Nikhef 2015-025

Manual for Axodraw Version 2

John C. Collins[a] and J.A.M. Vermaseren[b]

[a] *Department of Physics, Pennsylvania State University,*
University Park, Pennsylvania 16802, USA
`jcc8 at psu dot edu`

[b] *Nikhef Theory Group*
Science Park 105, 1098 XG Amsterdam, The Netherlands
`t68 at nikhef dot nl`

(2 September 2019)

Abstract

This is the reference manual for version two of the LaTeX graphical style file Axodraw. Relative to version one, it has a number of new drawing primitives and many extra options, and it can now work with `pdflatex` to directly produce output in PDF file format (but with the aid of an auxiliary program).

Contents

1 Introduction

This is the documentation for axodraw2, a LaTeX package for drawing Feynman graphs (and other simple graphics). This is version 2 and a substantial update of the original axodraw package [1], which was released in 1994, and which has become rather popular in the preparation of articles in elementary-particle physics. One of its advantages is that its drawing primitives are included in the `.tex` document, in a human-writable form. (This also allows convenient production of axodraw figures by other software, e.g., Jaxodraw [2, 3].) This is in distinction to methods that use a separate program to create graphics files that are read in during the processing of the LaTeX file. The objects needed in Feynman graphs are often difficult to draw at high quality with conventional computer graphics software.

The original axodraw package has hardly been modified since its introduction. The new version addresses several later needs. A detailed list of the changes is given in Sec. 2.

One change arises from the fact that TeX (and hence LaTeX) itself does not possess sufficiently useful methods of drawing complicated graphics, so that the drawing of the graphics is actually done by inserting suitable code in the final output file (postscript or pdf). The original axodraw worked only with the `latex-dvips` processing chain to put the diagrams in the final postscript file.[1] Now we also have in common use the `pdflatex` (and `lualatex` and `xelatex`) programs that directly produce pdf. The new version of axodraw works with `pdflatex`, `lualatex`, and `xelatex`, as well as with the `latex-dvips` method.

Furthermore, more kinds of graphical object and greater flexibility in their properties have been found useful for Feynman graphs. The new version provides a new kind of line, Bézier, and is able to make the various kinds of line doubled. There is now a very flexible configuration of arrows. Many of the changes correspond to capabilities of JaxoDraw [2, 3], which is a graphical program for drawing Feynman graphs, and which is able to write and to import diagrams in the axodraw format.

Finally, substantial improvements have been made in the handling of colors, with much better compatibility with modern packages used to set colors in the normal LaTeX part of a document.

Since some of the changes (especially in the internal coding) introduce incompatibilities (Sec. 2.3) with the original version of axodraw, the new version of the style file is given a new name `axodraw2.sty`. Then the many legacy documents (e.g., on `http://arxiv.org`) that use the old axodraw will continue to use the old version, and will therefore continue to be compilable without any need for any possible changes in the source document, and with unchanged output. Even so, as regards the coding of diagrams, there are very few backwardly incompatible changes in axodraw2.

The software is available under the GNU General Public License [4] version 3.

2 Changes

---

[1] A pdf file can be produced from the postscript file by a program like `ps2pdf`.

2.1 Changes relative to original, axodraw version 1

Relative to the original version of axodraw, the current version, axodraw2, has the following main changes:

- A bug that the line bounding an oval did not have a uniform width has been corrected.
- A bug has been corrected that axodraw did not work with the revtex4 document class when `\maketitle` and two-column mode were used.
- Axodraw2 works both when pdf output is produced directly using the programs `pdflatex`, `lualatex`, and `xelatex`, as well as when a postscript file is produced by the `latex–dvips` method. The old version only worked when postscript output was produced. However, an auxiliary program is needed when using `pdflatex`, `lualatex`, or `xelatex`. See Sec. 4.3 for how this is done.
- In the original axodraw, a diagram is coded inside a `picture` environment of LaTeX. Now, a specialized `axopicture` environment is provided and preferred; it provides better behavior, especially when diagrams are to be scaled.
- In association with this, there are some changes in how scaling of diagrams is done.
- An inconsistency in length units between postscript and TeX has been corrected. All lengths are now specified in terms of $1\,\mathrm{pt} = 1/72.27\,\mathrm{in} = 0.3515\,\mathrm{mm}$. Previously the unit length for graphics was the one defined by postscript to be $1\,\mathrm{bp} = 1/72\,\mathrm{in} = 0.3528\,\mathrm{mm}$.
- Substantial improvements have been made in the treatment of color. When named colors are used, axodraw2's use of color is generally compatible with that of the modern, LaTeX-standard `color.sty` package. It also provides all the color-setting macros that were defined in v. 1 of axodraw, including those of the `colordvi.sty` package used by v. 1.
- The various types of line can now be produced as double lines, e.g., . This is commonly used, for example, for notating Wilson lines.
- Lines can be made from Bézier curves. Currently this is only for simple lines, not photon, gluon, or zigzag lines.
- Gluon, photon, and zigzag lines can be dashed.
- Macros are provided for drawing gluon circles, without the endpoint effects given by the corresponding gluon arc macros.
- The positions and sizes of arrows can be adjusted. See Sec. 5.7 for all the possibilities. One example is
- Macros for drawing polygons and filled polygons are provided.

- Macros for drawing rotated boxes are provided.
- A macro `\ECirc` is provided for drawing a circle with a transparent interior.
- A macro `\EBoxc` is provided for drawing a box with a specified center.
- A macro `\AxoGrid` is provided for drawing a grid. One use is to provide a useful tool in designing pictures.
- Since there are now many more possibilities to specify the properties of a line, optional arguments to the main line drawing commands can be used to specify properties in a keyword style.
- A new macro named `\Arc` is introduced. With the aid of optional arguments, this unifies the behavior of various arc-drawing commands in the original axodraw.
- For consistency with the `\Gluon` macro, the `\GlueArc` macro has been renamed to `\GluonArc`, with the old macro retained as a synonym.
- The behavior of arcs is changed to what we think is more natural behavior when the specified opening angle is outside the natural range.
- What we call macros for drawing objects with postscript text are now implemented within LaTeX instead of relying on instructions inserted in the postscript code. Thus all the normal LaTeX commands, including mathematics, can now be used in all text objects, with proper scaling. The placement and scaling of text objects are more consistent.
- Some new named colors are provided: LightYellow, LightRed, LightBlue, LightGray, VeryLightBlue. (LightYellow, LightRed, LightBlue, LightGray, VeryLightBlue.)
- The macros originally specified as `\B2Text`, `\G2Text`, and `\C2Text` are now named `\BTwoText`, `\GTwoText`, and `\CTwoText`. The intent of the original code was to define macros with names `\B2Text`, etc. However in TeX, under normal circumstances, macro names of more than one character must only contain letters, unlike typical programming languages that also allow digits. So the rules for TeX macro names mean that in defining, for example `\def\B2Text(#1,#2)#3#4{...}`, the original version of axodraw actually defined a macro named named `\B`, obligatorially followed by `2Text`. This caused a conflict if the user wished to define a macro `\B`. If it is desired to retain the old behavior, then the following should be placed in the preamble of the `.tex` file, then the axodraw2 package should be invoked in the source document with the `v1compatible` option:

```
\usepackage[v1compatible]{axodraw2}
```

2.2 Changes relative to axodraw4j distributed with JaxoDraw

The JaxoDraw program [3] is distributed with a version of axodraw called axodraw4j. At the time of writing (July 2016), this was effectively a predecessor of axodraw2, but without the possibility of working with `pdflatex`. (The suffix “4j” is intended to mean “for JaxoDraw”.)

The changes in axodraw2 relative to the version of axodraw4j dated 2011/03/22 are the following subset of those listed in Sec. 2.1:

- The ability to work with `pdflatex`, `lualatex`, and `xelatex`.
- The improvements in the handling of color and fonts.
- The double and arrow options for Bézier lines.
- The dash option for gluons and photons.
- Color option for all lines.
- Correction of inconsistency of length unit between TEX and postscript.
- Better drawing of double gluons and photons.
- Addition of the macros for making gluon circles, polygons, and rotated boxes.
- Addition of the macros `\ECirc`, `\EBoxc`, and `\AxoGrid`.
- A series of “LongArrow” macros for drawing lines with the arrow at the end. The same effect could only be achieved in axodraw4j with arrowpos=1 option to the basic line-drawing commands.
- A series of macros like `\DashDoubleLine` to provide access to the dashed and double properties in the style of the macros provided in v. 1 of axodraw. This is in addition to the optional arguments that allow the same effect in axodraw4j and in axodraw2.
- The `v1compatible` and other options are provided for the package.
- Better treatment of the scaling of objects.
- The treatment of “postscript text objects” within LATEX itself.

2.3 Backward compatibility, etc

The official user interface of axodraw2 is backward-compatible with versions 1 and 4j, with the exception of the issue mentioned above about the commands that had the signatures `\B2Text`, `\G2Text`, and `\C2Text` in the old version.

As to behavior, there are some minor changes in the objects that are drawn, mostly concerning the exact dimensions of default arrows. The scaling of the sizes of text objects is changed. The scoping of color changes is substantially changed, but improved.

The old axodraw only used the tools available in LaTeX in the early 1990s. The new version needs a more modern installation. Installations for which axodraw2 has been tested include TeXLive 2011 and 2016.

We have tested backwards compatibility by compiling the version 1 manual with axodraw2; only a trivially modified preamble was needed. It also worked to compile Collins's QCD book[5], which has a large number of JaxoDraw figures (processed automatically to pieces of axodraw code imported into the document); only changes in the preamble were needed.

In addition, axodraw2 provides an `axopicture` environment inside of which axodraw2's graphics are coded and drawn. In the old axodraw, LaTeX's `picture` environment was used instead. In axodraw2, we recommend only the use of `axopicture` environment, and that is the only method we document. However, old diagrams coded with `picture` environment continue to work.

Some possible compatibility issues could arise because the old and new versions load a different set of external packages. The old version loaded the packages `epsf.sty` and `colordvi.sty`, and so macros defined by these packages were available. The new version does not load these now essentially-obsolete packages; so they should be loaded from the document if they happen to be needed. But the new version does define some macros for setting colors that correspond to those in `colordvi.sty`; these are defined using facilities from the standard LaTeX `color.sty` package. Axodraw2 loads the following LaTeX packages: `keyval`, `ifthen`, `graphicx`, `color`, `ifxetex`. It defines its own set of 73 named colors which are the 68 dvips-defined names (as coded in `colordvi.sty` and used in axodraw v. 1), plus 5 more.

3 Installation

3.1 Installation from standard TEX distribution

At the moment that this document was updated (January 2018), axodraw2 was part of both the main TEX distributions, TeXLive and MiKTeX. The easiest way to install axodraw2 is therefore from the package manager of your TEX distribution.

You can also obtain axodraw2 from CTAN at `http://ctan.org/pkg/axodraw2`, and install it manually, following the instructions in Sec. 3.2 below.

`axohelp` in TeXLive In TeXLive 2018 and later, a binary executable for the `axohelp` is provided, as part of the `axodraw2` package. So `axohelp` is available provided that the `axodraw2` package is installed..

`axohelp` in MiKTeX The axodraw2 package including an executable `axohelp.exe` was provided by MiKTeX when this was checked in January 2018.

3.2 Manual installation

For a manual installation, what needs to be done is to put the file `axodraw2.sty` in a place where it will be found by the `latex` program. If you wish to use axodraw2 with `pdflatex`, you will also need to compile the `axohelp` program — see Sec. 3.2.2 — and put it in an appropriate directory. Documentation can also be installed if you want.

3.2.1 Style file `axodraw2.sty`

If you merely want to try out axodraw2, just put the file `axodraw2.sty` in the same directory as the `.tex` file(s) you are working on.

Otherwise, put it in an appropriate directory for a LaTeX style file, and, if necessary, run the texhash program to ensure that the file is in the TeX system's database of files. For example, suppose that you have a TeXLive system installed for all users on a Unix-like system (e.g., Linux or OS-X), and that TeXLive is installed, as is usual, under the directory `/usr/local/texlive`. Then an appropriate place for axodraw2 is in a directory `/usr/local/texlive/texmf-local/tex/latex/axodraw2`. You will need to run the `texhash` program in this last case. For such a system-wide installation, you may have to do these operations as an administrative user (e.g., root), possibly supplemented by running the relevant commands with the `sudo` program.

If you later install the axodraw2 package from the package manager of your TeX distribution, it's a good idea to delete the files you installed manually. Otherwise when you use axodraw2 in a document, then the wrong version of `axodraw2.sty` may get used. This is a particularly important issue after possible future updates to axodraw2 get installed by the package manager.

3.2.2 Helper program `axohelp`

If you wish to use axodraw2 with `pdflatex`, `lualatex`, or `xelatex`, then you need to install the `axohelp` program. *(It is useful to reiterate here that the standard distributions of TeX currently supply the* `axohelp` *program. So the steps described here are only necessary if for some reason you wish to do a manual installation. One possible reason is to use a recent update of* `axohelp`*, since TeXLive normally only supplies updated versions of binary executable files with the initial release of one of TeXLive's yearly versions.)*

To install `axohelp` manually, you will first need to compile the program by a C compiler. Under a Unix-like operating system (linux or macOS) an appropriate shell command is

```
cc -o axohelp -O3 axohelp.c -lm
```

(Note that this is a C compiler, *not* a C++ compiler.) Most linux systems have the program `cc` already installed. This also applies to macOS(OS-X) at versions below 10.7.

