The mglTEX package*

Diego Sejas Viscarra
diego.mathematician@gmail.com

November 20, 2014

Abstract

MathGL is a fast and efficient library by Alexey Balakin for the creation of high-quality publication-ready scientific graphics. Although it defines interfaces for many programming languages, it also implements its own programming language, called *MGL*, which can be used independently. With the package mglTEX, MGL scripts can be embedded within any LATEX document, and the corresponding images are automatically created and included.

This manual documents the use of the commands and environments of mglTEX.

1 Introduction

MathGL is a fast and efficient library by Alexey Balakin for the creation of high-quality publication-ready scientific graphics. It implements more than 50 different types of graphics for 1d, 2d and 3d large sets of data. It supports exporting images to bitmap formats (PNG, JPEG, BMP, etc.), or vector formats (EPS, TEX, SVG, etc.), or 3d image formats (STL, OBJ, XYZ, etc.), and even its own 3d format, MGLD. MathGL also defines its own vector font specification format, and supports UTF-16 encoding with TEX-like symbol parsing. It supports various kinds of transparency and lighting, textual formula evaluation, arbitrary curvilinear coordinate systems, loading of subroutines from .dll or .so libraries, and many other useful features.

MathGL has interfaces for a wide variety of programming languages, such as C/C++, Fortran, Python, Octave, Pascal, Forth, and many others, but it also defines its own scripting language, called *MGL*, which can be used to generate graphics independently of any programming language. The mglTEX package adds support to embed MGL code inside LATEX documents, which is automatically extracted and executed, and the resulting images are included in the document.

Besides the obvious advantage of having available all the useful features of MathGL, mglTEX facilitates the maintenance of your document, since both code for text and code for graphics are contained in a single file.

*This document corresponds to mglTEX v2.0, dated /2014/11/18.

2 Usage

The simplest way to load `mglTEX` to a LaTeX document is to write the command

```
\usepackage{mgltex}
```

in the preamble. Alternatively, one can pass a number of options to the package by means of the syntax

`\usepackage[`⟨*options list*⟩`]{mgltex}`,

where ⟨*options list*⟩ is a comma-separated list that can contains one or more of the following options:

- `draft`: The generated images won't be included in the document. This option is useful when fast compilation of the document is needed.
- `final`: This overrides the `draft` option.
- `on`: To create the MGL scripts and corresponding images of the document every time LaTeX is run.
- `off`: To avoid creating the MGL scripts and corresponding images of the document, but still try to include the images.
- `comments`: To allow the contents of the `mglcomment` environments to be shown in the LaTeX document.
- `nocomments`: To not show the contents of the `mglcomment` environments in the LaTeX document.
- `png`, `jpg`, `jpeg`: To export images to the corresponding bitmap format.
- `eps`, `epsz`: To export to uncompressed/compressed EPS format as primitives.
- `bps`, `bpsz`: To export to uncompressed/compressed EPS format as bitmap.
- `pdf`: To export to 3D PDF format.
- `tex`: To export to LaTeX/*tikz* document.

It must be noted that the options `on` and `off` are exclusive, in the sense that if one specifies both of them, only the last one will be used. Likewise, the options that specify the format to save the graphics are exclusive.

Observe the option `off` is useful to save compilation time of a document. For example, if the graphics of an article are in final version, instead of compilling them over and over again every time LaTeX runs, they can be created only once with the `on` option, and then only included (but not recompiled) with the `off` option.

The are two ways to compile a document with `mglTEX`: The first way is to run `latex --shell-escape` ⟨*document*⟩

twice, since the first run will extract the MGL code, execute it and include some of the resulting graphics, while the second run will include the remaining graphics; the second way is to run `latex` ⟨*document*⟩ to extract the MGL code, then execute the generated scripts with the program `mglconv` (which comes with MathGL), and execute `latex` ⟨*document*⟩ once more to include the graphics.

2.1 Environments for MGL code embedding

`mgl` The main environment defined by mglTEX is `mgl`. It extracts its contents to a general script, called ⟨*document*⟩.mgl, where ⟨*document*⟩ stands for the name of the LaTeX file being compiled; this script is compiled, and the corresponding image is included. Its syntax is:

```
\begin{mgl}[⟨key-val list⟩]
        ⟨MGL code⟩
\end{mgl}
```

Here, ⟨*key-val list*⟩ accepts the same optional arguments as the `\includegraphics` command from the graphicx package, plus an additional one, `imgext`, which can be used to specify the extension to save the graphic. The ⟨*MGL code*⟩ doesn't need to contain any specific instruction to create the image, since mglTEX takes care of that.

`mgladdon` This environment adds its contents to the general script ⟨*document*⟩.mgl, but it doesn't produce any image. It doesn't require any kind of arguments.

```
\begin{mgladdon}
   ⟨MGL code⟩
\end{mgladdon}
```

`mglcode` This is the same as the `mgl` environment, but the corresponding code is written *verbatim* to a separate script, whose name is specified as mandatory argument. It accepts the same optional arguments as `mgl`.

```
\begin{mglcode}[⟨key-val list⟩]{⟨script_name⟩}
              ⟨MGL code⟩
\end{mglcode}
```

`mglscript` The code within `mglscript` is written verbatim to a script whose name is specified as mandatory argument, but no image is produced. It is useful for creation of MGL scripts which can be later post-processed by another package, like listings.

```
\begin{mglscript}{⟨script_name⟩}
        ⟨MGL code⟩
\end{mglscript}
```

`mglfunc` This is used to define MGL functions within the general script ⟨*document*⟩.mgl. It takes one mandatory argument, which is the name of the function, plus one optional argument, which specifies the number of arguments of the function. The environment needs to contain only the body of the function, since the lines "func ⟨*function_name*⟩ ⟨*number of arguments*⟩" and "return" are appended automatically at the beginning and the end, respectively. The resulting code is written at the end of the general script, after the `stop` command, which is also written automatically.

```
\begin{mglfunc}[⟨number of arguments⟩]{⟨function_name⟩}
        ⟨MGL function body⟩
\end{mglfunc}
```

`mglcommon` This is used to create a common "setup" script that will be executed together with each of the other scripts. It is useful to define constants, parameters, etc. that will be available to every script.

```
\begin{mglcommon}
   ⟨MGL code⟩
\end{mglcommon}
```

For example, one could make

```
\begin{mglcommon}
define gravity 9.81 # [m/s^2]
\end{mglcommon}
```

to make the constant *gravity* available to every script.

Observe this environment should be used only to define constants, parameters and things like that, but not graphical objects like axis or grids, because every image created with the `mgl` environment clears every graphical object before creating the image.[1]

`mglsignature` This environment is used to declare a signature (or commentary) that will be included at the beginning of every script generated by mglTEX. It is verbatim-like environment, so no LATEX cammand will be executed, but copied literally. However, the default signature is "This script was generated from ⟨*document*⟩.mgl on date ⟨*today*⟩".

[1]This problem occurs only with the `mgl` environment, so you could use `mglcommon` to create many graphics with the same axis, grid, etc., with environments like `mglcode`, but in that case the best option is to use the `mglsetup` environment together with the `\mglplot` command.

```
\begin{mglsignature}
⟨Signature for MGL scripts⟩
\end{mglsignature}
```

`mglcomment` This environment is used to embed commentaries in the LaTeX document. The commentary won't appear in the case of the user passing the option `nocomments` to the package, but it will be written *verbatim* is the user passes the option `comments`.

```
\begin{mglcomment}
⟨Commentary⟩
\end{mglcomment}
```

In the case of the user allowing commentaries, this will result in the appearance of the following commentary in the LaTeX document:

```
<------------------ MGL comment ------------------>
                   ⟨Commentary⟩
<------------------ MGL comment ------------------>
```

2.2 Fast creation of graphics

mglTEX defines a convenient way to work with many graphics that have exactly the same settings (for example, same angles of rotation, same type of grid, etc.): instead of writing repetitive code every time it's needed, it can be stored in memory with the `mglsetup` environment, and then can be used when needed with the `\mglplot` command.

