *text)</code>	[Method on <code>mglParse</code>]
<code>void Execute (mglGraph *gr, const wchar_t *text)</code>	[Method on <code>mglParse</code>]
<code>void mgl_parse_text (HMGL gr, HMPR p, const char *text)</code>	[C function]
<code>void mgl_parsew_text (HMGL gr, HMPR p, const wchar_t *text)</code>	[C function]
Main function in the class. Function parse and execute line-by-line MGL script in array <code>text</code> . Lines are separated by newline symbol ‘\n’ as usual.	
<code>void Execute (mglGraph *gr, FILE *fp, bool print=false)</code>	[Method on <code>mglParse</code>]
<code>int mgl_parse_file (HMGL gr, HMPR p, FILE *fp, int print)</code>	[C function]
The same as previous but read script from the file <code>fp</code> . If <code>print=true</code> then all warnings and information will be printed in stdout.	
<code>int Parse (mglGraph *gr, const char *str, long pos=0)</code>	[Method on <code>mglParse</code>]
<code>int Parse (mglGraph *gr, const wchar_t *str, long pos=0)</code>	[Method on <code>mglParse</code>]
<code>int mgl_parse (HMGL gr, HMPR p, const char *str, int pos)</code>	[C function]
<code>int mgl_parsew (HMGL gr, HMPR p, const wchar_t *str, int pos)</code>	[C function]
Function parses the string <code>str</code> and executes it by using <code>gr</code> as a graphics plotter. Returns the value depending on an error presence in the string <code>str</code> : 0 – no error, 1 – wrong command argument(s), 2 – unknown command, 3 – string is too long. Optional argument <code>pos</code> allows to save the string position in the document (or file) for using <code>for next</code> command.	
<code>void AddParam (int n, const char *str)</code>	[Method on <code>mglParse</code>]
<code>void AddParam (int n, const wchar_t *str)</code>	[Method on <code>mglParse</code>]

```

void mgl_add_param (HMPR p, int id, const char *val)           [C function]
void mgl_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 ... 'z'-'a'+10). String
    str shouldn't contain '$' symbol.

mglData * FindVar (const char *name)                         [Method on mglParse]
HMDT mgl_find_var (HMPR p, const char *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!

mglData * AddVar (const char *name)                          [Method on mglParse]
HMDT mgl_add_var (HMPR p, const char *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 mgl_del_var (HMPR p, const char *name)                [C function]
    Function delete the variable specified by its name or by its pointer.

void RestoreOnce ()                                       [Method on mglParse]
void mgl_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 'setsiz' 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.

int CmdType (const char *name)                           [Method on mglParse]
int mgl_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 - create data, 6 -
    define subplot, 7 - program flow.

const char * CmdFormat (const char *name)                [Method on mglParse]
const char * mgl_cmd_frmt (HMPR p, const char *name)   [C function]
    Return the format of arguments for MGL command name.

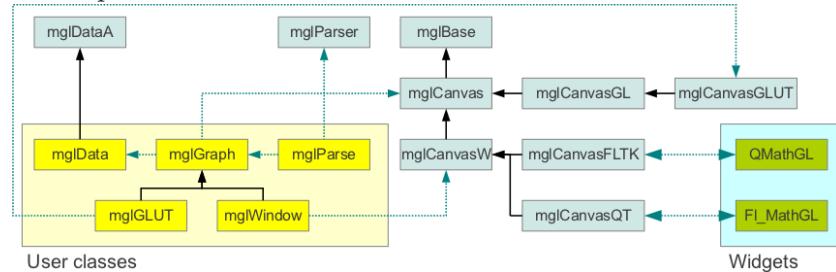
const char * CmdDesc (const char *name)                  [Method on mglParse]
const char * mgl_cmd_desc (HMPR p, const char *name)    [C function]
    Return the description of MGL command name.

```

8 Other classes

There are only 5 end-user classes: `mglGraph` (see Chapter 4 [MathGL core], page 116), `mglWindow` and `mglGLUT` (see Chapter 5 [Widget classes], page 182), `mglData` (see Chapter 6 [Data processing], page 191), `mglParse` (see Chapter 7 [MGL scripts], page 211). 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 2 classes which are widgets for FLTK and Qt libraries (green ones in the figure).

Below I show how this internal classes can be used.

8.0.1 Define new kind of plot (`mglBase` class)

Basically most of new kinds of plot can be created using just MathGL primitives (see Section 4.6 [Primitives], page 136). 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 float 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 and 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, float x, float y, float z, float 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, float x, float y, float z, float 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 113](#)).

```
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 = stl && strchr(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.

```
float c=gr->NextColor(pal), d;
```

Second one for border. I use black color (call `gr->AddTexture('k')`) if second color is not specified.

```
float 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.

```
float 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, float x, float y, float z, float 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 = stl && strchr(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.

```
float c=gr->GetC(ss,r);
```

Second one for border.

```
float 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;
    float 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, float r, const char *stl="", const char *opt="")
    { circle(p.x,p.y,p.z, r, stl, opt); }
    inline void CircleCS(mglPoint p, float 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>
//-----
#ifndef __cplusplus
extern "C" {
#endif
void circle(HMGL gr, float x, float y, float z, float r, const char *stl, const char *opt);
void circle_cs(HMGL gr, float x, float y, float z, float r, const char *stl, const char *opt)
#ifndef __cplusplus
}
#endif
//-----
class MyGraph : public mglGraph
{
public:
    inline void CircleCF(mglPoint p, float r, const char *stl="", const char *opt="")
    { circle(p.x,p.y,p.z, r, stl, opt); }
    inline void CircleCS(mglPoint p, float r, const char *stl="", const char *opt="")
    { circle_cs(p.x,p.y,p.z, r, stl, opt); }
};
//-----
void circle(HMGL gr, float x, float y, float z, float 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 = stl && strchr(stl,'@');
long pal=0;
char mk=gr->SetPenPal(stl,&pal);
float c=gr->NextColor(pal), d;
float 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;
    float 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, float x, float y, float z, float r, const char *stl, const char *op
{
    static int cgid=1; gr->StartGroup("CircleCS",cgid++);
    gr->SaveState(op);
    const int n = gr->MeshNum>1?gr->MeshNum : 41;
    bool fill = stl && strchr(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);
    float c=gr->GetC(ss,r);
    float 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;
    float 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());
}

```

8.0.2 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; /////
    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 float Maximal() const { return mgl_data_max(this); }
    /// Get minimal value of the data
    inline float 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];
      return use_abs? abs(res) : arg(res); }
    inline mreal dvy(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];
      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];
      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;
}

```

8.1 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 float 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 108.

`float r, g, b, a` [Parameter of `mglColor`]
 Reg, green and blue component of color.

`mglColor (float R, float G, float B, float A=1)` [Method on `mglColor`]
 Constructor sets the color by float values of Red, Green, Blue and Alpha channels.
 These values should be in interval [0,1].

`mglColor (char c='k', float bright=1)` [Method on `mglColor`]
 Constructor sets the color from character id. The black color is used by default.
 Parameter `br` set additional “lightness” of color.

`void Set (float R, float G, float B, float A=1)` [Method on `mglColor`]
 Sets color from values of Red, Green, Blue and Alpha channels. These values should
 be in interval [0,1].

`void Set (mglColor c, float bright=1)` [Method on `mglColor`]
 Sets color as “lighted” version of color `c`.

`void Set (char p, float bright=1)` [Method on `mglColor`]
 Sets color from symbolic id.

`bool Valid ()` [Method on `mglColor`]
 Checks correctness of the color.

`float Norm ()` [Method on `mglColor`]
 Gets maximal of spectral component.