But on macOS version 10.7 and higher, you will need to install a compiler, which can be done by installing XCode and the associated command-line utilities. If you have the GNU compilers installed, you might need to use the command `gcc` instead of `cc`.

For Microsoft Windows, you will need to have installed a C compiler, and use it to compile `axohelp.c`.

Once you have the executable (named `axohelp` on Unix-like systems, or `axohelp.exe` on a Microsoft system), put it in a directory where it will be found when you run programs from the command line.

3.2.3 Testing

To test whether the installation works, you need a simple test file. An example is given in Sec. 4.1, and is provided with the axodraw2 distribution as `example.tex`.

At a command line with the current directory set to the directory containing the file `example.tex`, run the following commands:

```
latex example
dvips example -o
```

If all goes well, you will obtain a file `example.ps`. When you view it, it should contain the diagram shown in Sec. 4.1. You can make a pdf file instead by the commands

```
latex example
dvipdf example
```

A more extensive test can be made by compiling the manual.

To make a pdf file directly, with `pdflatex`, you use the commands

```
pdflatex example
axohelp example
pdflatex example
```

The `axohelp` run takes as input a file `example.ax1` produced by the first run of `pdflatex` and makes an output file `example.ax2`. The second run of `pdflatex` reads the `example.ax2` file and uses the result to place the axodraw objects in the `example.pdf` file.

3.2.4 Documentation

Put the documentation in a place where you can find it. If you installed the `axodraw2.sty` file in `/usr/local/texlive/texmf-local/tex/latex/axodraw2`, the standard place for the documentation would be `usr/local/texlive/texmf-local/doc/latex/axodraw2`.

4 Use

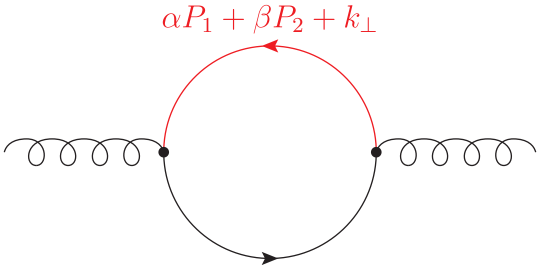
In this section we show how to use axodraw2, illustrated with an example.

4.1 Basic example

The principles of using axodraw2 are illustrated by the following complete LaTeX document:

```
\documentclass{article}
\usepackage{axodraw2}
\begin{document}
Example of Feynman graph using axodraw2 macros:
\begin{center}
  \begin{axopicture}(200,110)
    \SetColor{Red}
    \Arc[arrow](100,50)(40,0,180)
    \Text(100,100){$\alpha P_1 + \beta P_2 + k_\perp$}
    \SetColor{Black}
    \Arc[arrow](100,50)(40,180,360)
    \Gluon(0,50)(60,50){5}{4}
    \Vertex(60,50){2}
    \Gluon(140,50)(200,50){5}{4}
    \Vertex(140,50){2}
  \end{axopicture}
\end{center}
\end{document}
```

After compilation according to the instructions in Sec. 4.3, viewing the resulting file should show the following Feynman graph:



See Sec. 6 for more examples

Important note about visibility of graphics objects: If you view this document on a computer monitor, Feynman graphs drawn with narrow lines may not fully match what was intended. This is because of the way graphics viewers interact with the limited resolution of computer monitors. To see the example graphs properly, you may need to use a large enough magnification, or to use an actual print out.

Note about sending a document to others: If for example, you submit an article to arXiv.org, it is likely that their automated system for processing the file will not run `axohelp`. So together with the tex file, you one should also submit the `.ax2` file.

4.2 Document preparation

The general rules for preparation of a document are:

- Insert the following

  ```
  \usepackage{axodraw2}
  ```

  in the preamble of the `.tex` file. There are some options and commands that can be used to change axodraw2's behavior from its default. See Secs. 5.1 and 5.9 for details.

- Where you want to insert axodraw2 objects, put them inside an axopicture environment, specified in Sec. 5.2,

  ```
  \begin{axopicture}(x,y)
     ...
  \end{axopicture}
  ```

  Here `x` and `y` denote the desired size of the box that is to be inserted in the document and that contains the graph. An optional offset can be specified (as with LaTeX's `picture` environment). By default the units are 1 pt = 1/72.27 in = 0.3515 mm.

Full details of all these components are in Sec. 5.

The design of graphs can be done manually, and this can be greatly facilitated with the new `\AxoGrid` command. A convenient way of constructing diagrams is to use the graphical program JaxoDraw [2, 3], which is what most people do. This program can export axodraw code. It also uses axodraw as one way of making postscript and pdf files. The original version of axodraw was used by JaxoDraw until version 1.3. In version 2 of JaxoDraw, a specially adapted version of `axodraw.sty` is used, named `axodraw4j.sty`. The output from version 2 of JaxoDraw is compatible with axodraw2.

4.3 Document compilation

4.3.1 To make a postscript file

When a postscript file is needed, you just make the postscript file as usual. E.g., when the source file is `example.tex`, you run the following commands:

```
latex example
dvips example -o
```

which results in a postscript file `example.ps`. Of course, if there are cross references to be resolved, you may need multiple runs of `latex`, as usual. When needed, use of `bibtex`, `makeindex`, and other similar programs is also as usual. Instead of `latex`, one may also use the `dvilualatex` program, which behaves like `latex` except for providing some extra capabilities that are sometimes useful.

Internally, axodraw uses TEX's `\special` mechanism to put specifications of postscript code into the `.dvi` file, and `dvips` puts this code in the postscript file. This postscript code performs the geometrical calculations needed to specific axodraw's objects, and then draws them when the file is displayed or printed.

Important note about configuration of `dvips`: You may possibly find that when you run `dvips` that it spends a lot of time running `mktexpk` to make bitmapped fonts, or that the postscript file contains bitmapped type-3 fonts. This is *not* the default situation in typical current installations. But if you do find this situation, which is highly undesirable in most circumstances, you should arrange for `dvips` to use type 1 fonts. This can be done either by appropriately configuring your TEX installation, for which you will have to locate instructions, or by giving `dvips` its `-V0` option:

```
dvips -V0 example -o
```

Once you do this, you should see, from `dvips`'s output, symptoms of its use of type 1 fonts. *Let us re-emphasize that you do not have to be concerned with this issue, under normal circumstances. But since things were different within our memory, we give some suggestions as to what to do in what are currently abnormal circumstances.*

4.3.2 To make a pdf file via `latex`

There are multiple methods of making pdf files for a latex document; we will not give all the advantages and disadvantages here.

One way is to convert the postscript file, e.g., by

```
ps2pdf example.ps
```

You can also produce a pdf file from the dvi file produced by `latex` by the `dvipdf` command, e.g,.

```
dvipdf example
```

Important note: The program here is `dvipdf` and *not* the similarly named `dvipdfm` or `dvipdfmx`, which are incompatible with axodraw. The reason why `dvipdf` works is that it internally makes a postscript file and then converts it to pdf.

4.3.3 To make a pdf file by `pdflatex`, `lualatex`, or `xelatex`

A common and standard way to make a pdf file is the `pdflatex` program, which makes pdf directly. It has certain advantages, among which are the possibility of importing a wide variety of graphics file formats. (In contrast, the `latex` program only handles encapsulated postscript.)

However, to use axodraw2 with `pdflatex`, you need an auxiliary program, `axohelp`, as in

```
pdflatex example
axohelp example
pdflatex example
```

What happens is that during a run of `pdflatex`, axodraw2 writes a file `example.ax1` containing specifications of its graphical objects. Then running `axohelp` reads the `example.ax1` file, computes the necessary pdf code to draw the objects, and writes the results to `example.ax2`. The next run of `pdflatex` reads `example.ax2` and uses it to put the appropriate code in the output pdf file.

The reason for the extra program is that axodraw needs many geometrical calculations to place and draw its graphical objects. LaTeX itself does not provide anything convenient and efficient for these calculations, while the PDF language does not offer sufficient computational facilities, unlike the postscript language.

If you modify a document, and recompile with `pdflatex`, you will only need to rerun `axohelp` if the modifications involve axodraw objects. Axodraw2 will output an appropriate message when a rerun of `axohelp` is needed.

If you wish to use `lualatex` or `xelatex`, instead of `pdflatex`, then you can simply run the program `lualatex` or `xelatex` instead of `pdflatex`. These are equally compatible with axodraw2.

4.4 Automation of document compilation

It can be useful to automate the multiple steps for compiling a LaTeX document. One of us has provided a program `latexmk` to do this — see `http://www.ctan.org/pkg/latexmk/`. Here we show how to configure `latexmk` to run `axohelp` as needed when a document is compiled via the `pdflatex` route.

All you need to do is to put the following lines in one of `latexmk`'s initialization files (as specified in its documentation):

```
add_cus_dep( "ax1", "ax2", 0, "axohelp" );
sub axohelp { return system "axohelp \"$_[0]\""; }
$clean_ext .= " %R.ax1 %R.ax2";
```

The first two lines specify that `latexmk` is to make `.ax2` files from `.ax1` files by the `axohelp` program, whenever necessary. (After that `latexmk` automatically also does any further runs of `pdflatex` that are necessary.) The last line is optional; it adds `.ax1` and `.ax2` files to the list of files that will be deleted when `latexmk` is requested to do a clean up of generated, recreatable files.

`Latexmk` is installed by default by the currently common distributions of TEX software, i.e., TeXLive and MiKTeX. It has as an additional requirement a properly installed Perl system. For the TeXLive distribution, this requirement is always met.

With the above configuration, you need no change in how you invoke `latexmk` to compile a document, when it uses axodraw2. For producing postscript, you can simply use

```
latexmk -ps example
```

and for producing pdf via `pdflatex` you can use

```
latexmk -pdf example
```

Then `latexmk` takes care of whatever runs are needed of all the relevant programs, now including `axohelp`, as well whatever, possibly multiple, runs are needed for the usual programs (`latex`, `pdflatex`, `bibtex`, etc).

5 Reference

5.1 Package invocation

To use the axodraw2 package in a LATEX document, you simply put

```
\usepackage{axodraw2}
```

in the preamble of the document, as normal.

The `\usepackage` command takes optional arguments (comma-separated list of keywords) in square brackets, e.g.,

```
\usepackage[v1compatible]{axodraw2}
```

The options supported by axodraw2 are

- `v1compatible`: This makes axodraw2's operation more compatible with v. 1. It allows the use of `\B2Text`, `\G2Text`, and `\C2Text` as synonyms for the macros named `\BTwoText`, `\GTwoText`, and `\CTwoText`. (You may wish also to use the `canvasScaleisUnitLength` option, so that the scaling of the units in the `axopicture` environment is the same as it was for the `picture` environment used in v. 1.)

- `canvasScaleIs1pt`: Unit for canvas dimensions in an `axopicture` environment is fixed at 1 pt,

- `canvasScaleIsObjectScale`: Unit for canvas dimensions in an `axopicture` environment are the same as those set for axodraw objects (by the `\SetScale` macro). This is the default setting, so the option need not be given.

- `canvasScaleIsUnitLength`: Unit for canvas dimensions in an `axopicture` environment is the current value of `\unitlength`, exactly as for LaTeX's `picture` environment. (Thus, this corresponds to the behavior of the original axodraw v. 1, which simply used the `picture` environment.)
- `PStextScalesIndependently`: Axodraw's text objects are scaled by the factor set by the `\SetTextScale` command.
- `PStextScalesLikeGraphics`: Axodraw's text objects are scaled by the factor set by same factor for its graphics objects, i.e., the scale set by the `\SetScale` command.

(N.B. Default scaling factors are initialized to unity.)

Note: If you use `axodraw`'s commands for placing text and you use the standard TeX Computer Modern fonts for the document, then when you compile your document you may get a lot of warning messages. These are about fonts not being available in certain sizes. To fix this problem invoke the package `fix-cm` in your document's preamble:

```
\usepackage{fix-cm}
```

It is also possible to use the package `lmodern` for the same purpose.

5.2 Environment(s)

The graphical and other objects made by axodraw2 are placed in an `axopicture` environment, which is invoked either as

```
\begin{axopicture}(x,y)
   ...
\end{axopicture}
```

or

```
\begin{axopicture}(x,y)(xoffset,yoffset)
   ...
\end{axopicture}
```

Here, the ... denote sequences of axodraw2 commands, as documented in later sections, for drawing lines, etc. The `axopicture` environment is just like standard LaTeX's `picture` environment,[2], except for doing some axodraw-specific initialization. It inserts a region of size `x` by `y` (with default units of 1 pt = 1/72.27 in = 0.3515 mm). Here `x` and `y` are set to the numerical values you need.