`mglsetup` This environment stores its contents in memory for later use. It accepts one optional argument, which is a keyword (name) to be associated to the corresponding block of code, so different blocks of code can be stored with different names.

```
\begin{mglsetup}[⟨keyword⟩]
        ⟨MGL code⟩
\end{mglsetup}
```

`\mglplot` This command is used for fast generation of graphics with default settings, and can be used in parallel with the `mglsetup` environment. It accepts one mandatory argument which consists of MGL instructions, separated by the symbol ":", which can span through various text lines. It also accepts the same optional arguments as the `mgl` environment, plus an additional one, called `settings`, which can be used to specify a keyword used in a `mglsetup` environment. If the `settings` option is specified, the code in the mandatory argument will be appended to the block of code of the corresponding `mglsetup` environment.

```
\mglplot[⟨key-val list⟩]{⟨MGL code⟩}
```

2.3 Verbatim-like environments

`mglblock` It writes its contents *verbatim* to a file, whose name is given as mandatory argument, and then it also typesets its contents on the LaTeX document, numbering each line of code.

```
\begin{mglblock}{⟨script_name⟩}
        ⟨MGL code⟩
\end{mglblock}
```

`mglverbatim` It typesets its contents to the LaTeX document, numbering each line of code.

```
\begin{mglverbatim}
     ⟨MGL code⟩
\end{mglverbatim}
```

2.4 Working with external scripts

In case of having MGL scripts in their own files, mglTEX can work with them without needing to transcript them to the LaTeX document.

`\mglgraphics` This command takes one mandatory argument, which is the name of an external MGL script, which will be automatically executed, and the resulting image will be included. The same optional arguments as the `mgl` environment are accepted.

```
\mglgraphics[⟨key-val list⟩]{⟨script_name⟩}
```

`\mglinclude` This command takes one mandatory argument, which is the name of an external MGL script, which will be automatically transcript *verbatim* on the LaTeX document, and each line of code will be numerated.

```
\mglinclude{⟨script_name⟩}
```

2.5 Additional commands

`\mgldir` This command can be used to specify where mglTEX should create the MGL scripts and corresponding images. This is useful, for example, to avoid a lot of scripts and images from polluting the current directory.

```
\mgldir{⟨directory⟩}
```

This command must be used in the preamble of the document, since the first MGL script is created at the moment of the `\begin{document}` command; trying to use it somewhere else will issue an error. On the other hand, it is the responsibility of the user to create the ⟨*directory*⟩, since mglTEX won't do it automatically.

`\mglquality` This command can be used to specify the quality for the graphics created with mglTEX. An info message specifying the characteristics of the chosen quality is printed in the .log file.

```
\mglquality{⟨quality⟩}
```

The available qualities are described below:

Quality	Description
0	No face drawing (fastest)
1	No color interpolation (fast)
2	High quality (normal)
3	High quality with 3d primitives (not implemented yet)
4	No face drawing, direct bitmap drawing (low memory usage)
5	No color interpolation, direct bitmap drawing (low memory usage)
6	High quality, direct bitmap drawing (low memory usage)
7	High quality with 3d primitives, direct bitmap drawing (not implemented yet)
8	Draw dots instead of primitives (extremely fast)

`\mgltexon` This command has the same effect as the package option `on`, i.e., create all the scripts and corresponding graphics, but its effect is local, meaning that it work only from the point it is used on.

```
\mgltexon
```

`\mgltexoff` This command has the same effect as the package option `off`, i.e., DO NOT create the scripts and corresponding graphics, and include images anyway, but its effect is also local, meaning that it work only from the point it is used on.

```
\mgltexoff
```

Observe the commands `\mgltexon` and `\mgltexoff` can be used to save compilation time of a document. For example, when writing an article, if the graphics of the first section are already in final version, instead of compilling them every time LaTeX is called, they can be created only once, and then the section can be wrapped with `mgltexoff` and `mgltexon`, so the graphics do not get recompiled again (wasting time), but only included.

`\mglcomments` This command has the same effect as the package option `comments`, i.e., show all the commentaries contained int the `mglcomment` environments, but its effect is local, meaning that it work only from the point it is used on.

```
\mglcoments
```

`\mglnocomments` This command has the same effect as the package option `nocomments`, i.e., DO NOT show the contentsof the `mglcomment` environments, but its effect is also local, meaning that it work only from the point it is used on.

```
\mglnocomments
```

Observe the commands `\mglcomments` and `\mglnocomments` can be used to activate/deactivate commentaries on the document: just like LaTeX commentaries, but with the possibilty of making them visible/invisible. This feature could be used, for example, to show remainders or commentaries for readers of test versions of an article.

`\mglTeX` This command just pretty-prints the name of the package.

```
\mglTeX
```

2.6 User-definable macros

There are two macros that the user is allowed to modify:

`\mgltexsignature` As an alternative to the `mglsignature` environment for declaring signatures, the user can manually redefine the signature macro `\mgltexsignature`, according to the following rules:

- The positions of the comment signs for the MGL language have to be manually specified in the signature using the `\mglcomm` macro.
- The new-line character is declared as "`^^J`".
- A percent sign (`%`) has to be added at the end of every physical line of `\mgltexsignature`, otherwise an inelegant space at the beginning of every line will appear.
- Any LaTeX command can be used in this case.

For example, the default signature:

#
# This script was generated from ⟨*document*⟩.mgl on date ⟨*today*⟩
#

can be achieved with

```
\def\mgltexsignature{%
  \mglcomm^^J%
  \mglcomm\ This script was generated from \jobname.mgl on date \today^^J%
  \mglcomm%
}
```

`\mglcommonscript` It is the name for the common script that takes the contents of the `mglcommon` environment. For example, the default name of the script ("mgl_common_script") is defined by doing

```
\def\mglcommonscript{mgl_common_script}
```

2.7 Behavior of mglTEX

As a convenient feature, the environments `mglcode`, `mglscript` and `mglblock` will automatically check if they are being used to create different scripts with the same name, in which case mglTEX will issue a warning; however, if one of these environments overwrite an external script (not embedded in the document), it won't be noticed. Likewise, the user will be warned if the environment `mglfunc` is being used to create different MGL functions with the same name.

When mglTEX is unable to find a graphic that is supposed to include, instead of producing an error, it will warn the user about it, and will display a box in the corresponding position of the document, like the following one:

MGL
image
not
found

Notice that the first time LATEX is executed, many of these boxes will appear in the document because the graphics from the MGL scripts are created, but not all are included (until LATEX is run for the second time).

3 Warning for the user

mglTEX assummes that the `\begin{`⟨*environment*⟩`}` and `\end{`⟨*environment*⟩`}` commands will occupy their own physical line of LATEX code. So the correct form to use the environments is the following:

```
\begin{<environment>}
  <contents of the environment>
\end{<environment>}
```

The following forms of use could cause problems:

```
\begin{<environment>}<contents of the environment>\end{<environment>}
```

```
\begin{<environment>}<contents of the environment>
\end{<environment>}
```

```
\begin{<environment>}
<contents of the environment>
\end{<environment>}<text>
```

One of the reasons for this is that some of the environments in mglTEX are programmed to ignore the empty space following the `\begin{`⟨*environment*⟩`}`, which would cause an inelegant empty line in the script, so the first two incorrect forms would cause mglTEX to ignore a complete line of code. The other reason is the method used to detect the `\end{`⟨*environment*⟩`}` command, which could fail in the case of the third incorrect use.

4 Implementation

This section documents the implementation of mglTEX. Its purpose is to facilitate the comprehension and maintenance of the package.

4.1 Initialization

The keyval package is loaded to facilitate the declaration of ⟨*key*⟩=⟨*value*⟩ options for commands and environments; the graphicx package is loaded in order to manipulate and include the images created by MGL code.

```

\RequirePackage{keyval}
\RequirePackage{graphicx}
```

We declare the options of the package. The first two are `draft` and `final`, which are passed directly to the graphicx package.

```

\DeclareOption{draft}{%
  \PassOptionsToPackage{\CurrentOption}{graphicx}%
}
\DeclareOption{final}{%
  \PassOptionsToPackage{\CurrentOption}{graphicx}%
}
```

The next two options are `on` and `off`, where `on` indicates mglTEX to create every script and every corresponding image every time LATEX is executed, while `off` tells not to do it, but to include the images anyway. First we declare a flag (boolean variable) `\@mgltex@on@` to know if the used passed the `on` or the `off` option.

```
\newif\if@mgltex@on@
```

If the user passes the option `on`, `\@mgltex@on@` is true, and the command `\mgl@write` (which takes care of writing code to the scripts) is the normal LATEX `\immediate\write` commands;

```
\DeclareOption{on}{%
  \@mgltex@on@true%
  \def\mgl@write#1#2{%
    \immediate\write#1{#2}%
  }
}
```

if the user passes the option `off`, `\@mgltex@on@` is false, and the command `\mgl@write` does nothing (doesn't write to scripts).

```
\DeclareOption{off}{%
  \@mgltex@on@false%
  \def\mgl@write#1#2{}%
}
```