<code>bool operator==(const mglColor &c)</code>	[Method on <code>mglColor</code>]
<code>bool operator!=(const mglColor &c)</code>	[Method on <code>mglColor</code>]
Compare with another color	
<code>bool operator*=(float v)</code>	[Method on <code>mglColor</code>]
Multiplies color components by number <code>v</code> .	
<code>bool operator+=(const mglColor &c)</code>	[Method on <code>mglColor</code>]
Adds color <code>c</code> component by component.	
<code>bool operator-=(const mglColor &c)</code>	[Method on <code>mglColor</code>]
Subtracts color <code>c</code> component by component.	
<code>mglColor operator+ (const mglColor &a, const mglColor &b)</code>	[Library Function]
Adds colors by its RGB values.	
<code>mglColor operator- (const mglColor &a, const mglColor &b)</code>	[Library Function]
Subtracts colors by its RGB values.	
<code>mglColor operator* (const mglColor &a, float b)</code>	[Library Function]
<code>mglColor operator* (float a, const mglColor &b)</code>	[Library Function]
Multiplies color by number.	
<code>mglColor operator/ (const mglColor &a, float b)</code>	[Library Function]
Divide color by number.	
<code>mglColor operator! (const mglColor &a)</code>	[Library Function]
Return inverted color.	

8.2 `mglPoint` class

Structure describes point in space. This structure is defined in `#include <mgl2/type.h>`

<code>float x, y, z, c</code>	[Parameter of <code>mglPoint</code>]
Point coordinates {x,y,z} and one extra value <code>c</code> used for amplitude, transparency and so on. By default all values are zero.	
<code>mglPoint (float X=0, float Y=0, float Z=0, float C=0)</code>	[Method on <code>mglPoint</code>]
Constructor sets the color by float values of Red, Green, Blue and Alpha channels. These values should be in interval [0,1].	
<code>bool IsNaN ()</code>	[Method on <code>mglPoint</code>]
Returns <code>true</code> if point contain NAN values.	
<code>float norm ()</code>	[Method on <code>mglPoint</code>]
Returns the norm $\sqrt{x^2 + y^2 + z^2}$ of vector.	
<code>void Normalize ()</code>	[Method on <code>mglPoint</code>]
Normalizes vector to be unit vector.	

float val (int i)	[Method on <code>mglPoint</code>]
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* (float a, const mglPoint &b)	[Library Function]
mglPoint operator* (const mglPoint &a, float b)	[Library Function]
Multiplies (scale) points by number.	
mglPoint operator/ (const mglPoint &a, float b)	[Library Function]
Multiplies (scale) points by number $1/b$.	
float 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.	
mglPoint operator& (const mglPoint &a, const mglPoint &b)	[Library Function]
The part of a which is perpendicular to vector b .	
mglPoint operator (const mglPoint &a, const mglPoint &b)	[Library Function]
The part of a which is parallel to vector b .	
mglPoint operator! (const mglPoint &a)	[Library Function]
Return vector perpendicular to vector a .	
float mgl_norm (const mglPoint &a)	[Library Function]
Return the norm $\sqrt{ a ^2}$ of vector a .	
bool operator== (const mglPoint &a, const mglPoint &b)	[Library Function]
Return true if points are the same.	
bool operator!= (const mglPoint &a, const mglPoint &b)	[Library Function]
Return true if points are different.	

8.3 mglFont class

Class for working with font: load, get metrics, parse and draw strings. This class is defined in `#include <mgl2/font.h>`. This class is accessible only from C++ (and only from GNU compilers in default binary files).

The class is based on loading and drawing of vector Hershey font. There are two styles of specifying of the font type and aligning: by integer parameters or by string.

The string can be any combination of characters: ‘rbiLCRwou’. The font types are: ‘r’ – roman font, ‘i’ – italic style, ‘b’ – bold style. By default roman font (that is ‘’ or ‘r’) is used. The align types are: ‘L’ – align left (default), ‘C’ – align center, ‘R’ – align right. Additional font effects are: ‘w’ – wire, ‘o’ – over-lined, ‘u’ – underlined. Parsing of the string to special (TeX-like) commands will be done if variable `parse` is true (it’s default). See also see [Section 3.5 \[Font styles\]](#), page 110.

The over way of font and alignment setting is the using of the integer constant. Integer font Id can be one of predefined constants: `MGL_FONT_ITAL`, `MGL_FONT_BOLD`, `MGL_FONT_BOLD_ITAL` = `MGL_FONT_ITAL+MGL_FONT_BOLD`. Default font is `MGL_FONT_ROMAN`. Also there are flags `MGL_FONT_ULINE`, `MGL_FONT_OLINE`, `MGL_FONT_WIRE` for additional font effects. Align parameter controls the text alignment: 0 – align left, 1 – align center, 2 – align right.

`mglFont (const char *name=MGL_DEF_FONT_NAME, const char *path=NULL)` [Method on `mglFont`]

Initialize the font and load data from file `name` (default name is "STIX" for Linux and MacOS) or if `name=NULL` limited data from memory (default for Windows).

`bool Load (const char * base, const char *path=NULL)` [Method on `mglFont`]

Load font from file `path/base` into the memory. The font may contain 4 files: `base.vfm`, `base_b.vfm`, `base_i.vfm`, `base_bi.vfm`. Appendix contain detailed description of font format.

`void Restore ()` [Method on `mglFont`]

Restore default font.

`void Copy (mglFont *fnt)` [Method on `mglFont`]

Copy data from other font instance.

`void Clear ()` [Method on `mglFont`]

Clear memory by deleting the loaded font.

`unsigned GetNumGlyph ()` [Method on `mglFont`]

Return the number of glyphs in the font.

`bool Ready ()` [Method on `mglFont`]

Return true if font is loaded and ready for use.

`float Height (int font)` [Method on `mglFont`]

Gets height of text for font specified by integer constant.

`float Puts (const char *str, int font=0, int align=0)` [Method on `mglFont`]

Prints 8-bit text string for font specified by integer constant.

```

float Width (const char *str, int font=0) [Method on mglFont]
    Gets width of 8-bit text string for font specified by integer constant.

float Puts (const wchar_t *str, int font=0, int align=0) [Method on mglFont]
    Prints Unicode text string for font specified by integer constant.

float Width (const wchar_t *str, int font=0) [Method on mglFont]
    Gets width of Unicode text string for font specified by integer constant.

float Height (const char *how) [Method on mglFont]
    Gets height of text for font specified by string.

float Puts (const char *str, const char *how) [Method on mglFont]
    Prints 8-bit text string for font specified by string.

float Width (const char *str, const char *how) [Method on mglFont]
    Gets width of 8-bit text string for font specified by string.

float Puts (const wchar_t *str, const char *how) [Method on mglFont]
    Prints Unicode text string for font specified by string.

float Width (const wchar_t *str, const char *how) [Method on mglFont]
    Gets width of Unicode text string for font specified by string.

HMGL gr [Parameter of mglFont]
    Instance of mglGraph class which is used for character drawing.

bool parse [Parameter of mglFont]
    Flag for switching on/off the parsing of TeX commands. Default value is true.

```

8.3.1 Format of 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 (float), *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}.

Appendix A TeX-like symbols

The full list of TeX-like commands recognizable by MathGL is shown below. If command is not recognized then it will be printed as is by ommitting ‘\’ symbol. For example, ‘\#’ produce “#”, ‘\\’ produce “\”, ‘\qq’ produce “qq”.

Change case: _, ^, @.

Text style: \big, \b, \textbf, \i, \textit, \bi, \r, \textrm, \a, \overline, \u, \underline, \w, \wire, \#, \color[wkrgbcymhRGBCYMHWlenupqLENUPQ]

Roots: \sqrt, \sqrt3, \sqrt4

Fractions: \frac, \stack, \overset, \underset, \stackr, \stackl

Accents: \hat, \tilde, \dot, \ddot, \ddot, \ddot, \acute, \check, \grave, \vec, \bar, \breve

Special symbols:

\# (#), \% (%), \& (&), \^ (^).