---

[2] In fact, the `axopicture` is changed from the `picture` environment only by making some axodraw-specific settings. So the `picture` environment that was used in v. 1 may also be used with axodraw2; it merely has a lack of automation on the setting of the canvas scale relative to the object scale, and, in the future, other possible initializations.

The positioning of axodraw objects is specified by giving x and y coordinates, e.g., for the ends of lines. The origin of these coordinates is, by default, at the lower left corner of the box that `axopicture` inserts in your document. But sometimes, particularly after editing a graph, you will find this is not suitable. To avoid changing a lot of coordinate values to get correct placement, you can specify an offset by the optional arguments `(xoffset,yoffset)` to the `axopicture` environment, exactly as for LaTeX's `picture` environment. The offset `(xoffset,yoffset)` denotes the position of the bottom left corner of the box inserted in your document relative to the coordinate system used for specifying object positions. Thus

```
\begin{axopicture}(20,20)
   \Line(0,0)(20,20)
\end{axopicture}
```

and

```
\begin{axopicture}(20,20)(-10,20)
   \Line(-10,20)(10,40)
\end{axopicture}
```

are exactly equivalent.

Within an `axopicture` environment, all the commands that can be used inside an ordinary `picture` environment can also be used.

We can think of the `axopicture` environment as defining a drawing canvas for axodraw's graphical and text objects. There are possibilities for manipulating (separately) the units used to specify the canvas and the objects. These can be useful for scaling a diagram or parts of it from an originally chosen design. See Secs. 5.8 and 5.9 for details.

5.3 Graphics drawing commands

In this section we present commands for drawing graphical objects, split up by category. Later, we will give: details of options to the line-drawing commands, explanations of some details about specifying gluons and about specifying arrow parameters, and then commands for textual objects and for adjusting settings (e.g., separation in a double line). Mostly, we present the commands by means of examples. Note that many of the arguments of the commands, notably arguments for (x, y) coordinate values are delimited by parentheses and commas instead of the brace delimiters typically used in LaTeX.

It should also be noted that some commands provide different ways of performing the same task. For instance

```
\BCirc(50,50){30}
```

can also be represented by

```
\CCirc(50,50){30}{Black}{White}
```

when the current color is black. The presence of the BCirc command has been maintained both for backward compatibility, and because it represents a convenient short hand for a common situation. This also holds for similar commands involving boxes and triangles. For the new Polygon, FilledPolygon, RotatedBox and FilledRotatedBox commands we have selected a more minimal scheme.

Similar remarks apply to the new feature of options for line drawing commands. Originally in v. 1, a line with an arrow would be coded as

```
\ArrowLine(30,65)(60,25)
```

It is now also possible to code using the general `\Line` macro, but with a keyword optional argument:

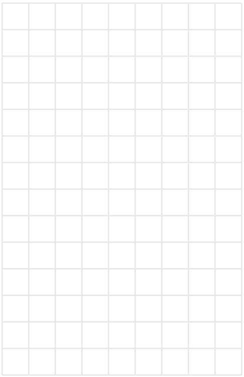
```
\Line[arrow](30,65)(60,25)
```

One advantage of the option method is a variety of other properties of an individual line may also be coded, as in

```
\Line[arrow,arrowpos=1](30,65)(60,25)
```

without the need to use separate global setting for the property, by the commands listed in Sec. 5.9, or by having a corresponding compulsory argument to the command. Which way to do things is a matter of user taste in particular situations.

5.3.1 Grid drawing



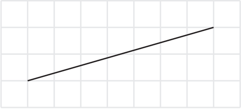
```
\AxoGrid(0,0)(10,10)(9,14){LightGray}{0.5}
```

This command is used in our examples to allow the reader to compare the coordinates in the commands with those of the actual picture. The arguments are first the position of the left bottom corner, then two values that tell the size of the divisions in the x and y direction. Next there are two values that specify how many divisions there should be in the x and y direction. Then the color of the lines is given and finally the width of the lines. Note that if there are (n_x, n_y) divisions there will be n_x+1 vertical lines and n_y+1 horizontal lines. The temporary use of this command can also be convenient when designing pictures manually.

5.3.2 Ordinary straight lines

All of the commands in this section can be given optional keyword arguments, which are defined in Secs. 5.5 and 5.7. These can be used to specify the type of line (dashed, double), to specify the use of an arrow, and its parameters, and to specify some of the line's parameters.

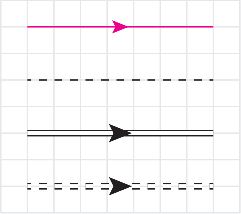
The basic line drawing command is `\Line`:



```
\Line(10,10)(80,30)
```

In this command we have two coordinates. The (solid) line goes from the first to the second.

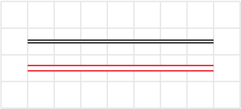
Examples of the use of optional arguments are:



```
\Line[color=Magenta,arrow](10,70)(80,70)
\Line[dash](10,50)(80,50)
\Line[arrow,double](10,30)(80,30)
\Line[arrow,dash,double](10,10)(80,10)
```

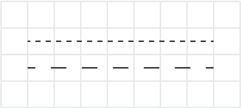
Details of the specification of arrows, together with alternative commands for making lines with arrows are given in Sec. 5.7.

Alternative commands for dashed and/or double lines are:



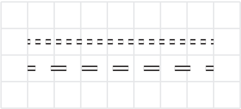
```
\DoubleLine(10,25)(80,25){1}
\DoubleLine[color=Red](10,15)(80,15){2}
```

In this command we have two coordinates as in the Line command but two lines are drawn. The extra parameter is the separation between the two lines. Note however that everything between the lines is blanked out.



```
\DashLine(10,25)(80,25){2}
\DashLine(10,15)(80,15){6}
```

In this command we have two coordinates. The dashed line goes from the first to the second. The extra parameter is the size of the dashes. The space between the dashes is transparent.



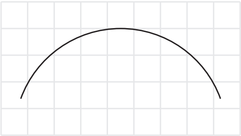
```
\DashDoubleLine(10,25)(80,25){1.5}{2}
\DashDoubleLine(10,15)(80,15){1.5}{6}
```

In this command we have two coordinates. The dashed lines go from the first to the second. The first extra parameter is the separation between the lines and the second extra parameter is the size of the dashes.

5.3.3 Arcs

The commands in this section draw circular arcs in types corresponding to the straight lines of Sec. 5.3.2. In v. 1, some of these commands had names containing “Arc” and some “CArc”. Some kinds had variant names containing “Arcn”, whose the direction of drawing was clockwise instead of anticlockwise. In v. 2, we have tried to make the situation more consistent. First, all the old names have been retained, for backward compatibility. Second, a general purpose command `\Arc` has been introduced; in a single command, with the aid of optional arguments, it covers all the variants. See Secs. 5.5 and 5.7 for full details. The options can be used to specify the type of line (dashed, double, clockwise or anticlockwise), to specify the use of arrow, and its parameters, and to specify some of the line's parameters. The other commands in this section can also be given optional keyword arguments.

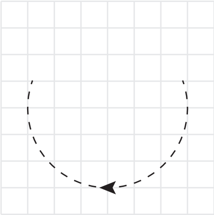
The basic `\Arc` command has the form



```
\Arc(45,0)(40,20,160)
```

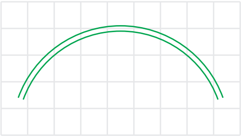
In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise.

An example of the use of the optional parameters is:



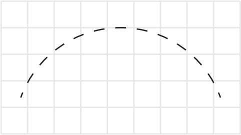
```
\Arc[arrow,dash,clockwise](40,40)(30,20,160)
```

Alternative commands for dashed and/or double arcs are as follows.



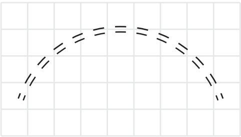
```
\DoubleArc[color=Green](45,0)(40,20,160){2}
```

In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The last argument is the line separation of the double line.



```
\DashArc(45,0)(40,20,160){4}
```

In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The last argument is the size of the dashes.



```
\DashDoubleArc(45,0)(40,20,160){2}{4}
```

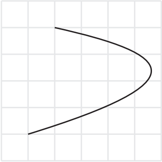
In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The last two arguments are the line separation of the double line and the size of the dashes.

5.3.4 Bézier lines

The commands in this section draw Bézier curves, specified by 4 points. The variants are just as for straight lines, Sec. 5.3.2.

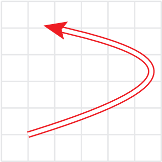
All of the commands in this section can be given optional keyword arguments, which are defined in Sec. 5.5. These can be used to specify the type of line (dashed, double), to specify the use of an arrow, and its parameters, and to specify some of the line's parameters.

The basic general purpose command is `\Bezier`:



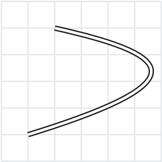
`\Bezier(10,10)(75,30)(65,40)(20,50)`
Draws a cubic Bézier curve based on the four given points. The first point is the starting point and the fourth the finishing point. The second and third points are the two control points.

An example of the use of optional arguments is

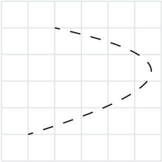


```
\Bezier[color=Red,arrow,double,arrowpos=1](10,10)%
  (75,30)(65,40)(20,50)
```

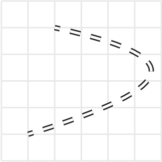
Alternative ways of making dashed and/or double Bézier curves are:



`\DoubleBezier(10,10)(75,30)(65,40)(20,50){1.5}`
Draws a cubic Bézier curve based on the four given points. The first four arguments are the same as for `\Bezier`. The final argument is the line separation.



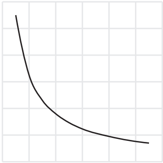
`\DashBezier(10,10)(75,30)(65,40)(20,50){4}`
Draws a cubic Bézier curve based on the four given points. The first four arguments are the same as for `\Bezier`. The final argument is the size of the dashes.



`\DashDoubleBezier(10,10)(75,30)(65,40)(20,50){1.5}{4}`
Draws a cubic Bézier curve based on the four given points. The first four arguments are the same as for `\Bezier`. The final two arguments are the line separation and the size of the dashes.

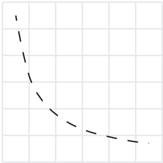
5.3.5 Curves

The commands in this section draw curves through an arbitrary sequence of points. They only exist in variants for continuous and dashed lines. No optional arguments are allowed.



```
\Curve{(5,55)(10,32.5)(15,23)(20,18)
        (25,14.65)(30,12.3)(40,9.5)(55,7)}
```

Draws a smooth curve through the given points. The x coordinates of the points should be in ascending order. The curve is obtained by constructing quadratic fits to each triplet of adjacent points and then in each interval between two points interpolating between the two relevant parabolas.



```
\DashCurve{(5,55)(10,32.5)(15,23)(20,18)
        (25,14.65)(30,12.3)(40,9.5)(55,7)}{4}
```

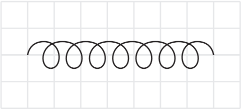
Draws a smooth dashed curve through the given points. The x coordinates of the points should be in ascending order. The last argument is the size of the dashes.

5.3.6 Gluon lines

The basic gluon drawing commands are `\Gluon`, `\GluonArc`, `\GluonCirc`. There are also variants for dashed and double gluons. But arrows aren't possible.

See Sec. 5.6 for additional information on the shape of gluon lines.

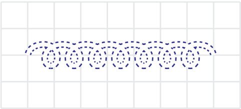
All of the commands in this section can be given optional keyword arguments, which are defined in Sec. 5.5. These can be used to specify the type of line (dashed, double), and to specify some of the line's parameters.



```
\Gluon(10,20)(80,20){5}{7}
```

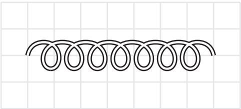
In this command we have coordinates for the start and end of the line, the amplitude of the windings and the number of windings. A negative value for the amplitude reverses the orientation of the windings — see Sec. 5.6 for details.

Optional arguments can be used, e.g.,



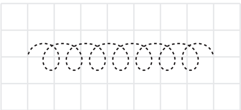
```
\Gluon[color=Blue,dash,double](10,20)(80,20){4}{7}
```

Examples of the other commands for various types of gluon line are as follows. They can all take optional arguments.



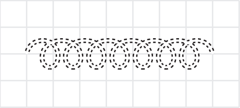
```
\DoubleGluon(10,20)(80,20){5}{7}{1.3}
```

The first 6 arguments are as in the `\Gluon` command. The extra argument is the line separation.



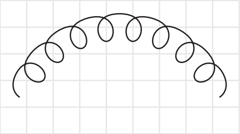
```
\DashGluon(10,20)(80,20){5}{7}{1}
```

The first 6 arguments are as in the `Gluon` command. The extra argument is the size of the dashes.



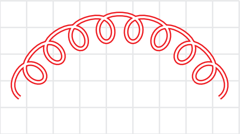
```
\DashDoubleGluon(10,20)(80,20){5}{7}{1.3}{1}
```

The first 7 arguments are as in the `DoubleGluon` command. The last two arguments are the line separation of the double line and the size of the dashes.