The next options are `comments` and `nocomments`, where `comments` indicates mglTEX to show the comments included inside `\mglcomments` environments, while `nocomments` tells not to do it. First we create a flag that will indicate which of these options is passed by the user.

```
\newif\if@mgl@comments@
```

If the user passes the option `comments`, `\@mgl@comments@` is true, and the `\mglcomments` environments print their contents;

```
\DeclareOption{comments}{%
  \@mgl@comments@true%
}
```

if the user passes the option `nocomments`, `\@mgl@comments@` is false, and the `\mglcomments` environments won't print their contents.

```
\DeclareOption{nocomments}{%
  \@mgl@comments@false%
}
```

We then indicate the supported extensions to save the images created by the package, and the corresponding package options. The chosen extension is stored in the `\mgl@image@ext` macro for future use.

```

\DeclareGraphicsExtensions{%
  .png,.eps,.jpg,.jpeg,.bps,.pdf,.epsz,.eps.gz,.bpsz,.bps.gz,.gif%
}

\DeclareOption{jpg}{\def\mgl@image@ext{.jpg}}
\DeclareOption{jpeg}{\def\mgl@image@ext{.jpeg}}
\DeclareOption{pdf}{\def\mgl@image@ext{.pdf}}
\DeclareOption{png}{\def\mgl@image@ext{.png}}
\DeclareOption{eps}{\def\mgl@image@ext{.eps}}
\DeclareOption{epsz}{\def\mgl@image@ext{.eps.gz}}
\DeclareOption{bps}{\def\mgl@image@ext{.bps}}
\DeclareOption{bpsz}{\def\mgl@image@ext{.bps.gz}}
\DeclareOption{gif}{\def\mgl@image@ext{.gif}}

\DeclareOption{tex}{\def\mgl@image@ext{.tex}}
```

Other options produce an error message.

```
\DeclareOption*{\@unknownoptionerror}
```

The default options for the package are set to `final` and `eps`, then the options passed by the user are processed.

```

```

```
\ExecuteOptions{final,on,nocomments,eps}
\ProcessOptions*
```

Declare the ⟨*key*⟩=⟨*value*⟩ pairs for the `mgl` environment and companions. The pairs corresponding to the `\includegraphics` command are repeated, and saved in the `\graph@keys` macro; the new option is `imgext`, which can be used to overwrite the default extension chosen for the package. Notice that `imgext` can be any supported extension by MathGL but, of course, not all of them are supported by LaTeX.

```

\define@key{mgl@keys}{bb}{\g@addto@macro{\graph@keys}{bb=#1,}}
\define@key{mgl@keys}{bbllx}{\g@addto@macro{\graph@keys}{bbllx=#1,}}
\define@key{mgl@keys}{bblly}{\g@addto@macro{\graph@keys}{bblly=#1,}}
\define@key{mgl@keys}{bburx}{\g@addto@macro{\graph@keys}{bburx=#1,}}
\define@key{mgl@keys}{bbury}{\g@addto@macro{\graph@keys}{bbury=#1,}}
\define@key{mgl@keys}{natwidth}{\g@addto@macro{\graph@keys}{natwidth=#1,}}
\define@key{mgl@keys}{natheight}{\g@addto@macro{\graph@keys}{natheight=#1,}}
\define@key{mgl@keys}{hiresbb}{\g@addto@macro{\graph@keys}{hiresbb=#1,}}
\define@key{mgl@keys}{viewport}{\g@addto@macro{\graph@keys}{viewport=#1,}}
\define@key{mgl@keys}{trim}{\g@addto@macro{\graph@keys}{trim=#1,}}
\define@key{mgl@keys}{angle}{\g@addto@macro{\graph@keys}{angle=#1,}}
\define@key{mgl@keys}{origin}{\g@addto@macro{\graph@keys}{origin=#1,}}
\define@key{mgl@keys}{width}{\g@addto@macro{\graph@keys}{width=#1,}}
\define@key{mgl@keys}{height}{\g@addto@macro{\graph@keys}{height=#1,}}
\define@key{mgl@keys}{totalheight}{\g@addto@macro{\graph@keys}{totalheight=#1,}}
\define@key{mgl@keys}{keepaspectratio}{\g@addto@macro{\graph@keys}{keepaspectratio=#1,}}
\define@key{mgl@keys}{scale}{\g@addto@macro{\graph@keys}{scale=#1,}}
\define@key{mgl@keys}{clip}[true]{\g@addto@macro{\graph@keys}{clip=#1,}}
\define@key{mgl@keys}{draft}[false]{\g@addto@macro{\graph@keys}{draft=#1,}}
\define@key{mgl@keys}{type}{\g@addto@macro{\graph@keys}{type=#1,}}
\define@key{mgl@keys}{ext}{\g@addto@macro{\graph@keys}{ext=#1,}}
\define@key{mgl@keys}{read}{\g@addto@macro{\graph@keys}{read=#1,}}
\define@key{mgl@keys}{command}{\g@addto@macro{\graph@keys}{command=#1,}}
\define@key{mgl@keys}{imgext}{\def\mgl@image@ext{.#1}}
```

We do the same for the `\mglplot` command. The options for the `\includegraphics` command are repeated and stored in the `\graph@keys` macro; the new options are `imgext`, which is the same as the one for the `mgl` environment, and `setup`, which is used to specify a keyword associated to a block of MGL code stored by the `mglsetup` environment.

```

\define@key{mglplot@keys}{bb}{\g@addto@macro{\graph@keys}{bb=#1,}}
\define@key{mglplot@keys}{bbllx}{\g@addto@macro{\graph@keys}{bbllx=#1,}}
\define@key{mglplot@keys}{bblly}{\g@addto@macro{\graph@keys}{bblly=#1,}}
\define@key{mglplot@keys}{bburx}{\g@addto@macro{\graph@keys}{bburx=#1,}}
\define@key{mglplot@keys}{bbury}{\g@addto@macro{\graph@keys}{bbury=#1,}}
\define@key{mglplot@keys}{natwidth}{\g@addto@macro{\graph@keys}{natwidth=#1,}}
\define@key{mglplot@keys}{natheight}{\g@addto@macro{\graph@keys}{natheight=#1,}}
\define@key{mglplot@keys}{hiresbb}{\g@addto@macro{\graph@keys}{hiresbb=#1,}}
\define@key{mglplot@keys}{viewport}{\g@addto@macro{\graph@keys}{viewport=#1,}}
```

```
\define@key{mglplot@keys}{trim}{\g@addto@macro{\graph@keys}{trim=#1,}}
\define@key{mglplot@keys}{angle}{\g@addto@macro{\graph@keys}{angle=#1,}}
\define@key{mglplot@keys}{origin}{\g@addto@macro{\graph@keys}{origin=#1,}}
\define@key{mglplot@keys}{width}{\g@addto@macro{\graph@keys}{width=#1,}}
\define@key{mglplot@keys}{height}{\g@addto@macro{\graph@keys}{height=#1,}}
\define@key{mglplot@keys}{totalheight}{\g@addto@macro{\graph@keys}{totalheight=#1,}}
\define@key{mglplot@keys}{keepaspectratio}{\g@addto@macro{\graph@keys}{keepaspectratio=#1,}}
\define@key{mglplot@keys}{scale}{\g@addto@macro{\graph@keys}{scale=#1,}}
\define@key{mglplot@keys}{clip}[true]{\g@addto@macro{\graph@keys}{clip=#1,}}
\define@key{mglplot@keys}{draft}[false]{\g@addto@macro{\graph@keys}{draft=#1,}}
\define@key{mglplot@keys}{type}{\g@addto@macro{\graph@keys}{type=#1,}}
\define@key{mglplot@keys}{ext}{\g@addto@macro{\graph@keys}{ext=#1,}}
\define@key{mglplot@keys}{read}{\g@addto@macro{\graph@keys}{read=#1,}}
\define@key{mglplot@keys}{command}{\g@addto@macro{\graph@keys}{command=#1,}}
\define@key{mglplot@keys}{imgext}{\def\mglplot@image@ext{.#1}}
\define@key{mglplot@keys}{setup}{\def\mglplot@setup{#1}}
```

A special extension for images created with MathGL is ".tex", so we store it within a macro for future use.

```

\def\TeX@ext{.tex}
```