\AA (), \AE (), \APLboxquestion (), \APLboxupcaret (), \APLnotbackslash (), \APLnotslash (), \Alpha (), \And (&), \Angstrom (), \Barv (), \BbbC (), \BbbGamma (), \BbbH (), \BbbN (), \BbbP (), \BbbPi (), \BbbQ (), \BbbR (), \BbbZ (), \Bbbgamma (), \Bbbpi (), \Bbbsum (), \Beta (), \Bumpeq (), \Cap (), \Chi (), \Colon (), \Coloneq (), \Cup (), \DDownarrow (), \DH (), \DJ (), \DashV (), \DashVDash (), \Dashv (), \Ddownarrow (), \Delta (), \Digamma (), \Doteq (), \Downarrow (), \Epsilon (), \Equiv (), \Eta (), \Eulerconst (), \Exclam (), \Finv (), \Game (), \Gamma (), \Gt (), \Hermaphrodite (), \Im (), \Iota (), \Kappa (), \Koppa (), \L (), \LLeftrightarrow (), \Lambda (), \Lbrbrak (), \Ldsh (), \Leftarrow (), \Leftrightarrow (), \Lleftarrow (), \Longleftarrow (), \Longleftrightarrow (), \Longmapsfrom (), \Longmapsto (), \Longrightarrow (), \Lparenctr (), \Lsh (), \Lt (), \Lvzigzag (), \Mapsfrom (), \Mapsto (), \Mu (), \NG (), \Nearrow (), \Not (), \Nu (), \Nwarrow (), \O (), \OE (), \Ohorn (), \Omega (), \Omicron (), \Otimes (), \P (), \Phi (), \Pi (), \Planckconst (), \Prec (), \PropertyLine (), \Psi (), \QED (), \Question (), \RRrightarrow (), \Rbrbrak (), \Rdsh (), \Re (), \Rho (), \Rightarrow (), \Rparenless (), \Rrightarrow (), \Rsh (), \Rvzigzag (), \S (), \Sc (), \Searrow (), \Sigma (), \Sqcap (), \Sqcup (), \Stigma (), \Subset (), \Supset (), \Swallow (), \TH (), \Tau (), \Theta (), \UUparrow (), \Uhorn (), \Uparrow (), \Updownarrow (), \Uuparrow (), \Vdash (), \Vbar (), \Vdash (), \Vee (), \Vert (), \Vvdash (), \Vvert (), \Wedge (), \XBox (), \Xi (), \Yup (), \Zbar (), \Zeta ().

\aa (), \ac (), \accurrent (), \acidfree (), \acwcirclearrow (), \acwgapcirclearrow (), \acwleftarcarrow (), \acwopencirclearrow (), \acwoverarcarrow (), \acwundercurvearrow (), \adots (), \ae (), \aleph (), \alpha (), \amalg (), \angdn (), \angle (), \angles (), \angleubar (), \approx (), \approxeq (), \approxeqq (), \approxident (), \arceq (), \aries (), \assert (), \ast (), \asteq (), \astrosun (), \asymp (), \awint () .

\bNot (), \backcong (), \backdprime (), \backepsilon (), \backprime (), \backsim (), \backsimeq (), \backslash (), \backrprime (), \bagmember (), \barV (), \barcap (), \barcup (), \bardownharpoonleft (), \bardownharpoonright (), \barleftarrow (), \barlefarrowrightarrowbar (), \barleftharpoondown (), \barleftharpoonup (), \barovernorthwes-tarrow (), \barrightarrowdiamond (), \barrightharpoondown (), \barrightharpoonup (), \baruparrow (), \barupharpoonleft (), \barupharpoonright (), \barvee (), \barwedge (),

\bbrktbrk (), \bdHrule (), \bdVrule (), \bdbVbH (), \bdbVbh (), \bdbVIH (), \bdbVlh () , \bdbVrH (), \bdbVrh (), \bdbvbH (), \bdbvbh (), \bdbvlH (), \bdbvlh (), \bdbvrlH () , \bdbvrlh (), \bdbvrH (), \bdbvrh (), \bddVbH (), \bddVbh (), \bddVIH (), \bddVlh (), \bddVrH (), \bddVrh () , \bddvbH (), \bddvbh (), \bddvlH (), \bddvlh (), \bddvrH (), \bddvrh (), \bdhrule (), \bdnesw (), \bdnwse (), \bdquadhdash (), \bdquadvdash (), \bdtriplevdash (), \bduVbH () , \bduVbh (), \bduVIH (), \bduVlh (), \bduVrH (), \bduVrh (), \bduvbH (), \bduvbh () , \bduvlH (), \bduvlh (), \bduvrH (), \bduvrh (), \bdvrule (), \because (), \benzenr (), \beta () , \beth (), \between (), \bigblacktriangledown (), \bigblacktriangleup (), \bigbot (), \bigcap (), \bigcup (), \bigslantedvee (), \bigslantedwedge (), \bigstar (), \bigtop (), \bigtriangledown (), \bigtriangleup (), \bigvee (), \bigwedge (), \bigwhitestar (), \blackcircledownarrow (), \blackcircledrightdot (), \blackcircledsanseight (), \blackcircledsansfive (), \blackcircledsansfour (), \blackcircledsansnine (), \blackcircledsansone (), \blackcircledsansseven (), \blackcircledsanssix (), \blackcircledsansten (), \blackcircledsanthree (), \blackcircledsanstwo (), \blackcircledtwodots (), \blackcircleulquadwhite (), \blackdiamonddownarrow (), \blackhourglass (), \blackinwhitediamond (), \blackinwhitesquare (), \blacklefthalfcircle (), \blackpointerleft (), \blackpointerright (), \blackrighthalfcircle (), \blacksmiley (), \blacktriangle (), \blacktriangledown (), \blacktriangleleft (), \blacktriangleright (), \blkhorzoval (), \blkvertoval (), \blockfull (), \blockhalfshaded (), \blocklighthalf (), \blocklowhalf (), \blockqtrshaded (), \blockrighthalf (), \blockthreeqtrshaded (), \blockuphalf (), \bot (), \botsemicircle (), \bowtie (), \box (), \boxast (), \boxbar (), \boxbox (), \boxbslash (), \boxcircle (), \boxdiag (), \boxdot (), \boxminus (), \boxonbox (), \boxplus (), \boxtimes (), \bsimilarleftarrow (), \bsimilarrightarrow (), \bsolhsub (), \btimes (), \bullet (), \bullseye (), \bumpeq (), \bumpeqq ().

\calB (), \calE (), \calF (), \calH (), \calM (), \calR (), \cap (), \capdot (), \capwedge (), \caretinsert (), \carreturn (), \carriagereturn (), \ccwundercurvearrow (), \cdot (), \cdotp () , \cdots (), \cdprime (), \checkmark (), \chi (), \cirE (), \cirbot (), \circ () , \circeq (), \circfint (), \circlebottomhalfblack (), \circledA (), \circledB (), \circledC (), \circledD (), \circledE (), \circledF (), \circledG (), \circledH (), \circledI (), \circledJ (), \circledK (), \circledL (), \circledM (), \circledN (), \circledO (), \circledP (), \circledQ (), \circledR (), \circledS (), \circledT (), \circledU (), \circledV (), \circledW (), \circledX (), \circledY (), \circledZ (), \circleda (), \circledast (), \circledb (), \circledbullet (), \circledc (), \circledcirc (), \circledd (), \circleddash (), \circlede (), \circledeight (), \circledequal (), \circledf (), \circledfive (), \circledfour (), \circledg (), \circledh (), \circledi (), \circledj (), \circledk (), \circledl (), \circledm (), \circledn (), \circlednine (), \circledo (), \circledone (), \circledarrow (), \circledp (), \circledparallel (), \circledq (), \circledr (), \circledrightdot (), \circleds (), \circledsanseight (), \circledsansfive (), \circledsansfour (), \circledsansnine (), \circledsansone (), \circledsansseven (), \circledsanssix (), \circledsansten (), \circledsanthree (), \circledsanstwo (), \circledseven (), \circledsix (), \circledstar (), \circledt (), \circledthree (), \circledtwo (), \circledtwodots (), \circledu (), \circledv (), \circledvert (), \circledw (), \circledwhitebullet (), \circledx (), \circledy (), \circledz (), \circledzero (), \circlehbar (), \circlelefthalfblack (), \circrellquad (), \circlelrquad (), \circleonleftarrow (), \circleonrightarrow (), \circlerighthalfblack (), \circletophalfblack (), \circleulquad (), \circleurquad (), \circleurquadblack (), \circlevertfill (), \cirmid (), \cirsir (), \clangle (), \closedvarcap (), \closedvarcup (), \closedvarcupsmashprod (), \closure (), \cloverleaf (), \clubsuit (), \colon (:), \colon (), \coloneq (), \commaminus (), \complement (), \concavediamond (), \concavediamondtick-

left (), \concavediamondtickright (), \cong (), \congdot (), \conictaper (), \conjunction (), \coprod (), \cprime (), \crangle (), \csub (), \csube (), \csup (), \csupe (), \cuberoot (), \cup (), \cupdot (), \cupleftarrow (), \cupvee (), \curlyeqprec (), \curlyeqsucc (), \curlyvee (), \curlywedge (), \curvearrowleft (), \curvearrowleftplus (), \curvearrowright (), \curvearrowrightminus (), \wcirclearrow (), \wgapcclearrow (), \wopencirclearrow (), \wrightarcarrow (), \wundercurvearrow (), \cylcty ().