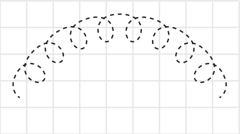
```
\GluonArc(45,0)(40,20,160){5}{8}
```

In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The final two parameters are the amplitude of the windings and the number of windings. Like the other commands in this section, this command can take optional arguments, Sec. 5.5.



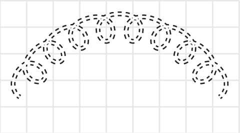
```
\DoubleGluonArc[color=Red](45,0)(40,20,160)%
                                {5}{8}{1.3}
```

The first 7 arguments are as in the `GluonArc` command. The extra argument is the separation in the double line.



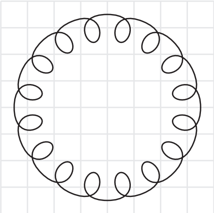
```
\DashGluonArc(45,0)(40,20,160){5}{8}{1.5}
```

The first 7 arguments are as in the `GluonArc` command. The extra argument is the size of the dash segments.



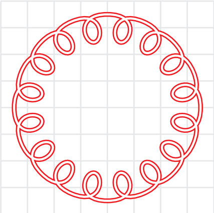
```
\DashDoubleGluonArc(45,0)(40,20,160){5}{8}{1.3}{1.5}
```

The first 7 arguments are as in the `GluonArc` command. The extra arguments are the separation of the lines and the size of the dash segments.



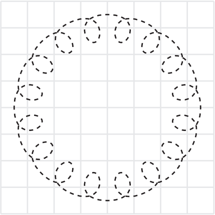
```
\GluonCirc(40,40)(30,0){5}{16}
```

The arguments are: Coordinates for the center of the circle, the radius and a phase, the amplitude of the gluon windings and the number of windings. Like the other commands in this section, this command can take optional arguments, Sec. 5.5. The phase argument specifies a counterclockwise rotation of the line relative to a default starting point.

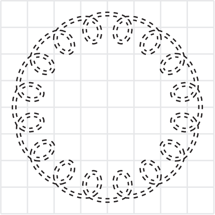


```
\DoubleGluonCirc[color=Red](40,40)(30,0){5}{16}{1.3}
```

The first 6 arguments are as for the `GluonCirc` command. The final argument is the line separation.



`\DashGluonCirc(40,40)(30,0){5}{16}{1.5}`
The first 6 arguments are as for the `GluonCirc` command. The final argument is the size of the dashes.

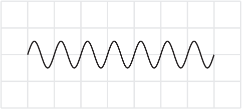


`\DashDoubleGluonCirc(40,40)(30,0){5}{16}{1.3}{1.5}`
The first 6 arguments are as for the `GluonCirc` command. The final 2 arguments are the line separation and the size of the dashes.

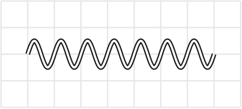
5.3.7 Photon lines

The basic drawing commands for drawing photon lines are `\Photon` and `\PhotonArc`. There are also variants for dashed and double photons. But arrows aren't possible.

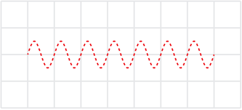
All of the commands in this section can be given optional keyword arguments, which are defined in Sec. 5.5. These can be used to specify the type of line (dashed, double), and to specify some of the line's parameters.



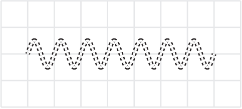
`\Photon(10,20)(80,20){5}{7}`
In this command we have two coordinates, the amplitude of the wiggles and the number of wiggles. A negative value for the amplitude will reverse the orientation of the wiggles. The line will be drawn with the number of wiggles rounded to the nearest half integer. Like the other commands in this section, this command can take optional arguments, Sec. 5.5.



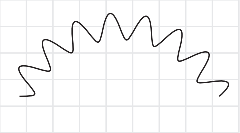
`\DoublePhoton(10,20)(80,20){5}{7}{1.3}`
The first 6 arguments are as in the `Photon` command. The extra argument is the line separation.



`\DashPhoton[color=Red](10,20)(80,20){5}{7}{1}`
The first 6 arguments are as in the `Photon` command. The extra argument is the size of the dashes.

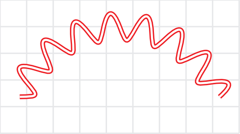


`\DashDoublePhoton(10,20)(80,20){5}{7}{1.3}{1}`
The first 6 arguments are as in the `Photon` command. The final 2 arguments are the line separation and the size of the dashes.



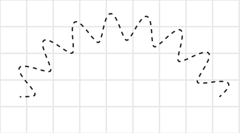
```
\PhotonArc(45,0)(40,20,160){5}{8}
```

In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The final two parameters are the amplitude of the wiggles and the number of wiggles. Like the other commands in this section, this command can take optional arguments, Sec. 5.5.



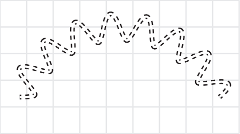
```
\DoublePhotonArc[color=Red](45,0)(40,20,160)%
                                 {5}{8}{1.3}
```

The first 7 arguments are as in the `PhotonArc` command. The extra argument is the separation of the double line.



```
\DashPhotonArc(45,0)(40,20,160){5}{8}{1.5}
```

The first 7 arguments are as in the `PhotonArc` command. The extra argument is the size of the dash segments.



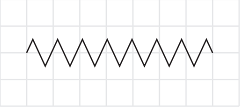
```
\DashDoublePhotonArc(45,0)(40,20,160){5}{8}{1.3}{1.5}
```

The first 7 arguments are as in the `PhotonArc` command. The extra arguments are the separation of the lines and the size of the dash segments.

5.3.8 Zigzag lines

The basic drawing commands for drawing zigzag lines are `\Zigzag` and `\ZigzagArc`. There are also variants for dashed and double lines. But arrows aren't possible.

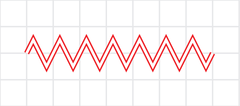
All of the commands in this section can be given optional keyword arguments, which are defined in Sec. 5.5. These can be used to specify the type of line (dashed, double), and to specify some of the line's parameters.



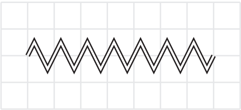
```
\ZigZag(10,20)(80,20){5}{7.5}
```

In this command we have two coordinates, the amplitude of the sawteeth and the number of sawteeth. A negative value for the amplitude will reverse the orientation of the sawteeth. The line will be drawn with the number of sawteeth rounded to the nearest half integer.

Like the other commands in this section, this command can take optional arguments, Sec. 5.5, e.g.,

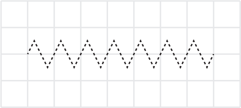


```
\ZigZag[color=Red,double,sep=1.5](10,20)(80,20){5}{7}
```



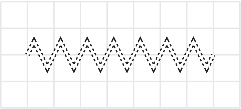
```
\DoubleZigZag(10,20)(80,20){5}{7}{1.3}
```

The first 6 arguments are as in the `ZigZag` command. The extra argument is the line separation.



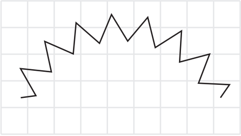
```
\DashZigZag(10,20)(80,20){5}{7}{1}
```

The first 6 arguments are as in the `ZigZag` command. The extra argument is the size of the dashes.



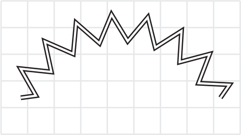
```
\DashDoubleZigZag(10,20)(80,20){5}{7}{1.3}{1}
```

The first 6 arguments are as in the `ZigZag` command. The extra arguments are the separation of the lines and the size of the dash segments.



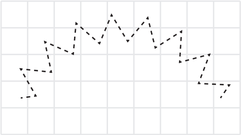
```
\ZigZagArc(45,0)(40,20,160){5}{8}
```

In this command we have one coordinate: the center of the circle. Then follow the radius of the circle, the start angle and the finishing angle. The arc will be drawn counterclockwise. The final two arguments are the amplitude of the sawteeth and the number of sawteeth. Like the other commands in this section, this command can take optional arguments, Sec. 5.5.



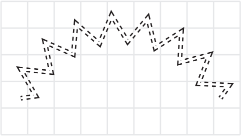
```
\DoubleZigZagArc(45,0)(40,20,160){5}{8}{1.3}
```

The first 7 arguments are as for the `ZigZagArc` command. The extra argument is the separation in the double line.



```
\DashZigZagArc(45,0)(40,20,160){5}{8}{1.5}
```

The first 7 arguments are as for the `ZigZagArc` command. The extra argument is the size of the dash segments.



```
\DashDoubleZigZagArc(45,0)(40,20,160){5}{8}{1.3}{1.5}
```

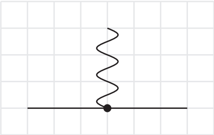
The first 7 arguments are as for the `ZigZagArc` command. The final 2 arguments are the separation of the lines and the size of the dash segments.

5.3.9 Vertices, circles, ovals; other graphics

The commands in this section are for graphical elements other than those that we conceived of as lines in Feynman graphs. Many of these have standard uses as components of Feynman graphs[3]. The commands here are mostly shown in association with other objects, to indicate some of their properties.

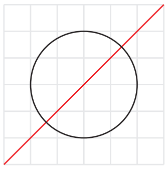
---

[3] Of course, none of the commands is restricted to its originally envisaged use, or to being used to draw Feynman graphs. But especially the line-drawing commands have been designed from the point-of-view of being suitable for the needs of drawing particular elements of Feynman graphs.



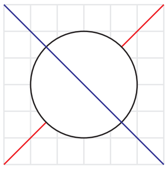
```
\Line(10,10)(70,10)
\Photon(40,10)(40,40){4}{3}
\Vertex(40,10){1.5}
```

`\Vertex` gives a vertex, as is often used for connecting lines in Feynman graphs. It gives a fat dot. The arguments are coordinates (between parentheses) for its center, and the radius of the dot.



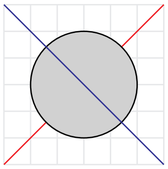
```
\Red{\Line(0,0)(60,60)}
\ECirc(30,30){20}
```

`\ECirc` draws a circle with its center at the specified coordinate (first two arguments) and the specified radius (third argument). The interior is transparent, so that it does not erase previously drawn material. If you need a filled circle, use the `\Vertex` command (to which we have defined a synonym `\FCirc` to match similar commands for other shapes).



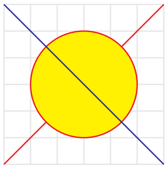
```
\Red{\Line(0,0)(60,60)}
\BCirc(30,30){20}
\Blue{\Line(60,0)(0,60)}
```

`\BCirc` draws a circle with the center at the specified coordinate (first two arguments) and the specified radius (third argument). The interior is white and opaque, so that it erases previously written objects, but not subsequently drawn objects.



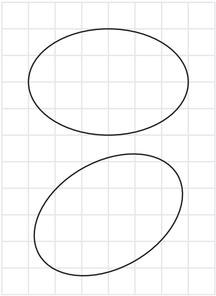
```
\Red{\Line(0,0)(60,60)}
\GCirc(30,30){20}{0.82}
\Blue{\Line(60,0)(0,60)}
```

`\GCirc` draws a circle with the center at the specified coordinate (first two arguments) and the specified radius (third argument). Previously written contents are overwritten and made gray according to the grayscale specified by the fourth argument (0=black, 1=white).



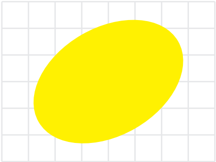
```
\Red{\Line(0,0)(60,60)}
\CCirc(30,30){20}{Red}{Yellow}
\Blue{\Line(60,0)(0,60)}
```

`\CCirc` draws a colored circle with the center at the specified coordinate (first two arguments) and the specified radius (third argument). The fourth argument is the name of the color for the circle itself. Its interior is overwritten and colored with the color specified by name in the fifth argument.



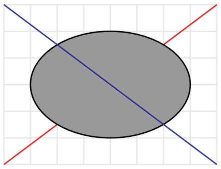
```
\Oval(40,80)(20,30)(0)
\Oval(40,30)(20,30)(30)
```

`\Oval` draws an oval. The first pair of values is the center of the oval. The next pair forms the half-height and the half-width. The last argument is a (counterclockwise) rotation angle. The interior is transparent, so that it does not erase previously drawn material.



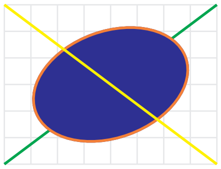
```
\SetColor{Yellow}
\FOval(40,80)(20,30)(30)
```

`\FOval` draws an oval filled with the current color overwriting previously written material. Its arguments are the same as for the `\Oval` command.



```
\Red{\Line(0,0)(80,60)}
\GOval(40,30)(20,30)(0){0.6}
\Blue{\Line(80,0)(0,60)}
```

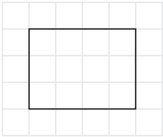
`\GOval` draws an oval with a gray interior. The first 5 arguments are the same as for the `\Oval` command. The last argument indicates the grayscale with which the oval will be filled, overwriting previously written contents (0=black, 1=white).



```
\Green{\Line(0,0)(80,60)}
\COval(40,30)(20,30)(20){Orange}{Blue}
\Yellow{\Line(80,0)(0,60)}
```

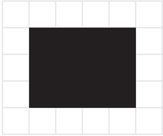
`\COval` draws a colored oval. The first 5 arguments are the same as for the `\Oval` command. The last two arguments are the names of two colors. The first is the color of the line that forms the oval and the second is the color of the inside.