4.2 Environments for MGL code embedding

\mgl@include@image This is the command that will include graphics created by MGL code. We can't use \includegraphics directly for two reasons: first, MathGL has the capacity of creating graphics with LaTeX commands (with the aid of the tikz package), in which case there is no image, but a ".tex" file, which has to be included; the second reason is that \includegraphics issues an error when the specified image doesn't exist, and remember that the first LaTeX run only creates the images at the end of the document, but they cannot be included yet, so there would be a lot of errors in the process of compilation.

```
\def\mgl@include@image#1{%
```

If the extension of the graphics is ".tex",

```
  \ifx\mgl@image@ext\TeX@ext%
```

first check if the file exists;

```
    \IfFileExists{#1.tex}{%
```

if so, include it,

```
      \include{#1}%
    }{%
```

otherwise use the command \mgl@img@not@found to create a warning.

```
      \mgl@img@not@found{#1}%
    }%
```

If the extension of the graphics is not ".tex",

```
  \else%
```

we define the next action to be performed as warning that requested image doesn't exist. This is stored in the `\next@action` macro, and will be overwriten if the image is found.

```
  \def\next@action{\mgl@img@not@found{#1}}%
```

For every extension supported by mglTEX,

```
    \@for\img@ext:=\Gin@extensions\do{%
```

if the file with the current extension exists,

```
      \IfFileExists{#1\img@ext}{%
```

overwrite the `\next@action` macro so it uses the `\includegraphics` command to include the image, otherwise do nothing.

```
        \def\next@action{%
          \expandafter\includegraphics\expandafter[\graph@keys]{#1}%
        }%
      }{}%
    }%
```

Execute `\next@action`.

```
    \next@action%
  \fi%
}
```

`\mgl@img@not@found` When this command is called with the name of a MGL image as argument, it issues a package warning indicating that the MGL image can't be found, and creates the following box in the corresponding position:

$$\boxed{\begin{array}{c}\textbf{MGL}\\ \textbf{image}\\ \textbf{not}\\ \textbf{found}\end{array}}$$

```
\def\mgl@img@not@found#1{%
  \PackageWarning{mgltex}{MGL image "#1" not found}%
  \framebox[10em]{%
    \centering%
    \bfseries\Huge%
    \vbox{MGL\\image\\not\\found}%
  }%
}
```

`mgl` This environment writes its contents to the main script ⟨*document*⟩.mgl.

First, declare a counter for numeration and naming of the images created from the main script ⟨*document*⟩.mgl.

```

\newcounter{mgl@image@no}
```

Create an output stream for the main script ⟨*document*⟩.mgl.

```

\newwrite\mgl@script
```

Open the main script at the beginning of the document (at the moment of the `\begin{document}` command).

```
\AtBeginDocument{%
  \if@mgltex@on@%
    \immediate\openout\mgl@script="\mgl@dir\jobname.mgl"%
    \mglsignature@write\mgl@script%
  \fi%
}
```

At the end of the document (at the moment of the `\end{document}` command):

```
\AtEndDocument{%
```

write an empty line on the main script (just for elegance),

```
  \mgl@write\mgl@script{}%
```

write the MGL *stop* command to stop the MathGL compiler.

```
  \mgl@write\mgl@script{stop}%
```

The `\mgl@func` is a buffer that contains instructions to write MGL functions declared with `mglfunc` environment. Here, we execute those instructions.

```
  \mgl@func%
```

Close the main script.

```
  \immediate\closeout\mgl@script%
```

Use the program `mglconv` (part of MathGL) to compile the main script.

```
  \mgl@write{18}{mglconv -n "\mgl@dir\jobname.mgl"}%
}

```

`\mgl` The beginning of the `mgl` environment.

```

\newcommand\mgl[1][]{%
```

First, process the ⟨*key*⟩=⟨*value*⟩ options for the environment.

```
  \def\graph@keys{}%
  \setkeys{mgl@keys}{#1}%
```

Now, make every "special" character (\, $, etc.) of category 13 (other), i.e., make them common characters.

```
  \let\do\@makeother \dospecials%
```

Add an end-line character at the end of every read line. This end-line character is declared active (category 12).

```
  \endlinechar`\^^M \catcode`\^^M\active%
```

Spaces characters are category 10; the spaces at the beginning of every read line are ignored.

```
  \catcode`\ =10%
```

Finally, the command that reads/writes each line of the contents of the environment is called.

```
  \mgl@write\mgl@script{quality \mgl@quality}%
  \expandafter\mgl@write@line%
}
```

`\end@mgl` Define a macro that contains the `\end{mgl}` command as text, so the end of the environment can be tested by comparison with it. From now on, we adopt the convention that the macro `\end@`⟨*environment*⟩ contains the `\end{`⟨*environment*⟩`}` command as text.

```
\begingroup%
  \escapechar=-1 \relax%
  \xdef\end@mgl{\string\\end\string\{mgl\string\}}%
\endgroup
```

`\mgl@write@line` This command reads each line from the `mgl` environment and writes it to the general script ⟨*document*⟩.mgl. We start by wrapping the new command with a LaTeX group because we will change the code of the end-line character to "active" *locally*, so we can indicate `\mgl@write@line` that its argument stretches until the end of the line.

```
\begingroup%
```

Declare the end-line character as active.

```
  \catcode`\^^M\active%
```

The command `\mgl@write@line` reads its argument until it finds the end-line character, i.e., it reads a complete line of text, which is MGL code in this case.

```
  \gdef\mgl@write@line#1^^M{%
```

The next action to be performed is write the read line of code to the main script ⟨*document*⟩.mgl and recursively call `\mgl@write@line`, so it reads the next line of text. These instructions are stored in the `\next@action` macro.

```
    \def\next@action{%
      \mgl@write\mgl@script{#1}%
      \mgl@write@line%
    }%
```

The `\test@end@mgl` command test if the end of the `mgl` environment has been reached in the current line. If so, it overwrites the `\next@action` macro so it doesn't read the next line of text, but executes the `\end{mgl}` command (see bellow).

```
    \test@end@mgl{#1}%
```

Execute the `\next@action` macro.

```
    \next@action%
  }%
\endgroup
```

`\test@end@mgl` This command checks if its argument is equal to `\end@mgl`; if so, overwrites the `\next@action` macro (see above) so that it executes the end of the `mgl` environment (`\end{mgl}`). Here, we adopt another convention: the `\test@end@`⟨*environment*⟩ checks if its argument is equal to `\end@`⟨*environment*⟩, i.e., tests whether the `\end{`⟨*environment*⟩`}` command has been reached, in which case, it executes that command.

```
\def\test@end@mgl#1{%
  \edef\this@line{#1}%
  \ifx\this@line\end@mgl%
    \def\next@action{\end{mgl}}%
  \fi%
}
```

`\endmgl` The end of the environment is quite simple: the `mgl@image@no` counter is increased by one, then the MGL command to save the corresponding image is written; the name given to the image is "⟨*document*⟩-mgl-⟨*mgl@image@no*⟩.⟨*mgl@image@ext*⟩"; the MGL *reset* command is written in the main script to clean the image and restart graphic parameters for the following image to be created. Finally, the `\mgl@include@image` command (see below) is called to include the image created.

```
\def\endmgl{%
  \stepcounter{mgl@image@no}%
  \mgl@write\mgl@script{%
    write '\mgl@dir\jobname-mgl-\arabic{mgl@image@no}\mgl@image@ext'%
  }%
  \mgl@write\mgl@script{reset}%
  \mgl@write\mgl@script{}%
  \mgl@include@image{\mgl@dir\jobname-mgl-\arabic{mgl@image@no}}%
}
```

`mgladdon` This is just a modification of the `mgl` environment. First, we define the `\end@mgladdon` to contain the `\end{mgladdon}` command as text as specified above, then we redefined `\test@end@mgl` command to check for the end of the `mgladdon` environment instead of `mgl`, finally we call the `\mgl` command with no options. The end of `mgladdon` is defined to do nothing.

```

\bgroup%
  \escapechar=-1\relax%
  \xdef\end@mgladdon{\string\\end\string\{mgladdon\string\}}%
\egroup%
\newenvironment{mgladdon}{%
  \def\test@end@mgl##1{%
    \edef\this@line{##1}%
    \ifx\this@line\end@mgladdon%
      \def\next@action{\end{mgladdon}}%
    \fi%
  }%
  \mgl[]%
}{}
```

mglcode This is like `mgl`, but it writes its contents to its own file, whose name is passed as mandatory argument.