 \dag (), \dagger (), \daleth (), \danger (), \dashV (), \dashVdash (), \dashcolon (), \dashleftharpoondown (), \dashrightharpoondown (), \dashv (), \dbkarow (), \ddag (), \ddagger (), \ddots (), \ddotseq (), \delta (), \dh (), \diameter (), \diamond (), \diamondbotblack (), \diamondcdot (), \diamondleftarrow (), \diamondleftarrowbar (), \diamondleftblack (), \diamondrightblack (), \diamondsuit (), \diamondtopblack (), \dicei (), \diceii (), \diceiii (), \diceiv (), \dicev (), \digamma (), \dingasterisk (), \dircurrent (), \disin (), \div (), \divideontimes (), \dj (), \dlcrop (), \doteq (), \dotequiv (), \dotminus (), \dotplus (), \dots (), \dotsim (), \dotsminusdots (), \dottedcircle (), \dottedsquare (), \dottimes (), \doublebarvee (), \doublebarwedge (), \doubleplus (), \downarrow (), \downarrowbar (), \downarrowbarred (), \downdasharrow (), \downdownarrows (), \downfishtail (), \downharpoonleft (), \downharpoonleftbar (), \downharpoonright (), \downharpoonrightbar (), \downharpoonsleftright (), \downrightcurvedarrow (), \downtriangleleftblack (), \downtrianglerightblack (), \downuparrows (), \downupharpoonsleftright (), \downwhitearrow (), \downzigzagarrow (), \dprime (), \draftingarrow (), \drbkarow (), \drcrop (), \dsol (), \dsub (), \dualmap ().

 \earth (), \egsdot (), \eighthnote (), \elinters (), \ell () , \elsdot (), \emdash (), \emptyset (), \emptysetoarr (), \emptysetoarrl (), \emptysetobar (), \emptysetocirc (), \endash (), \enleadertwodots (), \envelope (), \eparsl (), \epsilon (), \eqcirc (), \eqcolon (), \eqdef (), \eqdot (), \eqeq (), \eqeqeq (), \eqgtr (), \eqless (), \eqqgrt (), \eqqlss (), \eqqplus (), \eqqsim (), \eqqlantgrt (), \eqqlantless (), \eqsim (), \eqslantgrt (), \eqslantless (), \equalleftarrow (), \equalparallel (), \equalrightarrow (), \equiv (), \equivDD (), \equivVert (), \equivVvert (), \eqvparsl (), \errbarblackcircle (), \errbarblackdiamond (), \errbarblacksquare (), \errbarcircle (), \errbardiamond (), \errbarsquare (), \eta (), \euro (), \exists ().

 \fallingdotseq (), \fbowtie (), \fcmp (), \fdiagovnearrow (), \fdiagovrdiag (), \female (), \figdash (), \fint (), \fisheye (), \flat (), \fltns (), \forall (), \forks (), \forksnot (), \forkv (), \fourthroot (), \fourvdots (), \fracfiveeighths (), \fracfivesixths (), \fracfourfifths (), \fraconeeighth (), \fraconefifth (), \fraconesixth (), \fraconethird (), \fracseveneights (), \fracslash (), \fracthreeeighths (), \fracthreefifths (), \fractwofifths (), \fractwothirds (), \frakC (), \frakH (), \frakZ (), \frown (), \frownie (), \fullouterjoin ().

 \gamma (), \ge (), \geq (), \geqq (), \geqlant (), \gescc (), \gesdot (), \gesdoto (), \gesdotol (), \gesles (), \gets (), \gg (), \ggg (), \gggnest (), \gimel (), \glE (), \gla (), \gleichstark (), \glj (), \gnapprox (), \gneq (), \gneqq (), \gnsim (), \greater (>), \gsime (), \gsiml (), \gtcc (), \gtcir (), \gtlpar (), \gtquest (), \gttrapprox (), \gttrarr (), \gttrdot (), \gtreqless (), \gtreqqless (), \gtreqless (), \gtreqsim (), \guillemotleft (), \guillemotright (), \guilsinglleft (), \guilsinglright ().

 \harrowextender (), \hatapprox (), \hbar (), \heartsuit (), \hermitmatrix (), \hexagon (), \hexagonblack (), \hiraganano (), \hknearrow (), \hknarrow (), \hksearrow (), \hkswarow (), \hookleftarrow (), \hookrightarrow (), \horizbar (), \hourglass (), \house (), \hrectangle (), \hrectangleblack (), \hslash (), \hyphenbullet (), \hzigzag () .

$\backslash\text{iiiint}()$, $\backslash\text{iint}()$, $\backslash\text{infin}()$, $\backslash\text{int}()$, $\backslash\text{imageof}()$, $\backslash\text{in}()$, $\backslash\text{increare}()$, $\backslash\text{increment}()$, $\backslash\text{infty}()$, $\backslash\text{int}()$, $\backslash\text{intBar}()$, $\backslash\text{intbar}()$, $\backslash\text{intbottom}()$, $\backslash\text{intcap}()$, $\backslash\text{intclockwise}()$, $\backslash\text{intcup}()$, $\backslash\text{intercal}()$, $\backslash\text{interleave}()$, $\backslash\text{intextender}()$, $\backslash\text{intlharhk}()$, $\backslash\text{intprod}()$, $\backslash\text{intprodr}()$, $\backslash\text{inttop}()$, $\backslash\text{intx}()$, $\backslash\text{inversebullet}()$, $\backslash\text{inversewhitecircle}()$, $\backslash\text{invnot}()$, $\backslash\text{invwhitehalfcircle}()$, $\backslash\text{invwhiteupperhalfcircle}()$, $\backslash\text{iota}()$, $\backslash\text{ipasupgamma}()$, $\backslash\text{ipasupl}()$, $\backslash\text{ipasuperglotstpp}()$, $\backslash\text{ipasups}()$, $\backslash\text{ipasupx}()$, $\backslash\text{ipaunaspirated}()$, $\backslash\text{ipavoicing}()$, $\backslash\text{isinE}()$, $\backslash\text{isindot}()$, $\backslash\text{isinobar}()$, $\backslash\text{isins}()$, $\backslash\text{isinvb}()$, $\backslash\text{itBbbD}()$, $\backslash\text{itBbdd}()$, $\backslash\text{itBbbe}()$, $\backslash\text{itBbbi}()$, $\backslash\text{itBbbj}()$.

$\backslash\text{jupiter}()$, $\backslash\text{kappa}()$, $\backslash\text{kernelcontraction}()$, $\backslash\text{koppa}()$.