Commands for drawing boxes are in two series. For the first set, the box's position is specified by the coordinates of its bottom left corner and top right corner:



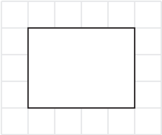
```
\EBox(10,10)(50,40)
```

Draws a box. The points specified are the bottom left corner and the top right corner. The interior is transparent, so that it does not erase previously drawn material.

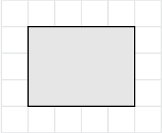


```
\FBox(10,10)(50,40)
```

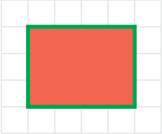
Draws a box filled with the current color overwriting previously written material. Its arguments are the same as for the `\EBox` command.



`\BBox(10,10)(50,40)`
Draws a blanked-out box. The points specified are the bottom left corner and the top right corner.

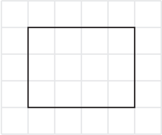


`\GBox(10,10)(50,40){0.9}`
Draws a box filled with a grayscale given by the fifth argument (black=0, white=1). The points specified are the bottom left corner and the top right corner.

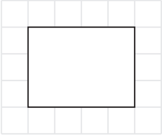


`\CBox(10,10)(50,40){Green}{LightRed}`
Draws a box in the color specified by name in the fifth argument. The contents are filled with the color specified by name in the sixth argument. The points specified are the bottom left corner and the top right corner.

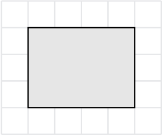
For the other series of box-drawing commands, the box's position is specified by its center, and its width and height. The command names end with a "`c`", for "center":



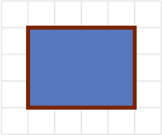
`\EBoxc(30,25)(40,30)`
Draws a box. The first two numbers give the center of the box. The next two numbers are the width and the height of the box. Instead of `\EBoxc` one may also use `\Boxc`.
There is also the similar command `\FBoxc` that draws a filled box.



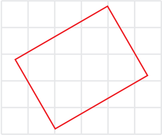
`\BBoxc(30,25)(40,30)`
Draws a box of which the contents are blanked out. The arguments are the same as for the `\EBoxc` command.



`\GBoxc(30,25)(40,30){0.9}`
Draws a box filled with a grayscale given by the fifth argument (black=0, white=1). The first 4 arguments are the same as for the `\EBoxc` command.

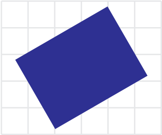


`\CBoxc(30,25)(40,30){Brown}{LightBlue}`
Draws a box in the color specified by name in the fifth argument. The contents are filled with the color specified by name in the sixth argument. The first 4 arguments are the same as for the `\EBoxc` command.



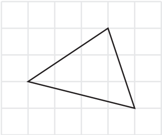
```
\RotatedBox(30,25)(40,30){30}{Red}
```

Draws a rotated box. The first two numbers give the center of the box. The next two numbers are the width and the height of the box. The fifth argument is the counterclockwise rotation angle and the sixth argument is the color of the box. The interior of the box is transparent.



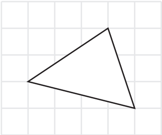
```
\FilledRotatedBox(30,25)(40,30){30}{Blue}
```

Draws a rotated box. The first 4 arguments are the same as for the `\RotatedBox` command. The fifth argument is the counterclockwise rotation angle and the sixth argument is the color of the inside of the box. If a differently colored outline is needed, it should be written with the `RotatedBox` command.



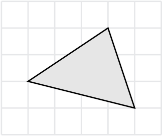
```
\ETri(10,20)(50,10)(40,40)
```

Draws a triangle. The three points specified are the corners of the triangle. The interior is transparent.
There is also the similar command `\FTri` that draws a filled triangle.



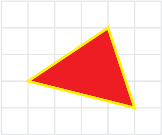
```
\BTri(10,20)(50,10)(40,40)
```

Draws a blanked-out triangle. The three points specified are the corners of the triangle.



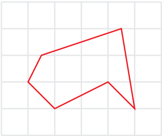
```
\GTri(10,20)(50,10)(40,40){0.9}
```

Draws a triangle of which the content are filled with the grayscale specified by the seventh argument (black=0, white=1). The three points specified are the corners of the triangle.



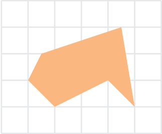
```
\CTri(10,20)(50,10)(40,40){Red}{Yellow}
```

Draws a triangle in the color named in the seventh argument. The contents are filled with the color named in the eightth argument. The three points specified are the corners of the triangle.

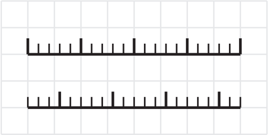


```
\Polygon{(10,20)(20,10)(40,20)(50,10)
          (45,40)(15,30)}{Red}
```

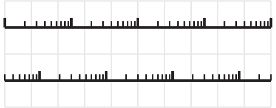
Draws a polygon. The first argument is a sequence of two dimensional points which form the corners of the polygon. The second argument is the name of the color of the polygon. The interior is transparent.



`\FilledPolygon{(10,20)(20,10)(40,20)(50,10)`
`                (45,40)(15,30)}{Apricot}`
Draws a polygon. The first argument is a sequence of two dimensional points which form the corners of the polygon. The second argument is the name of the color of the interior.



`\LinAxis(10,30)(90,30)(4,5,5,0,1)`
`\LinAxis(10,10)(100,10)(4,5,5,2,1)`
`\LinAxis`$(x_1,y_1)(x_2,y_2)(N_D,d,$hashsize,offset,width$)$
This draws a line to be used as an axis in a graph. Along the axis are hash marks. Going from the first coordinate to the second, the hash marks are on the left side if 'hashsize', which is the size of the hash marks, is positive and on the right side if it is negative. N_D is the number of 'decades', indicated by fat hash marks, and d is the (integer) number of subdivisions inside each decade. The offset parameter tells to which subdivision the first coordinate corresponds. When it is zero, this coordinate corresponds to a fat mark of a decade. Because axes have their own width, this is indicated with the last parameter.
When arguments are outside the natural range, which is a positive integer for the number of subdivisions d, and a real value between 0 and N_D for the offset, corrected values are used as follows: For d, it is first rounded to the nearest integer, and if it is zero or less, d is replaced by 1. The offset is used modulo the number of subdivisions.



`\LogAxis(0,30)(100,30)(4,3,0,1)`
`\LogAxis(0,10)(100,10)(4,3,3,1)`
`\LogAxis`$(x_1,y_1)(x_2,y_2)(N_L,$hashsize,offset,width$)$
This draws a line to be used as a logarithmic axis in a graph. Along the axis are hash marks. Going from the first coordinate to the second, the hash marks are on the left side if 'hashsize', which is the size of the hash marks, is positive and on the right side if it is negative. N_L is the number of orders of magnitude, indicated by fat hash marks. The offset parameter tells to which subdivision the first coordinate corresponds. When it is zero, this coordinate corresponds to a fat mark, which is identical to when the value would have been 1. Because axes have their own width, this is indicated with the last parameter.
When the offset is outside its natural range, which is a real value between 1 and 10, a corrected value is used as follows: If the offset is zero or less, it is replaced by 1. Then it is multiplied by an integer power of 10 to bring it into the range 1 to 10 (or equivalently, the offset's logarithm to base 10 is used modulo 1).

5.4 Text

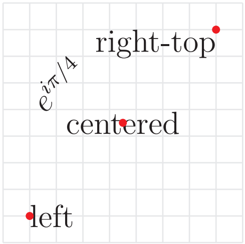
Axodraw2 provides several commands for inserting text into diagrams. Some are for plain text, with a chosen placement and angle. Some allow placement of text inside boxes. There are two sets of commands. Some we call TeX-text commands; these use the standard LaTeX fonts as used in the rest of the document. The others we call postscript-text commands; these use a user-specified standard postscript font or, if the user wishes, the usual document font, at a user-chosen size.

[*Side issue:* In version 1 of axodraw, the difference between the classes of text command was caused by a serious implementation difficulty. With the then-available LaTeX technology, certain graphic effects, could not be achieved within LaTeX, at least not easily. So direct programming in postscript was resorted to, with the result that normal LaTeX commands, including mathematics, were not available in the postscript-text commands. With the greatly improved methods now available, this has all changed, and the restrictions have gone. But since the commands and their basic behavior is already defined, we have retained the distinction between TeX-text commands and postscript-text commands.]

In the original version of Axodraw the commands for two lines inside a box were `B2Text`, `G2Text` and `C2Text`. This causes some problems explained in Sec. 2.1. If you need to retain compatibility with v. 1 on this issue, e.g., with old files or old diagrams or for personal preference, you can use the `v1compatible` option when loading axodraw2 — see Sec. 5.1.

5.4.1 TeX-type text

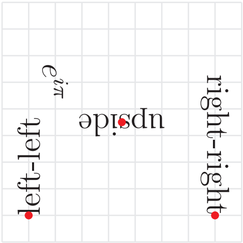
Illustrated by examples, the commands to insert text are as follows:



```
\Text(10,10)[l]{left}
\Text(45,45){centered}
\Text(80,80)[rt]{right-top}
\Text(20,60)(45){$e^{i\pi/4}$}
\SetColor{Red}
\Vertex(10,10){1.5}
\Vertex(45,45){1.5}
\Vertex(80,80){1.5}
```

`\Text` writes text in the current LaTeX font. The most general form is `\Text(x,y)(theta)[pos]{text}`; but either or both of the theta and pos arguments (and their delimiters) can be omitted. It puts the text at focal point (x, y), with a rotation by anticlockwise angle theta. The default angle is zero, and the default position is to center the text horizontally and vertically at the focal point. The position letters are any relevant combination of 'l', 'r', 't', and 'b', as in the various TeX/LaTeX box commands to indicate left, right, top or bottom adjustment with respect to the focal point. No indication means centered.

```
\rText(10,10)[l][l]{left-left}
\rText(45,45)[][u]{upside}
\rText(80,10)[r][r]{right-right}
\rText(20,60)[][r]{$e^{i\pi}$}
\SetColor{Red}
\Vertex(10,10){1.5}
\Vertex(45,45){1.5}
\Vertex(80,10){1.5}
```



The `\rText` command gives a subset of the functionality of the `\Text` command. It is used for backward compatibility with Axodraw v. 1. The general form of the command is `\rText(x,y)[mode][rotation]{text}`. Unlike the case with the `\Text` command and typical standard LaTeX commands, if the option letters are omitted, the square brackets must be retained. The coordinates (x, y) are the focal point of the text. The third argument is `l`, `r`, or empty to indicate the justification of the text. The fourth argument is `l`, `r`, `u`, or empty to indicate respectively whether the text is rotated left (anticlockwise) by 90 degrees, is rotated right (clockwise) by 90 degrees, is upside-down, or is not rotated. The fifth argument is the text. This command is retained only for backward compatibility; for new diagrams it is probably better to use the the `\Text`.

5.4.2 Postscript-type text

The remaining text-drawing commands can use postscript fonts with an adjustable size.

To set the font for later text-drawing commands in this class, the `\SetPFont` command sets the ‘Postscript’ font, e.g.,

```
\SetPFont{Helvetica}{20}
```

(This font is initialized by axodraw2 to Times-Roman at 10pt.) The font set in this way is used in the `PText`, `BText`, `GText`, `CText`, `BTwoText`, `GTwoText` and `CTwoText` commands. The fonts that can be used are the 35 fonts that are made available by Adobe and that are normally available in all postscript interpreters, including printers. The fonts, together with the names used to specify them in the normal font-setting commands of TeX and LaTeX, are shown in Table 1.

If you prefer to use the normal document font (which would normally be Computer Modern in the common document classes), you simply leave the fontname empty, e.g,.

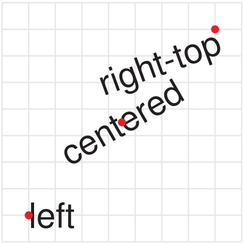
```
\SetPFont{}{20}
```

As for the second, fontsize argument, leaving it empty uses the size that LaTeX is using at the moment the text-drawing command starts, e.g.,

Font name	LaTeX	Font name	LaTeX
AvantGarde-Book	pagk	Helvetica-Narrow	phvrrn
AvantGarde-BookOblique	pagko	Helvetica-NarrowOblique	phvron
AvantGarde-Demi	pagd	NewCenturySchlbk-Bold	pncb
AvantGarde-DemiOblique	pagdo	NewCenturySchlbk-BoldItalic	pncbi
Bookman-Demi	pbkd	NewCenturySchlbk-Italic	pncri
Bookman-DemiItalic	pbkdi	NewCenturySchlbk-Roman	pncr
Bookman-Light	pbkl	Palatino-Bold	pplb
Bookman-LightItalic	pbkli	Palatino-BoldItalic	pplbi
Courier-Bold	pcrb	Palatino-Italic	pplri
Courier-BoldOblique	pcrbo	Palatino-Roman	pplr
Courier	pcrr	Symbol	psyr
Courier-Oblique	pcrro	Times-Bold	ptmb
Helvetica-Bold	phvb	Times-BoldItalic	ptmbi
Helvetica-BoldOblique	phvbo	Times-Italic	ptmri
Helvetica-NarrowBold	phvbrn	Times-Roman	ptmr
Helvetica-NarrowBoldOblique	phvbon	ZapfChancery-MediumItalic	pzcmi
Helvetica	phvr	ZapfDingbats	pzdr
Helvetica-Oblique	phvro		

Table 1: Available postscript fonts and their corresponding names in LaTeX.