\mgl@script@written The names of all the scripts written from the LaTeX document will be stored in this macro, so we can later check if some script is being overwritten. This macro will be used in other environments.

```
\def\mgl@script@written{}
```

\mgl@out@stream Declare an output stream for MGL scripts other than the main one. This stream will be used in other environments.

```
\newwrite\mgl@out@stream
```

\mglcode The beginning of the `mglcode` environment.

```
\newcommand\mglcode[2][]{%
  \def\graph@keys{}%
```

Process the ⟨*key*⟩=⟨*value*⟩ options. These are the same for the `mgl` environment.

```
  \setkeys{mgl@keys}{#1}%
```

Test if a script with the same name is already created from the LaTeX document. If so, a warning is issue, but we proceed anyway.

```
  \test@mgl@script@written{#2}%
```

Add the script's name to the `\mgl@script@written` macro.

```
  \xdef\mgl@script@written{\mgl@script@written#2,}%
```

Open the script for writing.

```
  \def\this@script{#2}%
  \if@mgltex@on@%
    \immediate\openout\mgl@out@stream=\mgl@dir\this@script.mgl%
    \mglsignature@write\mgl@out@stream%
  \fi%
```

Here, we do the same changes of categories as in the `mgl` environment, except for the spaces, which in this case will be respected, even the ones at the beginning of each like, i.e., we will write each line *verbatim.*

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \obeyspaces%
```

Call the command that will write each line of the contents of the environment.

```
  \expandafter\mglcode@write@line%
}
```

\test@mgl@script@written The macro that checks is we are overwriting any script.

```
\def\test@mgl@script@written#1{%
```

For every script already written (whose name is stored in `\mgl@script@written`), check if the current script's name matches; if so, issue a warning telling we are overwriting, but proceed.

```
  \edef\this@script{#1}%
  \@for\mgl@script@name:=\mgl@script@written\do{%
    \ifx\this@script\mgl@script@name%
      \PackageWarning{mgltex}{Overwriting MGL script "\this@script.mgl"}%
    \fi%
  }%
}
```

`\mglcode@write@line` This writes each line of the contents of the `mglcode` environment. However, contrary to the case of the `\mgl@write@line` command, it doesn't read line by line, but character by character, and stores each word in `\mgl@word` and each line in `\mgl@line`.

```
\newtoks\mgl@word
\newtoks\mgl@line
\def\mglcode@write@line#1{%
```

The next action (stored as `\next@action`) is to read the following character, unless overwritten later.

```
  \let\next@action\mglcode@write@line%
```

If the current character is an end-line character,

```
  \expandafter\if#1\^^M%
```

write the contents of `\mgl@line`, i.e., the current line, and clean `\mgl@word` and `\mgl@line`;

```
    \mgl@write\mgl@out@stream{\the\mgl@line}%
    \mgl@word{}%
    \mgl@line{}%
```

if the current character is a space, clean `\mgl@word`, but add the space to `\mgl@line`;

```
  \else\expandafter\if#1\space%
    \mgl@word{}%
    \mgl@line\expandafter{\the\mgl@line#1}%
```

otherwise, the current character is alphanumeric and is added both to `\mgl@word` and `\mgl@line`, and

```
  \else%
    \mgl@word\expandafter{\the\mgl@word#1}%
    \mgl@line\expandafter{\the\mgl@line#1}%
```

we test if the current word (`\mgl@word`) is `\end{mglcode}`, in which case, `\next@action` is overwritten to `\end{mglcode}`.

```
    \test@end@mglcode{\the\mgl@word}%
  \fi\fi%
```

Finally, execute `\next@action`.

```
  \next@action%
}
```

`\test@end@mglcode` The `\test@end@mglcode` checks if it's argument is equal to `\end@mglcode`, in which case overwrites `\next@action` to `\end{mglcode}`.

```
\begingroup%
  \escapechar=-1\relax%
  \xdef\end@mglcode{\string\\end\string\{mglcode\string\}}%
\endgroup%
\def\test@end@mglcode#1{%
  \edef\this@word{#1}%
  \ifx\this@word\end@mglcode%
    \def\next@action{\end{mglcode}}%
  \fi%
}
```

`\endmglcode` The end of the `mglcode` environment. It closes the output stream `\mgl@out@stream`, and calls the mglconv program (part of MathGL) to execute the script. Finally, the `\mgl@include@image` command is used to include the image created.

```
\def\endmglcode{%
  \immediate\closeout\mgl@out@stream%
  \mgl@write{18}{%
    mglconv "\mgl@dir\this@script.mgl" -s "\mgl@dir\mglcommonscript.mgl" -o "\mgl@dir\this@scrip
  }%
  \mgl@include@image{\mgl@dir\this@script}%
}
```

`mglscript` This is just a modification of the `mglcode` environment. First, we define the `\end@mglscript` macro; then we modify the `\test@end@mglcode` to check for `\end{mglscript}` instead of `\end{mglcode}`; finally, we call the `\mglcode` macro with the same mandatory argument as `mglscript`. The `\end{mglscript}` just closes the output stream `\mgl@out@stream`, but doesn't create nor includes any image.

```

\bgroup%
  \escapechar=-1\relax%
  \xdef\end@mglscript{\string\\end\string\{mglscript\string\}}%
\egroup%
\newenvironment{mglscript}[1]{%
  \def\test@end@mglcode##1{%
    \edef\this@word{##1}%
    \ifx\this@word\end@mglscript%
      \def\next@action{\end{mglscript}}%
    \fi%
  }%
  \mglcode{#1}%
}{%
  \immediate\closeout\mgl@out@stream%
}
```

`mglfunc` This environment is used to create MGL functions in the main script ⟨*document*⟩.mgl.

`\mglfunc@defined` Within this macro we will store the names of the MGL functions already defined from the LaTeX document, so that we can check if we are overwriting one of them

```

\def\mglfunc@defined{}
```

`\mgl@func` This is a buffer to store the instructions to write the MGL functions code when the `\end{document}` command is called. This is done this way, because the functions have to be after the *stop* command from the MGL language, which stops the execution of the MGL compiler, so no code should be after the *stop*, except for functions.

```
\def\mgl@func{}
```

`\mglgunc` The beginning of the `mglfunc` environment.

```

\newcommand\mglfunc[2][0]{%
```

First, check if a function with the current name is already defined, in which case we issue a warning, but proceed anyway.

```
  \test@mglfunc@defined{#2}%
```

Add the name of the current function to the list of functions defined.

```
  \g@addto@macro{\mglfunc@defined}{#2,}%
```

Here we do the same changes of categories as in the `mgl` environment.

```
  \let\do\@makeother \dospecials%
  \endlinechar‘\^^M \catcode‘\^^M\active%
  \catcode‘\ =10%
```

Write an empty line in the main script just for elegance (and to visually separate different functions, too).

```
  \g@addto@macro{\mgl@func}{\mgl@write\mgl@script{}}%
```

Write the heading of the function.

```
  \g@addto@macro{\mgl@func}{\mgl@write\mgl@script{func ’#2’ #1}}%
```

Call the command that will write each line of the contents of the environment.

```
  \expandafter\mglfunc@write@line%
}
```

`\test@mglfunc@defined` This command tests if a function with a given name—given as argument—is already defined from the LaTeX document; if so, a warning will be issued indicating multiple definitions for the same function, but we will proceed anyway.

```
\def\test@mglfunc@defined#1{%
  \def\this@func{#1}%
  \@for\mglfunc@name:=\mglfunc@defined\do{%
    \ifx\this@func\mglfunc@name%
      \PackageWarning{\mgl@name}{MGL function "#1" has multiple definitions}%
    \fi%
  }%
}
```

We declare *locally* the end-line character as active.

```
\begingroup%
  \catcode`\^^M\active%
```

\mglfunc@write@line

This is the command that reads each line of code of the `mglfunc` environment, and stores in the buffer `\mgl@func` the instructions to write each of these lines.

```
  \gdef\mglfunc@write@line#1^^M{%
```

The next action (`\next@action`) is to store in the buffer the instruction to write the current line, and then call recursively the `\mglfunc@write@line` command, unless overwritten below.

```
    \def\next@action{%
      \g@addto@macro{\mgl@func}{\mgl@write\mgl@script{#1}}%
      \expandafter\mglfunc@write@line%
    }%
```

Check for the end of the `mglfunc` environment, in which case, `\next@action` is redefined to be `\end{mglfunc}`.

```
    \test@end@mglfunc{#1}%
```

Execute `\next@action`.