$\backslash\text{l}()$, $\backslash\text{lAngle}()$, $\backslash\text{lBrace}()$, $\backslash\text{lBrack}()$, $\backslash\text{lParen}()$, $\backslash\text{lambdabar}()$, $\backslash\text{langle}()$, $\backslash\text{langledot}()$, $\backslash\text{laplac}()$, $\backslash\text{lasp}()$, $\backslash\text{lat}()$, $\backslash\text{late}()$, $\backslash\text{lbag}()$, $\backslash\text{lblkrbrak}()$, $\backslash\text{lbrace}()$, $\backslash\text{lbraceend}()$, $\backslash\text{lbracemid}()$, $\backslash\text{lbraceuend}()$, $\backslash\text{lbrack}()$, $\backslash\text{lbrackextender}()$, $\backslash\text{lbrackend}()$, $\backslash\text{lbracklltick}()$, $\backslash\text{lbrackubar}()$, $\backslash\text{lbrackuend}()$, $\backslash\text{lbrackultick}()$, $\backslash\text{lbrbrak}()$, $\backslash\text{lceil}()$, $\backslash\text{lcurvyangle}()$, $\backslash\text{dasharrhead}()$, $\backslash\text{le}()$, $\backslash\text{leadsto}()$, $\backslash\text{leftarrow}()$, $\backslash\text{leftarrowapprox}()$, $\backslash\text{leftarrowbackapprox}()$, $\backslash\text{leftarrowbsimilar}()$, $\backslash\text{leftarrowless}()$, $\backslash\text{leftarrowonoplus}()$, $\backslash\text{leftarrowplus}()$, $\backslash\text{leftarrowshortrightarrow}()$, $\backslash\text{leftarrowsimilar}()$, $\backslash\text{leftarrowsubset}()$, $\backslash\text{leftarrowtail}()$, $\backslash\text{leftarrowtriangle}()$, $\backslash\text{leftarrowx}()$, $\backslash\text{leftbkarrow}()$, $\backslash\text{leftcurvedarrow}()$, $\backslash\text{leftdasharrow}()$, $\backslash\text{leftdasharrowhead}()$, $\backslash\text{leftdbkarrow}()$, $\backslash\text{leftdbltail}()$, $\backslash\text{leftdotarrow}()$, $\backslash\text{leftdowncurvedarrow}()$, $\backslash\text{leftfishtail}()$, $\backslash\text{leftharpoondown}()$, $\backslash\text{leftharpoondownbar}()$, $\backslash\text{lefthpoonsupdown}()$, $\backslash\text{leftharpoonup}()$, $\backslash\text{leftharpoonupbar}()$, $\backslash\text{leftharpoonupdash}()$, $\backslash\text{leftleftarrows}()$, $\backslash\text{leftmoon}()$, $\backslash\text{leftouterjoin}()$, $\backslash\text{leftrightarrow}()$, $\backslash\text{leftrightarrowcircle}()$, $\backslash\text{leftrightharrows}()$, $\backslash\text{leftrightarrowtriangle}()$, $\backslash\text{leftrightharpoondowndown}()$, $\backslash\text{leftrightharpoonsupdownup}()$, $\backslash\text{leftrightharpoons}()$, $\backslash\text{leftrightharpoonsdown}()$, $\backslash\text{leftrightharpoonsup}()$, $\backslash\text{leftrightharpoonupdown}()$, $\backslash\text{leftrightharpoonupup}()$, $\backslash\text{leftrightsquigarrow}()$, $\backslash\text{leftsquigarrow}()$, $\backslash\text{leftsquigarrow}()$, $\backslash\text{lefttail}()$, $\backslash\text{leftthreearrows}()$, $\backslash\text{leftthreetimes}()$, $\backslash\text{leftwhitearrow}()$, $\backslash\text{leq}()$, $\backslash\text{leqq}()$, $\backslash\text{leqslant}()$, $\backslash\text{leqsslant}()$, $\backslash\text{leslant}()$, $\backslash\text{lescc}()$, $\backslash\text{lesdot}()$, $\backslash\text{lesdoto}()$, $\backslash\text{lesdotor}()$, $\backslash\text{lesges}()$, $\backslash\text{less}(<)$, $\backslash\text{lessapprox}()$, $\backslash\text{lessdot}()$, $\backslash\text{lesseqgtr}()$, $\backslash\text{lesseqqgr}()$, $\backslash\text{lessgtr}()$, $\backslash\text{lessim}()$, $\backslash\text{lfbowtie}()$, $\backslash\text{lfloor}()$, $\backslash\text{ltimes}()$, $\backslash\text{lgE}()$, $\backslash\text{gblkcircle}()$, $\backslash\text{gblksquare}()$, $\backslash\text{gwhtcircle}()$, $\backslash\text{gwhtsquare}()$, $\backslash\text{lhdc}()$, $\backslash\text{linefeed}()$, $\backslash\text{ll}()$, $\backslash\text{llangle}()$, $\backslash\text{llarc}()$, $\backslash\text{llblacktriangle}()$, $\backslash\text{llcorner}()$, $\backslash\text{lll}()$, $\backslash\text{lllnest}()$, $\backslash\text{llparenthesis}()$, $\backslash\text{lltriangle}()$, $\backslash\text{lmoustache}()$, $\backslash\text{lnapprox}()$, $\backslash\text{neq}()$, $\backslash\text{neqq}()$, $\backslash\text{nsim}()$, $\backslash\text{longdashv}()$, $\backslash\text{longdivision}()$, $\backslash\text{longleftarrow}()$, $\backslash\text{longleftrightarrow}()$, $\backslash\text{longleftsquigarrow}()$, $\backslash\text{longmapsfrom}()$, $\backslash\text{longmapsto}()$, $\backslash\text{longrightarrow}()$, $\backslash\text{longrightsquigarrow}()$, $\backslash\text{looparrowleft}()$, $\backslash\text{looparrowright}()$, $\backslash\text{lowint}()$, $\backslash\text{lozenge}()$, $\backslash\text{lozengeminus}()$, $\backslash\text{lparenextender}()$, $\backslash\text{lparenlend}()$, $\backslash\text{lparenless}()$, $\backslash\text{lparenuend}()$, $\backslash\text{lq}()$, $\backslash\text{lrarc}()$, $\backslash\text{rlblacktriangle}()$, $\backslash\text{rcorner}()$, $\backslash\text{rtriangle}()$, $\backslash\text{rtriangleeq}()$, $\backslash\text{sime}()$, $\backslash\text{simg}()$, $\backslash\text{sqhook}()$, $\backslash\text{tcc}()$, $\backslash\text{tcir}()$, $\backslash\text{times}()$, $\backslash\text{tlarr}()$, $\backslash\text{tquest}()$, $\backslash\text{trivb}()$, $\backslash\text{vboxline}()$, $\backslash\text{vzigzag}()$.

$\backslash\text{male}()$, $\backslash\text{maltese}()$, $\backslash\text{mapsdown}()$, $\backslash\text{mapsfrom}()$, $\backslash\text{mapsto}()$, $\backslash\text{mapsup}()$, $\backslash\text{mdblkdiamond}()$, $\backslash\text{mdblklozenge}()$, $\backslash\text{mdblkrci}()$, $\backslash\text{mdblksquare}()$, $\backslash\text{mdtblkcircle}()$, $\backslash\text{mdlblkdiamond}()$, $\backslash\text{mdlblklozenge}()$, $\backslash\text{mdlblksquare}()$, $\backslash\text{mdlwgwcircle}()$, $\backslash\text{mdlwgwdiamond}()$, $\backslash\text{mdlwgwtsquare}()$, $\backslash\text{mdsmblkcircle}()$, $\backslash\text{mdsmblksquare}()$, $\backslash\text{mdsmwhtcirc}()$, $\backslash\text{mdsmwhtsquare}()$, $\backslash\text{mdwhtcirc}()$, $\backslash\text{mdwhtdiamond}()$, $\backslash\text{mdwhtlozenge}()$, $\backslash\text{mdwhtsquare}()$, $\backslash\text{measangledtosw}()$, $\backslash\text{measangledrtose}()$, $\backslash\text{measangleldtosw}()$, $\backslash\text{measangleltonw}()$, $\backslash\text{measanglerdtose}()$, $\backslash\text{measanglerutone}()$, $\backslash\text{measangleltonw}()$, $\backslash\text{measangleurton}()$, $\backslash\text{measeq}()$, $\backslash\text{measuredangle}()$, $\backslash\text{measuredangleleft}()$, $\backslash\text{measuredrightangle}()$, $\backslash\text{medblackstar}()$, $\backslash\text{medmathspace}()$, $\backslash\text{medwhitestars}()$, $\backslash\text{mercury}()$, $\backslash\text{mho}()$, $\backslash\text{mid}()$,

\midbarvee (), \midbarwedge (), \midcir (), \minus (), \minusdot (), \minusfdots (), \minusrdots (), \mlcp (), \models (), \mp (), \mu (), \multimap (), \multimapinv ().