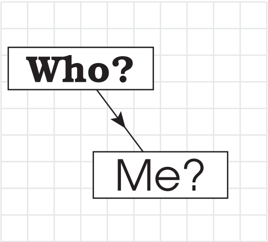
```
\SetPFont{Helvetica-Bold}{}
```



```
\SetPFont{Helvetica}{13}
\PText(10,10)(0)[l]{left}
\PText(45,45)(30)[]{centered}
\PText(80,80)(20)[rt]{right-top}
\SetColor{Red}
\Vertex(10,10){1.5}
\Vertex(45,45){1.5}
\Vertex(80,80){1.5}
```

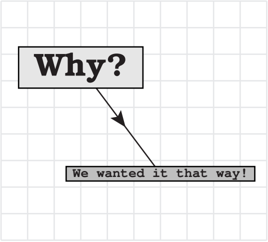
The `\PText` command writes in Axodraw's current Postscript font. The first two arguments give the focal point, the third argument is a rotation angle and the fourth argument is as in the various TeX/LaTeX box commands to indicate left, right, top or bottom adjustment with respect to the focal point. No indication means centered.

Note that use of normal LaTeX font setting commands or of math-mode will not normally have the desired effect.



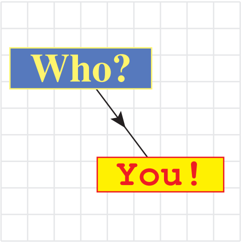
```
\ArrowLine(30,65)(60,25)
\SetPFont{Bookman-Demi}{14}
\BText(30,65){Who?}
\SetPFont{AvantGarde-Book}{16}
\BText(60,25){Me?}
```

The `\BText` command writes a centered box with text in it. It uses Axodraw's current Postscript font.



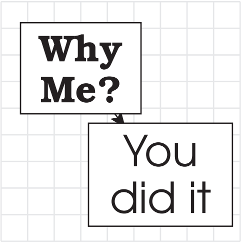
```
\ArrowLine(30,65)(60,25)
\SetPFont{Bookman-Demi}{12}
\GText(30,65){0.9}{Why?}
\SetPFont{Courier-Bold}{5}
\GText(60,25){0.75}{We wanted it that way!}
```

The `\GText` command writes a centered box with text in it. It uses Axodraw's current Postscript font. The third argument is the grayscale with which the box will be filled. 0 is black and 1 is white.



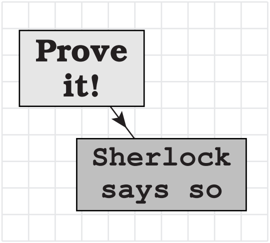
```
\ArrowLine(30,65)(60,25)
\SetPFont{Times-Bold}{15}
\CText(30,65){LightYellow}{LightBlue}{Who?}
\SetPFont{Courier-Bold}{14}
\CText(60,25){Red}{Yellow}{You!}
```

The `\CText` command writes a centered box with text in it. It uses Axodraw's current Postscript font. The third argument is the color of the box and the text. The fourth argument is the color with which the box will be filled.



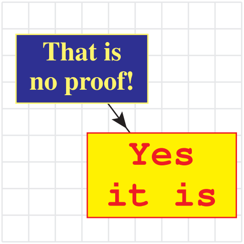
```
\ArrowLine(30,65)(60,25)
\SetPFont{Bookman-Demi}{14}
\BTwoText(30,65){Why}{Me?}
\SetPFont{AvantGarde-Book}{16}
\BTwoText(60,25){You}{did it}
```

The `\BTwoText` command writes a centered box with two lines of text in it. It uses Axodraw's current Postscript font.



```
\ArrowLine(30,65)(60,25)
\SetPFont{Bookman-Demi}{12}
\GTwoText(30,65){0.9}{Prove}{it!}
\SetPFont{Courier-Bold}{11}
\GTwoText(60,25){0.75}{Sherlock}{says so}
```

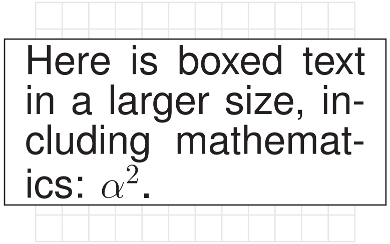
The `\GTwoText` command writes a centered box with two lines of text in it. It uses Axodraw's current Postscript font. The third argument is the grayscale with which the box will be filled. 0 is black and 1 is white.



```
\ArrowLine(30,65)(60,25)
\SetPFont{Times-Bold}{10}
\CTwoText(30,65){LightYellow}{Blue}
{That is}{no proof!}
\SetPFont{Courier-Bold}{14}
\CTwoText(60,25){Red}{Yellow}{Yes}{it is}
```

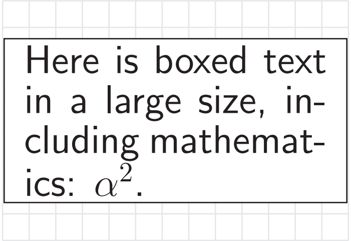
The `\CTwoText` command writes a centered box with two lines of text in it. It uses Axodraw's current Postscript font. The third argument is the color of both the box and the text. The fourth argument is the color with which the box will be filled.

Note that because you can now use LaTeX commands for the text arguments of the commands described in this section, the effects of the `\BTwoText`, `\GTwoText`, and `\CTwoText` can be achieved also by the use of regular `\BText` etc commands. Mathematics can also be used. (None of these was possible in v. 1 of axodraw.) Here are some examples:



```
\SetPFont{Helvetica}{15}
\BText(70,45){%
    \begin{minipage}{4.5cm}
      Here is boxed text in a
      larger size, including
      mathematics: $\alpha^2$.
    \end{minipage}%
}
```

This example shows that the `\BText` command can also be used with minipages and other LaTeX methods to make more complicated boxed texts.



Here is boxed text in a large size, including mathematics: α^2.

```
\SetPFont{}{15}
\BText(65,45){%
    \begin{minipage}{4cm}
      \sffamily Here is boxed text in a
      large size, including
      mathematics: $\alpha^2$.
    \end{minipage}%
}
```

But if you use mathematics, the text may be more elegant if you use the document font, which has matching fonts for text and mathematics. Use of a sans-serif font (by `\sffamily`) may be better in a diagram.

5.5 Options

Almost all of axodraw2's line-drawing commands take optional arguments. The form here is familiar from many standard LaTeX commands. The optional arguments are placed in square brackets after the command name, and are made of a comma-separated list of items of the form: `keyword` or `keyword=value`. The required arguments are placed afterwards.

Optional arguments can be used to set particular characteristics of a line, e.g., whether it is dashed or has an arrow. They can also be used to set some of the line's parameters, to be used instead of default values. (The default values can be adjusted by commands listed in Sec. 5.9. Those commands are useful for adjusting parameters that apply to multiple lines, while the optional arguments are useful for setting parameters for individual lines.)

The original axodraw only had different command names to determine whether lines were dashed, or had arrows, etc. The new version retains these commands, but now the basic commands (`\Line`, `\Arc`, `\Gluon`, etc) can also be treated as generic commands, with the different varieties (dashed, double, and/or with an arrow) being set by options.

The same set of options are available for all types of line. However, not all apply or are implemented for particular types of line. Thus, `clockwise` is irrelevant for a straight line, while `arrow` is not implemented for gluons, photons and zigzag lines. Warnings are given for unimplemented features, while inapplicable arguments are ignored.

The full set of options.

color=⟨colorname⟩	Set the line in this color.
colour=⟨colorname⟩	Same as color=⟨colorname⟩.
dash	Use a dashed line.
dsize=⟨number⟩	Set the dash size (when a line is dashed).
dashsize=⟨number⟩	Same as dsize=⟨number⟩.
double	Use a double line.
sep=⟨number⟩	Sets the separation for a double line.
linesep=⟨number⟩	Same as sep=⟨number⟩.
width=⟨number⟩	Sets line width for this line only.
clock	For arcs, makes the arc run clockwise.
clockwise	For arcs, makes the arc run clockwise.
arrow	Use an arrow.
flip	If there is an arrow, its direction is flipped.
arrowpos=⟨number⟩	The number should be between zero and one and indicates where along the line the arrow should be. 1 is at the end. 0.5 is halfway (the initial default).
arrowaspect=⟨number⟩	See Sec. 5.7.
arrowlength=⟨number⟩	See Sec. 5.7.
arrowheight=⟨number⟩	See Sec. 5.7.
arrowinset=⟨number⟩	See Sec. 5.7.
arrowscale=⟨number⟩	See Sec. 5.7.
arrowstroke=⟨number⟩	See Sec. 5.7.
arrowwidth=⟨number⟩	See Sec. 5.7.
inset=⟨number⟩	Same as arrowinset.

The options without an extra argument, e.g., `arrow`, are actually of a boolean type. That is, they can also be used with a suffix "`=true`" or "`=false`", e.g., `arrow=true` or `arrow=false`.

If an option is not provided, its default value is used. Defaults are no dashes, no double lines, anticlockwise arcs, no arrow and if an arrow is asked for, its position is halfway along the line. Other arrow settings are explained in Sec. 5.7. There are also default values for dash size (3) and the separation of double lines (2).

The full set of the generic line commands with their syntax is

```
\Line[options](x1,y1)(x2,y2)
\Arc[options](x,y)(r,theta1,theta2)
\Bezier[options](x1,y1)(x2,y2)(x3,y3)(x4,y4)
\Gluon[options](x1,y1)(x2,y2){amplitude}{windings}
\GluonArc[options](x,y)(r,theta1,theta2){amplitude}{windings}
\GluonCirc[options](x,y)(r,phase){amplitude}{windings}
\Photon[options](x1,y1)(x2,y2){amplitude}{windings}
\PhotonArc[options](x,y)(r,theta1,theta2){amplitude}{windings}
\ZigZag[options](x1,y1)(x2,y2){amplitude}{windings}
\ZigZagArc[options](x,y)(r,theta1,theta2){amplitude}{windings}
```

The applicability of the options is as follows

	Arrow, etc	Clockwise
`\Line`	Y	N
`\Arc`	Y	Y
`\Bezier`	Y	N
`\Gluon`	N	N
`\GluonArc`	N	Y
`\GluonCirc`	N	N
`\Photon`	N	N
`\PhotonArc`	N	Y
`\ZigZag`	N	N
`\ZigZagArc`	N	Y

The arrow options include those for setting the arrow dimensions. Options not indicated in the last table apply to all cases.

Some examples are:

```
\Line[double,sep=1.5,dash,dsize=4](10,10)(70,30)
\Line[double,sep=1.5,arrow,arrowpos=0.6](10,10)(70,30)
```

The options can also be used on the more explicit commands as extra options. Hence it is possible to use

```
\DoubleLine[dash,dsize=4](10,10)(70,30){1.5}
```

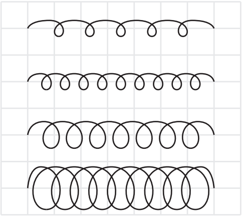
instead of the first line in the previous example.

One may notice that some of the options are not accessible with the more explicit commands. For example, it is possible to put arrows on Bézier curves only by using the option ‘arrow’ for the Bézier command.

5.6 Remarks about Gluons

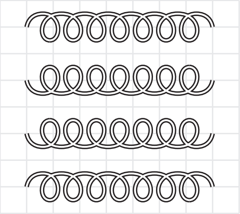
There are 12 commands that concern gluons. This allows much freedom in developing one’s own style. Gluons can be drawn as single solid lines, as double lines, as dashed lines and as dashed double lines.

Gluons have an amplitude and a number of windings. By varying these quantities one may obtain completely different gluons as in:

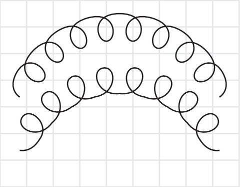


```
\Gluon(10,70)(80,70){3}{5}
\Gluon(10,50)(80,50){3}{9}
\Gluon(10,30)(80,30){5}{7}
\Gluon(10,10)(80,10){8}{9}
```

One may change the orientation of the windings by reversing the direction in which the gluon is drawn and/or changing the sign of the amplitude:



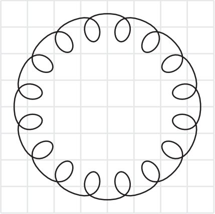
```
\DoubleGluon(10,70)(80,70){5}{7}{1.2}
\DoubleGluon(80,50)(10,50){5}{7}{1.2}
\DoubleGluon(10,30)(80,30){-5}{7}{1.2}
\DoubleGluon(80,10)(10,10){-5}{7}{1.2}
```



```
\GluonArc(45,20)(40,20,160){5}{8}
\GluonArc(45,0)(40,20,160){-5}{8}
```

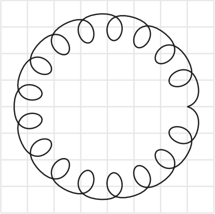
Here one can see that the sign of the amplitude gives a completely different aspect to a gluon on an arc segment.