```
    \next@action%
  }%

\endgroup
```

\end@mglfunc
\test@end@mglfunc

By now, we already know now these two commands work.

```
\begingroup%
  \escapechar=-1 \relax%
  \xdef\end@mglfunc{\string\\end\string\{mglfunc\string\}}%
\endgroup
\def\test@end@mglfunc#1{%
  \edef\this@line{#1}%
  \ifx\this@line\end@mglfunc%
    \def\next@action{\end{mglfunc}}%
  \fi%
}
```

\endmglfunc

Just stores in the buffer the instruction that closes the MGL function with the *return* command.

```
\def\endmglfunc{%
  \g@addto@macro{\mgl@func}{\mgl@write\mgl@script{return}}%
}

% \begin{environment}{mglcommon}
% Writes its contents to a common script that will be executed together with each of the other
% \begin{macro}{\mglcommonscript}
% \changes{v2.0}{2014/11/20}{Add \texttt{\backslash{}mglcommonscript} user-definable macro}
% We define a macro to store the name of the setup script that will contain common code to all
%    \begin{macrocode}
```

```
\def\mglcommonscript{mgl_common_script}
```

\end@mglcommon We already know the purpose of this macro.

```
\bgroup%
  \escapechar=-1\relax%
  \xdef\end@mglcommon{\string\\end\string\{mglcommon\string\}}%
\egroup%
```

The `mglcommon` environment redefines the `\test@end@mglcode` so it detects the `\end{mglcommon}` command instead, and uses the `\mglcode` to create the common script.

```
\newenvironment{mglcommon}{%
  \def\test@end@mglcode##1{%
    \edef\this@word{##1}%
    \ifx\this@word\end@mglcommon%
      \def\next@action{\end{mglcommon}}%
    \fi%
  }%
  \mglcode{\mglcommonscript}%
}{%
  \mgl@write\mgl@out@stream{quality \mgl@quality}%
  \immediate\closeout\mgl@out@stream%
}
```

This environment can be used only in the preamble.

```
\@onlypreamble\mglcommon
```

mglsignature This environment is used to declare signature text that will be written as comment on every script generated by mglTEX.

\mglcomm We store the comment sign for MGL in this macro. For that, we need to declare *locally* the symbol "#" as one of category 12.

```
\bgroup
  \catcode`#=12
  \gdef\mglcomm{#}
\egroup
```

\mgltexsignature The buffer where the signature will be stored. Here, we declare a default signature.

```
\def\mgltexsignature{%
  \mglcomm^^J%
  \mglcomm\space This file was autogenerated from the document \jobname.tex on date \today^^J%
  \mglcomm%
}
```

\mglsignature The beginning of the `mglsignature` environment.

```
\newcommand\mglsignature{%
```

Delete `\mgltexsignature` contents.

```
  \def\mgltexsignature{}%
```

We do the same changes of category as in the `mglcode` environment.

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \@vobeyspaces%
```

Call the command that will store each line of the signature in the `\mgltexsignature` macro.

```
  \expandafter\mglsignature@write@line%
}
```

`\end@mglsignature` We already know the purpose of this command.

```
\begingroup%
  \escapechar=-1 \relax%
  \xdef\end@mglsignature{\string\\end\string\{mglsignature\string\}}%
\endgroup
```

`\mglsignature@write@line` This command stores each line of the signature in the `\mgltexsignature` buffer.

```
\begingroup%
%   \catcode`\\=0%
  \catcode`\^^M\active%
  \gdef\mglsignature@write@line#1^^M{%
```

Unless overwritten later, the next action (`\next@action`) is to store the current line of the signature in the `\mgltexsignature` buffer, ending with a new-line character, and call `\mglsignature@write@line` recursively.

```
    \def\next@action{%
      \g@addto@macro{\mgltexsignature}{\mglcomm\space#1^^J}
      \mglsignature@write@line%
    }%
```

We check if the current line is `\end{mglsignature}`, in which case, overwrite `\next@action` to that command.

```
    \test@end@mglsignature{#1}%
```

Execute `\next@action`.

```
    \next@action%
  }%
\endgroup
```

`\test@end@mglsignature` We already know the purpose of this command.

```
\def\test@end@mglsignature#1{%
  \edef\this@line{#1}%
  \ifx\this@line\end@mglsignature%
    \def\next@action{\end{mglsignature}}%
  \fi%
}
```

`\endmglsignature` The end of the `mglsignature` environment. It just adds a comment sign to `\mgltexsignature` for elegance.

```
\def\endmglsignature{%
```

```
  \g@addto@macro{\mgltexsignature}{\mglcomm}
}
```

`\mglsignature@write` It takes care of writing the signature to the output stream which is passed as its argument.

```
\def\mglsignature@write#1{\mgl@write#1{\mgltexsignature}}
```

`mglcomment` An environment to contain multiline comments that won't be printed to the document nor to any script in the case of the user passes the option `nocomments` to the package, and it'll print the comments if the `comments` option is passed to the package.

`\mglcomment` The beginning of the `mglcomment` environment. Here, we change categories of special characters (like #, , etc.) and indicate to obey lines and spaces.

```

\def\mglcomment{%
  \let\do\@makeother\dospecials%
  \obeylines%
  \@vobeyspaces%
  \verbatim@font%
  \small%
```

Call the command that will ignore all the commentary.

```
  \mgl@comment%
}
```

`\mgl@comment` This command reads everything up to the `\end{mglcomment}` and ignores it if the `nocomments` option is passed to the package, or prints it otherwise. (We use the trick to consider everything up to the `\end{mglcomment}` the argument of `\mgl@comment`.)

```
\begingroup%
```

We do some adequate changes of code locally, so that \, { and } are special, and |, `[` and `]` take their functions, respectively.

```
  \catcode`|=0\catcode`[= 1\catcode`]=2\catcode`\{=12\catcode`\}=12\catcode`\\=12%
```

Define `\mgl@comment` to do nothing with its argument if the `nocomments` option has been passed to the package; otherwise, if the `comments` options has been passed, it will print the commentary, with delimiters to indicate where it starts and where it ends. Then call the end of the environment.

```
  |gdef|mgl@comment#1\end{mglcomment}[%
    |if@mgl@comments@%
      |begin[center]%
        <------------------ MGL comment ------------------>%
        #1%
        <------------------ MGL comment ------------------>%
      |end[center]%
    |fi%
    |end[mglcomment]]%
|endgroup%
```

`\endmglcomment` The end of the environment; it does nothing.

```
\def\endmglcomment{}
```

4.3 Fast creation of graphics

`mglsetup` This environment is used to store lines of code that need to be repeated many times. Later, the `\mglplot` command (see below) uses this lines of code without the need to repeat them.

`\mglsetup@defined` A macro to list the names of all the setups already defined.

```

\def\mglsetup@defined{}
```

`\mglsetup` The beginning of the `mglsetup` environment. It accepts one optional argument, which is a name (keyword) to be associated to the block of code.

```
\newcommand\mglsetup[1][generic]{%
```

Test if there already exists a setup with the current name; if so, issue a warning of redefinition of the setup, but proceed anyway.

```
  \test@mglsetup@defined{#1}%
```

Add the name of the current setup to `\mglsetup@defined`.

```
  \g@addto@macro{\mglsetup@defined}{#1,}%
```

Define a new buffer which will contain the instructions to write the contents of the environment when the `\mglplot`. command is used. If the `mglsetup` environment is called like `\mglsetup\oarg{\meta{keyword}}`, the buffer will be called `\mgl@setup@\meta{keyword}`; if no name is given, use “generic” as keyword.

```
  \expandafter\def\csname mgl@setup@#1\endcsname{\mgl@write\mgl@script{}}%
  \expandafter\def\csname mgl@setup@#1\endcsname{\mgl@write\mgl@script{quality \mgl@quality}}%
```

Here, we do the same changes of category for special characters as we did in the `mgl` environment.

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \catcode`\ =10%
```

Call the command that will store in the buffer the instructions to write the lines of MGL code.

```
  \expandafter\mglsetup@write@line%
}
```

`\test@mglsetup@defined` For every name stored in `\mglsetup@defined`, check if its argument (the name of the current setup) matches, in which case we will issue a warning, but proceed.

```
\def\test@mglsetup@defined#1{%
  \def\this@setup{#1}%
  \@for\mglsetup@name:=\mglsetup@defined\do{%
    \ifx\this@mglsetup\mglsetup@name%
      \PackageWarning{\mgl@name}{Redefining "#1" setup for \noexpand\mglplot}%
    \fi%
  }%
}
```

\mglsetup@write@line This works exactly as the \mgl@write@line, but instead of writing directly to a script, it stores the writing instructions in the buffer.