\nHdownarrow (), \nHuparrow (), \nLeftarrow (), \nLeftrightarrow (), \nRightarrow (), \nVDash (), \nVdash (), \nLeftarrow (), \nVleftarrowtail (), \nVleftrightarrow () , \nVrightarrow (), \nVrightarrowtail (), \nVtwoheadleftarrow (), \nVtwoheadleftarrowtail (), \nVtwoheadrightarrow (), \nVtwoheadrightarrowtail (), \nabla (), \napprox (), \nasmp (), \natural (), \ncong (), \neq (), \nearrow (), \neg (), \neovnarrow (), \neovsearrow (), \neptune (), \neq (), \nequiv (), \nesarrow (), \neuter (), \nexists (), \ng (), \ngeq (), \ngtr (), \ngtrless (), \ngtrsim (), \nhVvert (), \nhpar (), \ni (), \niobar (), \nis (), \nisd (), \nleftarrow (), \nleftrightarrow (), \nleq (), \unless (), \nlessgtr (), \nlessim (), \nmid (), \nni (), \nobreakhyphen (), \notin (), \nparallel (), \nparallelint (), \nprec (), \npreccurlyeq (), \nrightarrow (), \nsim (), \nsime (), \nsqsubseteq (), \nsupseteq (), \nsubset (), \nsubeteq (), \nsucc (), \nsucccurlyeq (), \nsupset (), \nsupseteq (), \ntriangleleft (), \ntrianglelefteq (), \ntriangleright (), \ntrianglerighteq (), \nu (), \nvDash (), \nvLeftarrow (), \nvLeftrightarrow (), \nvRightarrow (), \nvdash (), \nvinfy (), \nvleftarrow (), \nvleftarrowtail (), \nvleftrightarrow (), \nvrightarrow (), \nvrightarrowtail (), \nvtwoheadleftarrow (), \nvtwoheadleftarrowtail (), \nvtwoheadrightarrow (), \nvtwoheadrightarrowtail (), \nwarrow (), \nwarrowtail (), \nwarrownearrow (), \nwsearrow ().

\o (), \obar (), \obot (), \obrbrak (), \obslash (), \odiv (), \odot (), \odotslashdot (), \oe (), \ogreaterthan (), \ohorn (), \oiint (), \oint () , \ointccw (), \olcross (), \oldKoppa (), \oldkoppa (), \olesthan (), \omega (), \omicron (), \ominus (), \operp (), \oplus (), \opluslhrim (), \oplusrhrim (), \origof (), \oslash (), \otimes (), \otimeshat (), \otimeslhrim (), \otimesrhrim (), \overbrace (), \overbracket (), \overline (), \overparen (), \owns () .

\parallel (), \parallelogram (), \parallelogramblack (), \parsim (), \partial (), \partialslashdot (), \partialmeetc (), \pentagon (), \pentagonblack (), \perp (), \perpss (), \phi (), \phone (), \pi (), \pitchfork (), \plusdot (), \plusqq (), \plushat (), \plussim (), \plussubtwo (), \plustrif (), \pluto (), \pm (), \pointnt (), \postalmark (), \prec (), \precapprox (), \preccurlyeq (), \preceq (), \preceqq (), \precnapprox (), \precneq (), \precneqq (), \precnsim (), \precnsim (), \prime (), \prod (), \profalar (), \proffine (), \profsurf (), \proto (), \prurel (), \psi (), \pullback (), \pushout ().

\qprime (), \quarternote (), \questeq (), \quotdblbase (), \quotdblright (), \quotsinglbase (), \quotsinglright ().

\rAngle (), \rBrace (), \rBrack (), \rParen (), \rangle (), \rangledot (), \rangleldzgarrow (), \rasp (), \rbag (), \rlblkbrbrak (), \rbrace ({}), \rbracelend (), \rbracemid (), \rbraceuend (), \rbrack ([]), \rbrackextender (), \rbracklend (), \rbrackltick (), \rbrackubar (), \rbrackuend (), \rbrackurtick (), \rbrbrak (), \rceil (), \rcurvyangle (), \rdiagovfdiag (), \rdiagovsearrow (), \recorder (), \revangle (), \revangleubar (), \revemptyset (), \revnmid (), \rfbowtie (), \rfloor (), \rfloortimes (), \rhd (), \rho (), \righarrowbsimilar (), \rightangle (), \rightangledot (), \rightanglesqr (), \rightarrow (), \rightarrowapprox (), \rightarrowbackapprox (), \rightarrowbar (), \rightarrowdiamond (), \rightarrowgtr (), \rightarrowonoplus (), \rightarrowplus (), \rightarrowshortleftarrow (), \rightarrowsimilar (), \rightarrowsupset (), \rightarrowtail (), \rightarrowtriangle (), \rightarrowx (), \rightbkarrow (), \rightcurvedarrow (), \rightdasharrow (), \rightdbltail (), \rightdotarrow (), \rightdowncurvedarrow (), \rightfishtail (), \rightharpoondown (), \rightharpoondownbar (), \rightharpoonsupdown

(),
 \rightleftharpoons (),
 \rightleftharpoonsbar (),
 \rightleftharpoonsdash (),
 \rightthicksim (),
 \rightleftarrows (),
 \rightleftharpoons (),
 \rightleftharpoonsdown (),
 \rightleftharpoonsup (),
 \rightmoon (),
 \rightouterjoin (),
 \rightpentagon (),
 \rightpentagonblack (),
 \rightrightarrows (),
 \rightsquigarrow (),
 \rightsquigarrow (),
 \righttail (),
 \rightthreearrows (),
 \rightthreetimes (),
 \rightwhitearrow (),
 \ringplus (),
 \risingdotseq (),
 \rmoustache (),
 \rparenextender (),
 \rparengtr (),
 \rparenlend (),
 \rparenruend (),
 \rppolint (),
 \rq (),
 \rrangle (),
 \rrparenthesis (),
 \rsolbar (),
 \rsqhook (),
 \rsub (),
 \rtimes (),
 \rtriltri (),
 \ruledelayed (),
 \rvboxline (),
 \rvzigzag ()�

\sampi (),
 \sansLmirrored (),
 \sansLturned (),
 \saturn (),
 \scissors (),
 \scpolint (),
 \scrB (),
 \scrE (),
 \scrF (),
 \scrH (),
 \scrI (),
 \scrL (),
 \scrM (),
 \scrR (),
 \scre (),
 \scrg (),
 \scro (),
 \scurel (),
 \searrow (),
 \seovnearrow (),
 \setminus (),
 \setminus (),
 \sharp (),
 \shortdowntack (),
 \shortleftarrow (),
 \shortlefttack (),
 \shortrightarrow (),
 \shortrightarrow (),
 $\shortrightarrowleftarrow$ (),
 \shortuptack (),
 \shuffle (),
 σ (),
 \silon (),
 \silon (),
 \sim (),
 \simeq (),
 \simgE (),
 \simgtr (),
 \similarleftarrow (),
 \similarrightarrow (),
 \simlE (),
 \simless (),
 \simminussim (),
 \simneqq (),
 \simplus (),
 \simrdots (),
 \sinewave (),
 \slash (),
 \smallblacktriangleleft (),
 \smallblacktriangleright (),
 \smalldiamond (),
 \smallin (),
 \smallint (),
 \smalllni (),
 \smallsetminus (),
 \smalltriangleleft (),
 \smalltriangleright (),
 \smashtimes (),
 \smbldiamond (),
 \smblozenge (),
 \smbllksquare (),
 \smeparsl (),
 \smile (),
 \smiley (),
 \smt (),
 \smte (),
 \smwhitestar (),
 \smwhtcircle (),
 \smwhtlozenge (),
 \smwhtsquare (),
 \spadesuit (),
 \sphericalangle (),
 \sphericalangleup (),
 \sqcap (),
 \sqcup (),
 \sqint (),
 \sqlozenge (),
 \sqrt (),
 $\sqrt3$ (),
 $\sqrt4$ (),
 \sqrtbottom (),
 \sqsubset (),
 \sqsubseteq (),
 \sqsubsetneq (),
 \sqsupset (),
 \sqsupseteq (),
 \sqsupsetneq (),
 \squarecrossfill (),
 \squaregrayfill (),
 \squarehfill (),
 \squarehvfill (),
 \squareleftblack (),
 \squareleftblack (),
 \squarellblack (),
 \squarellquad (),
 \squarelrblack (),
 \squarelrquad (),
 \squareneswfill (),
 \squarenwsefill (),
 \squareulblack (),
 \squareulquad (),
 \squareurblack (),
 \squareurquad (),
 \squarevfill (),
 \squoval (),
 \ss (),
 \star (),
 \stareq (),
 \sterling (),
 \stigma (),
 \strns (),
 \subdot (),
 \submult (),
 \subrarr (),
 \subset (),
 \subsetapprox (),
 \subsetcirc (),
 \subsetdot (),
 \subseteq (),
 \subseteqq (),
 \subsetneq (),
 \subsetneqq (),
 \subsetplus (),
 \subsim (),
 \subsub (),
 \subsup (),
 \succ (),
 \succapprox (),
 \succcurlyeq (),
 \succeq (),
 \succeqq (),
 \succnapprox (),
 \succneq (),
 \succneqq (),
 \succnsim (),
 \succsim (),
 \sum (),
 \sumbotom (),
 \sumint (),
 \sumtop (),
 \sun (),
 \supdsub (),
 \supdot (),
 \suphsol (),
 \suphsub (),
 \suplarr (),
 \supmult (),
 \supn (),
 \supset (),
 \supsetapprox (),
 \supsetcirc (),
 \supsetdot (),
 \supseteq (),
 \supseteqq (),
 \supsetneq (),
 \supsetneqq (),
 \supsetplus (),
 \supsim (),
 \supsub (),
 \supsup (),
 \surd (),
 \swarrow ()