There are two ways of drawing a gluon circle. One is with the command GluonCirc and the other is an arc of 360 degrees with the GluonArc command. The second way has a natural attachment point, because the GluonArc command makes gluons with a begin- and endpoint.



```
\GluonCirc(40,40)(30,0){5}{16}
```

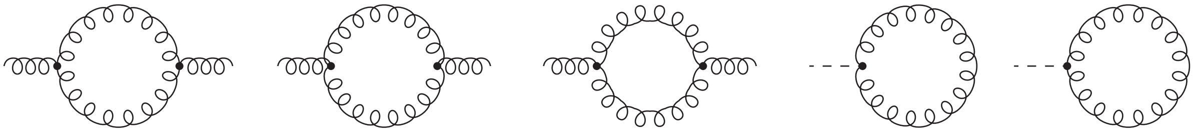
This is the 'complete circle'. If one likes to attach one or more lines to it one should take into account that the best places for this are at a distance radius+amplitude from the center of the circle. One can rotate the circle by using the phase argument.



```
\GluonArc(40,40)(30,0,360){5}{16}
```

In the 360 degree arc there is a natural point of attachment. Of course there is only one such point. If one needs more than one such point one should use more than one arc segment.

Some examples are:

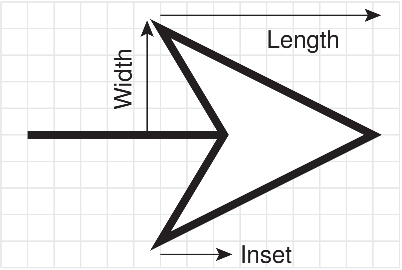


This picture was generated with the code:

```
\begin{center} \begin{axopicture}{(460,60)(0,0)}
   \Gluon(7,30)(27,30){3}{3}
   \GluonCirc(50,30)(20,0){3}{16}
   \Gluon(73,30)(93,30){3}{3}
   \Vertex(27,30){1.5}
   \Vertex(73,30){1.5}
   \Gluon(110,30)(130,30){3}{3}
   \GluonArc(150,30)(20,0,180){3}{8}
   \GluonArc(150,30)(20,180,360){3}{8}
   \Gluon(170,30)(190,30){3}{3}
   \Vertex(130,30){1.5}
   \Vertex(170,30){1.5}
   \Gluon(210,30)(230,30){3}{3}
   \GluonArc(250,30)(20,0,180){-3}{8}
   \GluonArc(250,30)(20,180,360){-3}{8}
   \Gluon(270,30)(290,30){3}{3}
   \Vertex(230,30){1.5}
   \Vertex(270,30){1.5}
   \DashLine(310,30)(330,30){3}
   \GluonArc(350,30)(20,-180,180){3}{16}
   \Vertex(330,30){1.5}
   \DashLine(387,30)(407,30){3}
   \GluonCirc(430,30)(20,0){3}{16}
   \Vertex(407,30){1.5}
\end{axopicture} \end{center}
```

5.7 Remarks about arrows

The old Axodraw arrows were rather primitive little triangles. The JaxoDraw program has introduced fancier arrows which the user can also customize. There are parameters connected to this as shown in the figure:



```
\Line[arrow,arrowinset=0.3,arrowaspect=1,arrowwidth=40,arrowpos=1,
                  arrowstroke=3](10,50)(100,50)
```

The full set of parameters is:

aspect A multiplicative parameter when the length is calculated from the width. The normal formula is: length = 2 × width × aspect.

inset The fraction of the length that is taken inward.

length The full length of the arrowhead.

position The position of the arrow in the line as a fraction of the length of the line.

scale A scale parameter for the complete arrowhead.

stroke The width of the line that makes up the arrowhead. If the value is not set (default value is zero) the arrow is filled and overwrites whatever was there. In the case of a stroke value the contents are overwritten in the background color.

width The half width of the arrowhead.

The parameters can be set in two ways. One is with one of the commands

`\SetArrowScale{number}`	Initial value is 1.
`\SetArrowInset{number}`	Initial value is 0.2
`\SetArrowAspect{number}`	Initial value is 1.25
`\SetArrowPosition{number}`	Initial value is 0.5
`\SetArrowStroke{number}`	Initial value is 0

(A complete list of commands for setting defaults is in Sec. 5.9.) These commands determine settings that will hold for all following commands, up to the end of whatever LaTeX or TeX grouping the default setting is given in. E.g., setting a default value inside an `axopicture` environment sets it until the end of the environment only. (Thus the settings obey the normal rules of LaTeX for scoping.)

The other way is to use one or more of these parameters as options in a command that uses an arrow. The general use of options is in Sec. 5.5. The options that are available are

arrow	initial default=false
arrowscale=⟨number⟩	initial default=1
arrowwidth=⟨number⟩	initial default=0
arrowlength=⟨number⟩	initial default=0
arrowpos=⟨number⟩	initial default=0.5
arrowinset=⟨number⟩	initial default=0.2
arrowstroke=⟨number⟩	initial default=0
arrowaspect=⟨number⟩	initial default=1.25
flip	initial default=false

The arrow option tells the program to draw an arrow. Without it no arrow will be drawn. The flip option indicates that the direction of the arrow should be reversed from the 'natural' direction.

When neither the width nor the length are specified, but instead both are given as zero, they are computed from the line width (and the line separation when there is a double line). The formula is:

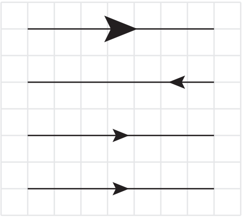
$$\begin{aligned} \text{Arrowwidth} &= 1.2 \times (\text{linewidth} + 0.7 \times \text{separation} + 1) \times \text{arrowscale}, \qquad (1) \\ \text{Length} &= 2 \times \text{arrowwidth} \times \text{arrowaspect}. \qquad (2) \end{aligned}$$

If, however, $1.2 \times (\text{linewidth} + 0.7 \times \text{separation} + 1)$ is less than 2.5, the formula for the arrow width becomes $\text{arrowwidth} = 2.5 \times \text{arrowscale}$.

If only one of the arrowwidth or the arrowlength parameters is zero, it is computed from the other non-zero parameter using formula (2). When both are non-zero, those are the values that are used.

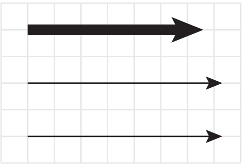
The position of the arrowhead is a bit tricky. The arrowpos parameter is a fraction of the length of the line and indicates the position of the center of the arrowhead. This means that when arrowpos is one, the arrowhead sticks out beyond the end of the line by half the arrowlength. When for instance the line width is 0.5, the default length of the arrowhead defaults to 6.25. Hence if one would like to compensate for this one should make the line 3.125 points shorter. Usually 3 pt will be sufficient.

Because of backward compatibility axodraw2 has many individual commands for lines with arrows. We present them here, together with some 'options' varieties.



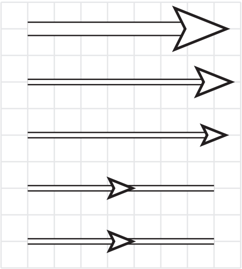
```
\Line[arrow,arrowscale=2](10,70)(80,70)
\Line[arrow,arrowpos=0.8,flip](10,50)(80,50)
\Line[arrow](10,30)(80,30)
\ArrowLine(10,10)(80,10)
```

The default position for the arrow is halfway (arrowpos=0.5). With the line command and the options we can put the arrow in any position.



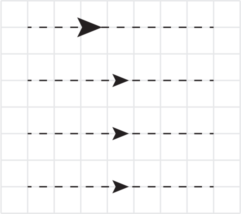
```
\Line[arrow,arrowpos=1](10,30)(80,30)
\LongArrow(10,10)(80,10)
\SetWidth{4}
\LongArrow[arrowscale=0.8](10,50)(70,50)
```

The `\LongArrow` command just places the arrowhead at the end of the line. The size of the arrowhead is a function of the linewidth.



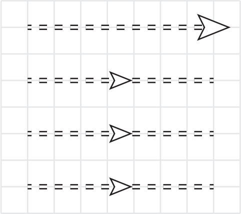
```
\SetArrowStroke{1}
\Line[arrow,arrowpos=1,double,sep=5,arrowscale=1.3]
    (10,90)(75,90)
\Line[arrow,arrowpos=1,double,sep=2,arrowscale=1.5]
    (10,70)(80,70)
\Line[arrow,arrowpos=1,double,sep=2](10,50)(80,50)
\Line[arrow,double,sep=2](10,30)(80,30)
\ArrowDoubleLine(10,10)(80,10){2}
```

As one can see, the arrows also work with double lines.



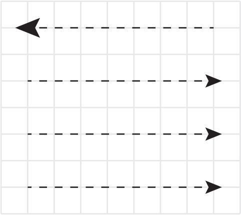
```
\Line[arrow,arrowpos=0.3,dash,dsize=3,arrowscale=1.5]
(10,70)(80,70)
\DashArrowLine(10,50)(80,50){3}
\Line[arrow,dash,dsize=3](10,30)(80,30)
\ArrowDashLine(10,10)(80,10){3}
```

We have not taken provisions for the dashes to be centered in the arrowhead, because at times that is nearly impossible. The commands `\ArrowDashLine` and `\DashArrowLine` are identical.



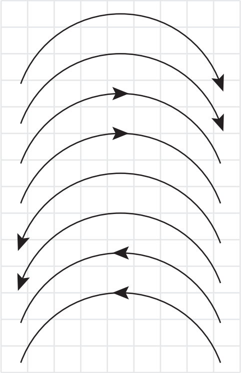
```
\SetArrowStroke{0.5}
\Line[arrow,arrowpos=1,dash,dsize=3,double
,sep=1.5,arrowscale=1.5](10,70)(80,70)
\DashArrowDoubleLine(10,50)(80,50){1.5}{3}
\Line[arrow,dash,dsize=3](10,30)(80,30)
\ArrowDashDoubleLine(10,10)(80,10){1.5}{3}
```

The `\ArrowDashDoubleLine` and `\DashArrowDoubleLine` commands are identical.



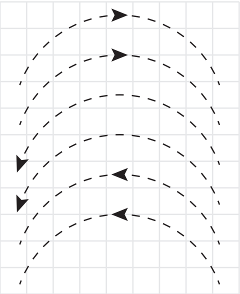
```
\Line[arrow,arrowpos=0,dash,dsize=3,arrowscale=1.5
,flip](10,70)(80,70)
\DashLongArrowLine(10,50)(80,50){3}
\Line[arrow,arrowpos=1,dash,dsize=3](10,30)(80,30)
\LongArrowDashLine(10,10)(80,10){3}
```

The commands `\LongArrowDashLine`, `\DashLongArrowLine`, `\LongArrowDash` and `\DashLongArrow` are identical.



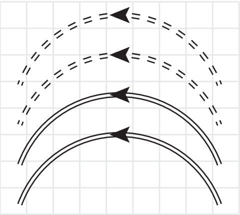
```
\Arc[arrow,arrowpos=0,flip](45,95)(40,20,160)
\LongArrowArcn(45,80)(40,20,160)
\Arc[arrow,arrowpos=0.5](45,65)(40,20,160)
\ArrowArcn(45,50)(40,20,160)
\Arc[arrow,arrowpos=1](45,35)(40,20,160)
\LongArrowArc(45,20)(40,20,160)
\Arc[arrow,arrowpos=0.5](45,5)(40,20,160)
\ArrowArc(45,-10)(40,20,160)
```

The `Arc` and the `CArc` commands are identical.



```
\Arc[arrow,dash,dsize=3,arrowpos=0.5]
(45,65)(40,20,160)
\ArrowDashArcn(45,50)(40,20,160){3}
\Arc[arrow,dash,dsize=3,arrowpos=1]
(45,35)(40,20,160)
\LongArrowDashArc(45,20)(40,20,160){3}
\Arc[arrow,dash,dsize=3,arrowpos=0.5]
(45,5)(40,20,160)
\ArrowDashArc(45,-10)(40,20,160){3}
```

The `DashArrowArc` and the `ArrowDashArc` commands are identical. So are the commands `DashArrowArcn` and `ArrowDashArcn`.



```
\Arc[arrow,dash,dsize=3,double,sep=1.5
,arrowpos=0.5](45,35)(40,20,160)
\ArrowDashDoubleArc(45,20)(40,160,20){1.5}{3}
\Arc[arrow,double,sep=1.5,arrowpos=0.5]
(45,5)(40,20,160)
\ArrowDoubleArc(45,-10)(40,20,160){1.5}
```

Other commands involving Long do not exist. The options can take care of their functionality.