```
\begingroup%
  \catcode`\^^M\active%
  \gdef\mglsetup@write@line#1^^M{%
    \def\next@action{%
      \expandafter\g@addto@macro\csname mgl@setup@\this@setup\endcsname{%
        \mgl@write\mgl@script{#1}%
      }%
      \expandafter\mglsetup@write@line%
    }%
    \test@end@mglsetup{#1}%
    \next@action%
  }%
\endgroup
```

\end@mglsetup \test@end@mglsetup We already know how these two macros work

```
\begingroup%
  \escapechar=-1 \relax%
  \xdef\end@mglsetup{\string\\end\string\{mglsetup\string\}}%
\endgroup
\def\test@end@mglsetup#1{%
  \edef\this@line{#1}%
  \ifx\this@line\end@mglsetup%
    \def\next@action{\end{mglsetup}}%
  \fi%
}
```

\endmglsetup The end of the mglsetup environment. It does nothing.

```
\def\endmglsetup{}
```

\mglplot This macro uses the blocks of code stored by mglsetup environments to complete the code contained in its mandatory argument.

If there is an optional argument, make \@mglplot process it, otherwise pass no argument to \@mglplot.

```

\def\mglplot{%
  \@ifnextchar[{\@mglplot}{\@mglplot[]}%
}
```

\@mglplot This command receives one mandatory argument, but enclosed between brackets; so it receives the optional argument of \mglplot.

```
\def\@mglplot[#1]{%
```

Unless overwritten by the user with the setup=\meta{setup} option, the default setup is “generic”; initialize the \graph@keys macro; process the ⟨*key*⟩=⟨*value*⟩

pairs passed by the user; increase the counter `mgl@image@no` for numbering and naming of images.

```
  \def\mglplot@setup{generic}%
  \def\graph@keys{}%
  \setkeys{mglplot@keys}{#1}%
  \stepcounter{mgl@image@no}%
```

If the given setup is undefined, issue a package error; otherwise, execute the buffer of the setup, which will write the contents of the corresponding `mglsetup` blocks to the general script.

```
  \ifx\csname mgl@setup@\mglplot@setup\endcsname\@undefined%
    \PackageError{\mgl@name}{Setup "\mglplot@setup" undefined}{}%
  \else%
    \csname mgl@setup@\mglplot@setup\endcsname%
  \fi%
```

Call `\@@mglplot` (see below).

```
  \@@mglplot%
}
```

`\@@mglplot` This command writes its argument verbatim to the main script, then writes the command to save the corresponding image, and the *reset* command to prepare MathGL for the next image; finally, it uses the `\mgl@include@image` to include the corresponding graphics in the document.

```
\long\def\@@mglplot#1{%
  \mgl@write\mgl@script{\detokenize{#1}}%
  \mgl@write\mgl@script{%
    write '\mgl@dir\jobname-mgl-\arabic{mgl@image@no}\mgl@image@ext'%
  }%
  \mgl@write\mgl@script{reset}%
  \mgl@include@image{\mgl@dir\jobname-mgl-\arabic{mgl@image@no}}%
}
```

4.4 Verbatim-like environments

`mgl@verb@line@no` We create a counter to number the lines of code in verbatim-like environments.

```

\newcounter{mgl@verb@line@no}
```

`mglverbatim` This environment writes its contents *verbatim* to the LaTeX document, numbering each line of code.

`\mglverbatim` The beginning of the `mglverbatim` environment.

```

\def\mglverbatim{%
```

Initialize the counter for lines of code.

```
  \setcounter{mgl@verb@line@no}{0}%
```

We use the list environment to set the numeration of the lines of code that will be written to the LaTeX document as items of the list. We also set the separation between lines of code, the indentation of the line, and some other length parameters.

```
  \list{\itshape\footnotesize\arabic{mgl@verb@line@no}.}{}%
  \setlength{\labelsep}{1em}%
  \itemsep\z@skip%
  \leftskip\z@skip\rightskip\z@skip%
  \parindent\z@\parfillskip\@flushglue\parskip\z@skip%
```

We do the same changes of categories as in the `mglcode` environment.

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \obeyspaces%
```

use verbatim font.

```
  \verbatim@font%
```

Call the command that will write each line of the contents of the environment.

```
  \expandafter\mglverbatim@ignore@line%
}
```

`\mglverbatim@ignore@line` This command ignores the first line of the `verbatim` environment, which is an empty line.

```
\def\mglverbatim@ignore@line#1{%
  \expandafter\mglverbatim@write@line%
}
```

`\mglverbatim@write@line` Reads the contents of the `mglverbatim` character by character, and stores words in the `\mgl@word` buffer and lines in the `\mgl@line` buffer, just like the `mglcode` environment did.

```
\def\mglverbatim@write@line#1{%
```

Unless overwritten later, the next action (`\next@action`) is recursively call `\mglverbatim@write@line`.

```
  \let\next@action\mglverbatim@write@line%
```

If the character read is an end-line character,

```
  \expandafter\if#1\^^M%
```

increase the line of code counter, write the line contained in `\mgl@line` as an item of the `list` environment, and clean `\mgl@word` and `\mgl@line`;

```
    \stepcounter{mgl@verb@line@no}%
    \item\mbox{\the\mgl@line}%
    \mgl@word{}%
    \mgl@line{}%
```

if the character is a space, clean `\mgl@wors`, but add the space to `\mgl@line`;

```
  \else\expandafter\if#1\space%
    \mgl@word{}%
    \mgl@line\expandafter{\the\mgl@line#1}%
```

otherwise, the character is aphanumeric, so add it to the `\mgl@word` and `\mgl@line` buffers, and check if `\mgl@word` is `\end{mglverbatim}`, in which case overwrite `\next@action` to be that command.

```
  \else%
    \mgl@word\expandafter{\the\mgl@word#1}%
    \mgl@line\expandafter{\the\mgl@line#1}%
    \test@end@mglverbatim{\the\mgl@word}%
  \fi\fi%
  \next@action%
}
```

`\end@mglverbatim` `\test@end@mglverbatim` We already know the purpose of these macros.

```
\begingroup%
  \escapechar=-1\relax%
  \xdef\end@mglverbatim{\string\\end\string\{mglverbatim\string\}}%
\endgroup%
\def\test@end@mglverbatim#1{%
  \edef\this@word{#1}%
  \ifx\this@word\end@mglverbatim%
    \def\next@action{\end{mglverbatim}}%
  \fi%
}
```

`\endmglverbaim` The end of the `mglverbatim` environment. It just closes the `list` environment.

```
\def\endmglverbatim{\endlist}
```

`mglblock` This environment writes its contents to a script, whose name is passed as mandatory argument, ad then it also writes its contents to the LaTeX document, numbering each line.

`\mglblock` The beginning of the `mglblock environment`.

```

\def\mglblock#1{%
```

Check if the script already exists, in which case we issue a warning, but proceed anyway.

```
  \test@mgl@script@written{#1}%
```

Add the name of the script to the list of scripts written.

```
  \xdef\mgl@script@written{\mgl@script@written#1,}%
```

We make the same changes of categories as in the `mglcode` environment.

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \obeyspaces%
```

Open the output stream for the current script.

```
  \def\this@script{#1}%
  \if@mgltex@on@%
    \immediate\openout\mgl@out@stream="\mgl@dir\this@script.mgl"%
    \mglsignature@write\mgl@out@stream%
  \fi%
```

Call the command that will write each line of the contents of the environment.

```
  \expandafter\mglblock@write@line%
}
```

\mglblock@write@line

This macro reads characater by character the code inside `mglblock`, and uses the `\mgl@word` and `\mgl@line` buffers to store words and lines of codes, just like we did with the `mglcode` environment.

```
\def\mglblock@write@line#1{%
```

The next action (`\next@action`) is set to recursively call `\mglblock@write@line`, unless it is overwritten later.

```
  \let\next@action\mglblock@write@line%
```

If the read character is an end-line character, write the contents of `\mgl@line` to the script, and the clean `\mgl@word` and `\mgl@line`;

```
  \expandafter\if#1\^^M%
    \mgl@write\mgl@out@stream{\the\mgl@line}%
    \mgl@word{}%
    \mgl@line{}%
```

if the read character if a space, clean `\mgl@word`, but add the space to `\mgl@line`;

```
  \else\expandafter\if#1\space%
    \mgl@word{}%
    \mgl@line\expandafter{\the\mgl@line#1}%
```

otherwise, the character is alphnumeric, and should be added to `\mgl@word` and `\mgl@line`, and we test if `\mgl@word` is `\end{mglblock}`, in which case, we overwrite `\next@action` to that command.