\talloblong (),
 \target (),
 τ (),
 τrus (),
 \testhookx (),
 \textAsterisks (),
 \textacute (),
 \textadvanced (),
 \textain (),
 \textasciacute (),
 \textasciicircum ($\hat{}$),
 \textasciidiereis (),
 \textasciigrave ($\acute{}$),
 \textasciimacron (),
 \textasciitilde ($\tilde{}$),
 \textasterisklow (),
 \textbackdprime (),
 \textbackprime (),
 \textbacktrprime (),
 \textbardotlessj (),
 \textbardotlessjvar (),
 \textbarglotstop (),
 \textbari (),
 \textbarl (),
 \textbaro (),
 \textbarrevglotstop (),
 \textbaru (),
 \textbeltl (),
 \textbenttailyogh (),
 \textbreve (),
 \textbrokenbar (),
 \textbullet (),
 \textbullseye (),
 \textcent (),
 \textcircledP (),
 \textclosepsilon (),
 \textcloseomega (),
 \textcloserevepsilon (),
 \textcopyright (),
 \textcrb (),
 \textcrh (),
 \textcrinvglotstop (),
 \textcrlambda (),
 \textcrtwo (),
 \textctc (),
 \textctd (),
 \textctesh (),
 \textctj (),
 \textctl (),
 \textctn (),
 \textctt (),
 \textctyogh (),
 \textctz (),
 \textcurrency (),
 \textdctzlig (),
 \textdegree (),
 \textdiscount (),
 \textdollar (\$),
 \textdotaccent (),
 \textdotlessj (),
 \textdoubleacute (),
 \textdoublebarpipe (),
 \textdoublepipe (),
 \textdtd ()

prime (), \textdptr (), \textdyoghlig (), \textdzlig (), \textepsilon (), \textesh (), \textestimated (), \textexclam (), \textexclamdown (), \textfishhookr (), \textflorin (), \textfranc (), \textgamma (), \textglotstop (), \textgrave (), \texthalflength (), \texthamza (), \textthen (), \textheng (), \texthooks (), \texthookz (), \texthtb (), \texthtc (), \texthtd (), \texthtg (), \textthth () , \textththeng (), \texthtk (), \texthttp (), \texthtq (), \texthtscg (), \texthtt (), \texthvlig (), \texthyphen (), \textinvglotstop (), \textinvscr (), \texttiota (), \textlengthmark (), \textlhalfing (), \textlhookd (), \textlhookk (), \textlhookl (), \textlhookt (), \textlhti (), \textlira (), \textlongegr (), \textlongy (), \textlongy (), \textlooptoprevesh (), \textlowacute (), \textlowered (), \textlowgrave (), \textlowmacron (), \textlptr (), \textltailm (), \textltailn (), \textltilde (), \textlyoghlig (), \textmacron (), \textmu (), \textnumero (), \textogonek (), \textohm (), \textonehalf (), \textonequarter (), \textonesuperior (), \textopeno (), \textordfeminine (), \textordmasculine (), \textovercross (), \textoz (), \textpertenthousand (), \textperthousand (), \textpesetas (), \textphi (), \textpipe (), \textprime (), \textprimstress (), \textqprime (), \textquestiondown (), \textquotedbl ("), \textquotedblleft (), \textquotedblright (), \textraised (), \textraiseglotstop (), \textraiserevglotstop (), \textramshorns (), \textrecipe (), \textreferencemark (), \textregistered (), \textretracted (), \textreve (), \textrevepsilon (), \textrevglotstop (), \textrhalfing (), \textrhookrevespsilon (), \textrhookschwa (), \textrhicity (), \textringaccent (), \textrptr (), \textrtaild (), \textrtaill (), \textrtailn (), \textrtailr (), \textrtails (), \textrtailt (), \textrtailz (), \textscsca (), \textscb (), \textscsce (), \textscg (), \textscscha (), \textscsi (), \textscsl (), \textscsn (), \textscsolig (), \textscsr (), \textscripta (), \textscriptg (), \textscriptv (), \textscu (), \textscy (), \textsecstress (), \textsemicolonreversed (), \textsilon (), \textsmalltilde (), \textstretchcvar (), \textsubw (w), \textsuph () , \textsuphth (), \textsupinvscr (), \textsupj (), \textsupr (), \textsupturnr (), \textsupturnrrtail (), \textsupw (), \textsupy (), \texttctlig (), \texttctlig (), \textthreequarters (), \textthreesuperior (), \texttrademark (), \texttrprime (), \texttslig (), \textturna (), \textturncomma (), \textturnh (), \textturnk (), \textturnlongegr (), \textturnnm (), \textturnmrleg (), \textturnr (), \textturnrrtail (), \textturnscripta (), \textturnt (), \textturnv (), \textturnw (), \textturny (), \texttwosuperior (), \textupsilon (), \textupr (), \textvibyi (), \textvisiblespace (), \textyogh (), \textth (), \texttherefore (), \textthermod (), \texttheta (), \textthickapprox (), \textthicksim (), \textthreeangle (), \textthreecolon (), \textconcat (), \textiefty (), \texttimes (), \texttimesbar (), \textminus (), \textto (), \textoea (), \texttona (), \textonebarextrahigh (), \textonebarextralow (), \textonebarhigh (), \textonebarlow (), \textonebarmid (), \texttop (), \texttopbot (), \textpcir (), \texttopfork (), \textsemicircle (), \textosa (), \textowa (), \textplus (), \texttrapezium (), \texttrianglecdot (), \texttriangledown (), \texttriangleexclam (), \texttriangleleft (), \texttriangleleftblack (), \texttrianglelefteq (), \texttriangleminus (), \texttriangleodot (), \texttriangleplus (), \texttriangleq (), \texttriangleright (), \texttrianglerightblack (), \texttrianglerighteq (), \texttriangles (), \texttriangleserifs (), \texttriangletimes (), \texttriangleubar (), \texttriangleup (), \texttrprime (), \textturnangle (), \textturnediota (), \textturnnot (), \texttwocaps (), \texttwocups (), \texttwoheaddownarrow (), \texttwoheadleftarrow (), \texttwoheadleftarrowtail (), \texttwoheadleftdbarrow (), \texttwoheadmapsfrom (), \texttwoheadmapsto (), \texttwoheadrightarrow (), \texttwoheadrightarrowtail (), \texttwoheaduparrow (), \texttwoheaduparrowcircle (), \texttwolowline (), \texttwonotes (), \texttypecolon () .

\ubrbrak (), \uhorn (), \ularc (), \ulblacktriangle (), \ulcorner (), \ulcrop (), \ultriangle (), \uminus (), \underbrace (), \underbracket (), \underparen (), \unlhd (), \unrhd (), \upand (), \uparrow (), \uparrowbarred (), \uparrowoncircle (), \updasharrow (), \updownarrow (), \updownarrowbar (), \updownarrows (), \updownharpoonleftleft (),

\updownharpoonleftright (), \updownharpoonrightleft (), \updownharpoonrightright (), \updownharpoonsleftright (), \upfishtail (), \upharpoonleft (), \upharpoonleftbar (), \upharpoonright (), \upharpoonrightbar (), \upharpoonsleftright (), \upin (), \upint (), \uplus (), \uprightcurvearrow (), \upuparrows (), \upwhitearrow (), \urarc (), \urblacktriangle (), \urcorner (), \urcrop (), \urtriangle ().