```
  \else%
    \mgl@word\expandafter{\the\mgl@word#1}%
    \mgl@line\expandafter{\the\mgl@line#1}%
    \test@end@mglblock{\the\mgl@word}%
  \fi\fi%
```

Execute `\next@action`.

```
  \next@action%
}
```

\end@mglblock
\test@end@mglblock

We already know the purpose of these macros.

```
\begingroup%
  \escapechar=-1\relax%
  \xdef\end@mglblock{\string\\end\string\{mglblock\string\}}%
\endgroup%
\def\test@end@mglblock#1{%
  \edef\this@word{#1}%
  \ifx\this@word\end@mglblock%
    \def\next@action{\end{mglblock}}%
  \fi%
}
```

`\mgl@in@stream` We create an input stream to read from MGL scripts.

```
\newread\mgl@in@stream
```

`\endmglblock` The end of the `mglblock` environment.

```
\def\endmglblock{%
```

Close the output stream.

```
  \immediate\closeout\mgl@out@stream%
```

Open the input stream.

```
  \immediate\openin\mgl@in@stream="\mgl@dir\this@script.mgl"%
```

Here, we use the `list` environment to set the numeration of the lines of code that will be written to the LaTeX document as items of the list. We also set the separation between lines of code, the indentation of the line, and some other lenght parameters.

```
  \begingroup%
  \list{\itshape\footnotesize\arabic{mgl@verb@line@no}.}{}%
  \setlength{\labelsep}{1em}%
  \itemsep\z@skip%
  \leftskip\z@skip\rightskip\z@skip%
  \parindent\z@\parfillskip\@flushglue\parskip\z@skip%
```

Use the verbatim font, and obey spaces, including spaces at the beggining of the line.

```
  \verbatim@font%
  \@vobeyspaces%
```

Call the command that will write the lines of code to the LaTeX document.

```
  \mglblock@read@line%
}
```

`\mglblock@read@line` This command reads lines of code from the input stream and writes them as items of the `list` environment.

```
\def\mglblock@read@line{%
```

Increase the line counter.

```
  \stepcounter{mgl@verb@line@no}%
```

Read a line from the input stream.

```
  \read\mgl@in@stream to \this@line%
```

If the end of file has been reached, define `\next@action` to close the input stream, and en the `list` environment;

```
  \ifeof\mgl@in@stream%
    \def\next@action{%
      \immediate\closein\mgl@in@stream%
      \endlist%
      \endgroup%
    }%
```

otherwise, `\next@action` is write the read line as an item of the `list` environment, and recursively call `\mglblock@read@line`.

```
  \else%
    \def\next@action{%
      \item\mbox{\this@line}%
      \mglblock@read@line%
    }%
  \fi%
```

Execute `\next@action`.

```
  \next@action%
}
```

4.5 Working with external scripts

`\mglgraphics` This command allows to generate and include graphics from a external (not embedded) script.

```

\newcommand\mglgraphics[2][]{%
```

Initialize `\graph@keys`, which will contain the ⟨*key*⟩=⟨*value*⟩ options for the `\includegraphics`command.

```
  \def\graph@keys{}%
```

Process the ⟨*key*⟩=⟨*value*⟩ options passed by the user.

```
  \setkeys{mgl@keys}{#1}%
```

Execute the program `mglconv` (included in MathGL) to compile the corresponding script.

```
  \mgl@write{18}{mglconv "\mgl@dir#2.mgl" -s "\mgl@dir\mglcommonscript.mgl" -o "\mgl@dir#2\mgl@
```

Include the generated image with the `\mgl@include@image` command.

```
  \mgl@include@image{\mgl@dir#2}%
}
```

`\mglinclude` This command copies verbatim the contents of an external script, and numerates each line of code.

```

\def\mglinclude#1{%
```

Initialize the line counter.

```
  \setcounter{mgl@verb@line@no}{0}%
```

Open the script in the input stream.

```
  \immediate\openin\mgl@in@stream="\mgl@dir#1.mgl"%
```

Here, we use the `list` environment to numerate each line of code as an item. We also set some length parameters.

```
  \begingroup%
  \list{\itshape\footnotesize\arabic{mgl@verb@line@no}.}{}%
  \setlength{\labelsep}{1em}%
  \itemsep\z@skip%
  \leftskip\z@skip\rightskip\z@skip%
  \parindent\z@\parfillskip\@flushglue\parskip\z@skip%
```

We do the same changes of category as in the `mglcode` environment, and set the font to verbatim font.

```
  \let\do\@makeother \dospecials%
  \endlinechar`\^^M \catcode`\^^M\active%
  \@vobeyspaces%
  \verbatim@font%
```

We (re)use the `\mglblock@read@line` command to numerate and write each line of code.

```
  \mglblock@read@line%
}
```

4.6 Additional commands

`\mgldir` A command to specify a directory to write the scripts and create the images. First, we create a macro that will store the specified directory for later use.

```

\def\mgl@dir{}
```

The command `\mgldir` is the only way to modify `\mgl@dir`. This is done so the user won't be able to modify the default directory, dangerously altering the internal behavior of the package.

```
\def\mgldir#1{%
  \def\mgl@dir{#1}%
}
```

Declare `\mgldir` so that it can only be used in the preamble. This is because the main script ⟨*document*⟩.mgl is opened at the moment of the `\begin{document}` instruction.

```
\@onlypreamble\mgldir
```

`\mgl@quality` We define a macro to store the quality.

```
\def\mgl@quality{2}
```

`\mglquality` This is used to define the quality for MGL graphics.

```
\def\mglquality#1{%
```

Write the quality command to a setup script.

```
  \def\mgl@quality{#1}%
  \if@mgltex@on@%
    \immediate\openout\mgl@out@stream="\mgl@dir\mglcommonscript.mgl"%
    \mgl@write\mgl@out@stream{quality #1}%
    \immediate\closeout\mgl@out@stream%
```

Print an info message about the corresponding quality, or a warning if the quality doesn't exist.

```
    \ifcase#1
```

```
      \PackageInfo{mgltex}{Quality 0: No face drawing (fastest)}%
    \or%
      \PackageInfo{mgltex}{Quality 1: No color interpolation (fast)}%
    \or%
      \PackageInfo{mgltex}{Quality 2: High quality (normal)}%
    \or%
      \PackageInfo{mgltex}{Quality 3: High quality with 3d primitives (not implemented yet)}%
    \or%
      \PackageInfo{mgltex}{Quality 4: No face drawing, direct bitmap drawing (low memory usage)
    \or%
      \PackageInfo{mgltex}{Quality 5: No color interpolation, direct bitmap drawing (low memory
    \or%
      \PackageInfo{mgltex}{Quality 6: High quality, direct bitmap drawing (low memory usage)}%
    \or%
      \PackageInfo{mgltex}{Quality 7: High quality with 3d primitives, direct bitmap drawing (n
    \or%
      \PackageInfo{mgltex}{Quality 8: Draw dots instead of primitives (extremely fast)}%
    \else%
      \PackageWarning{mgltex}{Quality #1 not available. Using default (2)}%
    \fi%
  \else%
    \PackageWarning{mgltex}{mglTeX is off, quality changes won’t have effect}%
  \fi%
}
```

`\mgltexon` Has the same effect as the package option `on`, but its effect is local, meaning that works only from the point this command is called on.

```

\def\mgltexon{
  \@mgltex@on@true
  \def\mgl@write##1##2{%
    \immediate\write##1{##2}%
  }
}
```

`\mgltexoff` Has the same effect as the package option `off`, but its effect is local.

```
\def\mgltexoff{%
  \@mgltex@on@false
  \def\mgl@write##1##2{}%
}
```

`\mglcomments` Has the same effect as the package option `comments`, but its effect is local, meaning that works only from the point this command is called on.

```

\def\mglcomments{
  \@mgl@comments@true
}
```

`\mglnocomments` Has the same effect as the package option `off`, but its effect is local.

```
\def\mglnocomments{%
  \@mgl@comments@false
}
```

`\mglTeX` Just pretty-prints the name of the package.

```

\def\mglTeX{mgl\TeX}
```

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.