\v (), \vBar (), \vBarv (), \vDash (), \varTheta (), \varVdash (), \varbarwedge (), \varbeta (), \varclubsuit (), \vardiamondsuit (), \vardoublebarwedge (), \varepsilon (), \varheartsuit (), \varhexagon (), \varhexagonblack (), \varhexagonrbonds (), \varin (), \varisobar (), \varisins (), \varkappa (), \varltriangle (), \varni (), \varniorbar (), \varnis () , \varnothing (), \varointclockwise (), \varphi (), \varpi (), \varproto (), \varrho (), \varrowextender (), \varsigma (), \varspadesuit (), \varstar (), \vartheta (), \vartriangle (), \vartriangleleft (), \vartriangleright (), \varveebar (), \vbraceextender (), \vbrtri () , \vdash (), \vdots (), \vectimes (), \vee (), \veebar (), \veedot (), \veedoublebar (), \veeeq (), \veemidvert (), \veeodot (), \veeonvee (), \veeonwedge (), \vert (), \viewdata (), \vlongdash (), \vrectangle (), \vrectangleblack (), \vysmlblksquare (), \vysmlwhtsquare (), \vzigzag ().

\watchicon (), \wedge (), \wedgebar (), \wedgedot (), \wedgedoublebar (), \wedgemidvert (), \wedgegeodot (), \wedgeonwedge (), \wedgeq (), \whitearrowupfrombar (), \whiteinwhitetriangle (), \whitepointerleft (), \whitepointerright (), \whitesquaretickleft (), \whitesquaretickright (), \whthorzoval (), \whtvertoval (), \wideangledown (), \wideangleup (), \wp (), \wr ().

\xbsol (), \xi (), \xsol (), \yen (), \zeta (), \zpipe (),

IF ANYBODY WILL CHECK WHETHER ALL NAMES CORRESPOND TO RIGHT
TEX SYMBOLS I SHALL APPRECIATE IT GREATLY.

Appendix B 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. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, L^AT_EX 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	143
AddLight	117
Adjust	124
alpha	113
Alpha	116
alphadef	113
AlphaDef	116
ambient	113
Ambient	117
Area	144
Arrows	108
ArrowSize	118
ask	212
Aspect	126
Axial	154
Axis	123, 141
AxisStl	124

B

Ball	136
Barh	144
Bars	144
BarWidth	118
Beam	160
Belt	154
Box	141
BoxPlot	144
Boxs	154

C

call	213
Candle	144
Chart	144
chdir	212
ClearLegend	143
Clf	136
CloseGIF	134
Cloud	160
Color scheme	109
Colorbar	141
Column	198
ColumnPlot	126
Combine	136, 198
Cone	136
Cones	144
Cont	154
Cont3	160
ContD	154
ContF	154
ContF3	160
ContXYZ	171

CopyFont	119
CosFFT	201
CRange	122
Create	192
Crop	192
Crust	171
CTick	124
CumSum	201
Curve	136
cut	113
Cut	119
CutOff	119

D

defchr	212
define	212
defnum	212
defpal	213
Delete	192
Dens	154
Dens3	160
DensXYZ	171
Dew	167
Diff	201
Diff2	201
diffuse	113
Dots	171
Drop	136

E

else	213
elseif	213
EndFrame	134
endif	213
Envelop	201
Error	136, 144
Evaluate	198
Export	197
Extend	192

F

Face	136
FaceX	136
FaceY	136
FaceZ	136
Fall	154
fgets	139
Fill	194
Find	205
FindAny	206
Fit	177

Fit2.....	177	Mark style	108		
Fit3.....	177	MarkSize.....	118		
FitS.....	177	MathGL overview	1		
Fl_MathGL.....	182, 184	MathGL setup	116		
Flow.....	167	Max.....	198		
FlowP.....	167	Maximal	204		
Fog.....	118	Mesh.....	154		
Font	119	meshnum.....	113		
Font styles	110	MeshNum.....	118		
fontsize.....	113	Message.....	121		
for	213	mglColor	224		
FPlot.....	171	mglData	192		
FSurf.....	171	mglFitPnts	177		
func.....	213	mglFont	227		
G					
GetNumFrame	134	mglGLUT	182		
GetNx	204	mglGraph	116		
GetNy	204	mglParse	214		
GetNz.....	204	mglPoint	225		
GetWarn.....	121	mglWindow	182		
Glyph.....	136	Min	198		
Grad.....	154	Minimal	204		
Grid.....	141, 154	Mirror	201		
Grid3	160	Modify	194		
H					
Hankel	201	Momentum	198, 205		
Hist	179, 198	MPI_Recv	136		
I					
if.....	213	MPI_Send	136		
Import	197	MultiPlot	126		
InPlot	126	N			
Insert	192	NewFrame	134		
Integral	201	next.....	213		
L					
Label.....	139, 141, 144	Norm	201		
Last.....	205	NormSl	201		
legend.....	114	O			
Legend	143	once.....	213		
Light.....	117	Origin	122		
Line.....	136	P			
Line style	108	Palette	120		
Linear.....	203	Perspective	126		
Linear1.....	203	Pipe	167		
List	194	Plot	144		
LoadFont	119	Pop	126		
M					
Map	164	PrintInfo	204		
Mark.....	136, 144	Push	126		
QMATHGL	182, 186	Puts	139		
QuadPlot	171	PutsFit	177		
Putsw	139	Putsw	139		

R

Radar	144
Ranges	122
Read	197
ReadAll	197
ReadHDF	197
ReadMat	197
ReadRange	197
Rearrange	192
Region	144
ResetFrames	134
Resize	198
RestoreFont	119
return	213
Roll	201
Rotate	126
RotateN	126
RotateText	119

S

Save	197
SaveHDF	197
Set	194
SetAlphaDef	116
SetAmbient	117
SetArrowSize	118
SetAxisStl	124
SetBarWidth	118
SetCoor	123
SetCut	119
SetCutBox	119
SetFontDef	119
SetFontSize	119
SetFontSizeCM	119
SetFontSizeIN	119
SetFontSizePT	119
SetFunc	123
SetLegendBox	143
SetLegendMarks	143
SetMarkSize	118
SetMeshNum	118
SetOrigin	122
SetPalette	120
SetPlotId	118
SetRange	122
SetRanges	122
SetRotatedText	119
SetSize	129
SetTickLen	124
SetTickRotate	124
SetTicks	124
SetTickSkip	124
SetTicksVal	124
SetTickTempl	124
SetTickTime	124
SetTranspType	116
SetTuneTicks	124

SetWarn	121
Sew	201
ShowImage	130
SinFFT	201
Smooth	201
Sphere	136
Spline	203
Spline1	203
Squeeze	192
StartGIF	134
Stem	144
Step	144
STFA	164
StickPlot	126
stop	213
SubData	198
SubPlot	126
Sum	198
Surf	154
Surf3	160
Surf3A	164
Surf3C	164
SurfA	164
SurfC	164
Swap	201

T

Tape	144
Tens	144
Ternary	123
Text	139
TextMark	144
Textual formulas	112
TickLen	124
Tile	154
TileS	164
Title	126
Torus	144
Trace	198
Traj	167
Transpose	192
TranspType	116
TriCont	171
TriPlot	171
Tube	144

V

value	114
Var	194
Vect	167
View	126

W

widgets	6, 182, 184, 186
window	6, 182

Write	130
WriteBMP	130
WriteBPS	130
WriteEPS	130
WriteFrame	130
WriteGIF	130
WriteIDTF	130
WriteJPEG	130
WriteOBJ	130
WritePNG	130
WriteSVG	130
WriteTEX	130
WriteTGA	130
WriteWGL	130

X

xrange	113
XRange	122
XTick	124

Y

yrange	113
YRange	122
YTick	124

Z

zrange	113
ZRange	122
ZTick	124