Computing the position of the arrow in a Bézier curve is a bit complicated. Let us recall the definition of a cubic Bézier curve:

$$\begin{aligned} x &= x_0(1-t)^3 + 3x_1t(1-t)^2 + 3x_2t^2(1-t) + x_3t^3 \\ y &= y_0(1-t)^3 + 3y_1t(1-t)^2 + 3y_2t^2(1-t) + y_3t^3 \end{aligned} \tag{3}$$

Computing the length of the curve is done with the integral

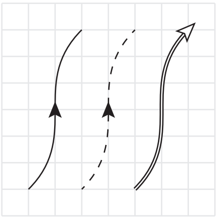
$$L = \int_0^1 dt \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}, \tag{4}$$

which is an integral over the square root of a quartic polynomial. This we do with a 16 point Gaussian quadrature and it gives us more than enough accuracy[4]. Let us assume now that we want the arrow at 0.6 of the length. To find the exact fraction of the length involves finding the upper limit of the integral for which the length is $0.6L$. This requires an iteration procedure till we have a reasonable accuracy for the position (x, y). After that we have to calculate the derivative in this point as well.

Because the Bézier curves are new commands in axodraw2 there is no need for backwards compatibility in the use of arrows. Hence all arrow commands are done by means of the options. Some examples are:

---

[4] We need to compute the length of the Bézier curve also when we want to put a dash pattern on it. The exact dash size is determined such that an integer number of patterns fits in the line.



```
\Bezier[arrow](10,10)(30,30)(10,50)(30,70)
\Bezier[arrow,dash,dsize=3](30,10)(50,30)
(30,50)(50,70)
\Bezier[arrow,arrowpos=1,double,sep=1,arrowstroke
=0.5](50,10)(70,30)(50,50)(70,70)
```

5.8 Units and scaling

When you have constructed a diagram, you may need to change its scale, to make it larger or smaller. Axodraw2 provides ways of doing this, for scaling diagrams without recoding all the individual coordinates. However the requirements for the nature of the scaling change between different cases. For example, suppose a diagram is designed for use in a journal article and you wish to use it in the slides for a seminar. Then you will want to enlarge both the geometric size of the diagram's objects and the text labels it contains. But if you wish to use a scaled diagram in another place in a journal article, you will wish to scale its lines etc, but will probably not wish to scale the text (to preserve its legibility).

Axodraw2 therefore provides tools for the different situations, so we will now explain what to do. The commands to achieve this all appear in the list of parameter-setting commands in Sec. 5.9.

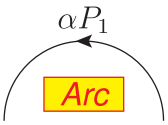
5.8.1 Scaling for slides

Suppose the original diagram is

```
\SetPFont{Helvetica-Oblique}{12}
Document text.  Then diagram:
\begin{axopicture}(60,43)
   \Arc[arrow](30,0)(30,0,180)
   \Text(30,33)[b]{$\alpha P_1$}
   \CText(30,10){Red}{Yellow}{Arc}
\end{axopicture}
```

to give

Document text. Then diagram:

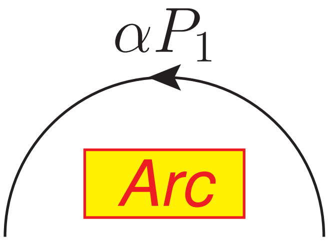


Then you could double the scale of the diagram by

```
\SetScale{2}
\fontsize{24}{26}\selectfont
\SetPFont{Helvetica-Oblique}{12}
Document text.  Then diagram:
\begin{axopicture}(60,43)
   \Arc[arrow](30,0)(30,0,180)
   \Text(30,33)[b]{$\alpha P_1$}
   \CText(30,10){Red}{Yellow}{Arc}
\end{axopicture}
```

to get

Document text. Then diagram:



We have changed the size of the document font, as would be appropriate for a make slides for a presentation; this we did by the `\fontsize` command. The arc and the space inserted in the document for the diagram have scaled up. The label inserted by the `\Text` command has changed to match the document font. The postscript text in the `\CText` was specified to be at 12 pt, but is now scaled up also.

The above behavior is what axodraw2 does by default, and is what v. 1 did.

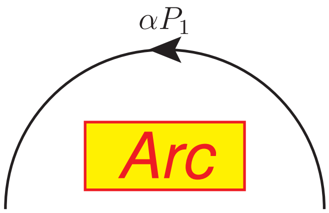
5.8.2 Scaling within article

If you wanted to make an enlarged figure in a journal article, you would not change the document font. But the obvious modification to the previous example is

```
\SetScale{2}
\SetPFont{Helvetica-Oblique}{12}
Document text.  Then diagram:
\begin{axopicture}(60,43)
   \Arc[arrow](30,0)(30,0,180)
   \Text(30,33)[b]{$\alpha P_1$}
   \CText(30,10){Red}{Yellow}{Arc}
\end{axopicture}
```

which gives

Document text. Then diagram:

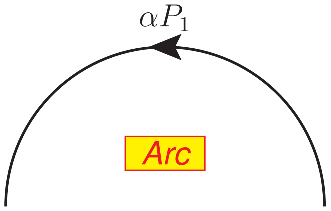


The label αP_1 is now not enlarged, since it copies the behavior of the document font. But the postscript text is enlarged, which is probably undesirable. If you were scaling down the diagram instead of scaling it up, the situation would be worse, because the postscript font would be difficult to read.

So in this situation, of scaling the diagram while keeping the document font intact, you probably also want to leave unchanged the size of the postscript font. You can achieve this by the `\PSTextScalesLikeGraphicsfalse` command:

```
\SetScale{2}
\PSTextScalesLikeGraphicsfalse
\SetPFont{Helvetica-Oblique}{12}
Document text.  Then diagram:
\begin{axopicture}(60,43)
   \Arc[arrow](30,0)(30,0,180)
   \Text(30,33)[b]{$\alpha P_1$}
   \CText(30,10){Red}{Yellow}{Arc}
\end{axopicture}
```

Document text. Then diagram:



To achieve this on a document-wide basis, which is probably what you want, you can use the `PStextScalesIndependently` option when you load axodraw2 — see Sec. 5.1.

Nevertheless, if you turn off the default scaling of postscript text, you may still want to scale text. For this you can use the `\SetTextScale` command, as in `\SetTextScale{1.2}`. This only has an effect when you have turned off the scaling of postscript text with graphics objects; but then it applies to TEX text inserted by axodraw2's `\Text` and `\rText` commands, as well text inserted by axodraw2's "postscript-text" commands.

If you are confused by the above, we recommend experimentation to understand how to achieve the effects that you specifically need. We could have made the set of commands and options simpler, but only at the expense of not being able to meet the demands of the different plausible situations that we could imagine and have to deal with ourselves.

5.8.3 Canvas and object scales

When you use `\SetScale` outside an `axopicture` environment, as above, the scaling applies to both the axodraw2 objects and the space inserted for the `axopicture` environment in the document, as is natural. But you may find you need to scale a subset of objects inside the diagram, e.g.,

```
\begin{axopicture}(...)
      (First block)
\SetScale{0.5}
      (Second block)
\end{axopicture}
```

In this case, the units for specifying the objects in the second block are different from those for specifying the `axopicture` environment's size (as well as the first block of objects). We thus distinguish object units from canvas units, where "canvas" refers to the `axopicture` environment as a whole.

Another complication is that the LaTeX `picture` environment has is own `\unitlength` parameter. In v. 1 of axodraw, the canvas scale was determined by LaTeX's `\unitlength`. But there was an independent unit for the object scale; this was the one determined by axodraw's `\SetScale` command. Also, not all objects used the object scale. The situation therefore got quite confusing. In v. 1, if, as is often natural, you wished to scale the canvas as well as the objects, you would have needed to set LaTeX's `\unitlength` parameter as well as using axodraw's `\SetScale` command.

So now we have arranged things so that the canvas and object scales are tied by default, provided that you use axodraw2's `\SetScale` command, and that axodraw diagrams are inside `axopicture` environments (in contrast to the `picture` environment used in the original axodraw). However, it may be necessary to keep backward compatibility in some cases, and we weren't certain that the new behavior is exactly what is always desired. So in axodraw2, we have provided three choices, given by the `canvasScaleIs1pt`, `canvasScaleIsObjectScale`, and `canvasScaleIsUnitLength` options when loading axodraw2 — see Sec. 5.1. Naturally, `canvasScaleIsObjectScale` is the default. If you wish to change the setting mid-document, there are corresponding commands — Sec. 5.9.

5.9 Settings

Axodraw2 has a number of parameters that can be set by the user. The parameters include defaults for line types, dimensions, etc. The parameters can be set either inside the axopicture environment or outside. If they are set outside they modify the default value for subsequent pictures. If set inside they only affect the current picture. (In general, the parameters obey the usual rules for the scope of LaTeX variables.) In many cases, the parameters provide default values for a command to draw an object and can be overridden for a single object by using an optional parameter in invoking the command for the object.

The unit for lengths is the current object scale, as set by the `\SetScale` command.

The parameter-setting commands are:

Command	Commentary
Lines:	
`\SetDashSize{#1}`	This sets the default size for the size of the dashes of dashed lines. Its initial value is 3.
`\SetLineSep{#1}`	This sets the default separation of double lines. Its initial value is 2.
`\SetWidth{#1}`	This sets the default width of lines. Its initial value is 0.5.
Arrows:	
`\SetArrowAspect{#1}`	See Sec. 5.7.
`\SetArrowInset{#1}`	See Sec. 5.7.
`\SetArrowPosition{#1}`	Determines where the arrowhead is on a line. The position is the fraction of the length of the line.
`\SetArrowScale{#1}`	A scale parameter for the entire head of an arrow.
`\SetArrowStroke{#1}`	This parameter determines the linewidth of the arrowhead if it is just outlined. Its initial value is zero (filled arrowhead).
Scaling:	
`\canvasScaleOnept`	Sets canvas scale to 1 pt.
`\canvasScaleObjectScale`	Sets canvas scale to equal the value set by `\SetScale` in units of points. This is the initial default of axodraw2, unless overridden.
`\canvasScaleUnitLength`	The canvas scale is the same as LaTeX's length parameter `\unitlength`.
`\SetScale{#1}`	This sets a scale factor. This factor applies a magnification factor to all axodraw2 graphics objects. When the setting that postscript-text-scales-like-graphics is set (as is true by default), it also applies to axodraw2's "postscript-text" writing commands (`\PText`, `\BText`, etc), but not to its TeX-text commands (`\Text` etc). The initial scale factor is unity.

`\SetTextScale{#1}`	This factor applies a magnification factor to all axodraw2 text objects, but *only when* the setting that postscript-text-scales-like-graphics is turned off.
`\PSTextScalesLikeGraphicsfalse`	Text drawn by all of Axodraws's text commands scales with the factor set by `\SetTextScale`. See Sec. 5.4.
`\PSTextScalesLikeGraphicstrue`	(Default setting.) Text drawn by Axodraw's postscript-text commands scales with the same factor as graphics objects, as set by `\SetScale`. Text drawn by Axodraw's TeX-text commands is unscaled. See Sec. 5.4.

Offsets:

`\SetOffset(#1,#2)`	Sets an offset value for all commands of axodraw2. Its value is not affected by the scale variable.
`\SetScaledOffset(#1,#2)`	Sets an offset for all commands of axodraw2. This offset is affected by the scale factor.

Color:

`\SetColor{#1}`	Sets the named color, for both axodraw2 objects and regular text. See Sec. 5.10 for details on using color with axodraw2.
`\textRed`	Alternative command for setting named a color for both axodraw2 objects and regular text. See Sec. 5.10 for details on using color with axodraw2. There is one such command for each axodraw2 named color.

Font:

`\SetPFont{#1}{#2}`	Sets the Postscript font, and its size in units of points. See Sec. 5.4.2 for the commands that use this font, for a table of the names of the fonts. An empty first argument, instead of a font name, (as in `\SetPFont{}{20}` indicates that the normal document font is to be used at the indicated size. An empty second argument, instead of the font size, (as in `\SetPFont{Helvetica}{}` or `\SetPFont{}{}`) indicates that the font size is to be LaTeX's document font size at the time the text-making command is executed.

5.10 Colors

TeX and LaTeX by themselves do not provide any means to set colors in a document. Instead, one must use a suitable package to achieve the effect; the current standard one is `color.sty`. Such a package performs its work by passing graphics commands to the viewable output file. Since axodraw also works in a similar fashion, there is a potentiality for conflicts.

Axodraw version 1, released in 1994, used the package `colordvi.sty` for applying color to normal textual material, and its own separate methods for applying color to its graphical objects. They both defined the same convenient set of named colors that could be used, but they had to be set separately for text and graphics[5]. The `colordvi.sty` package also had an important disadvantage that its color settings did not respect TeX grouping and LaTeX environments, so that a color setting made for text in an environment continued to apply after the end of the environment.

Since then, the available tools, notably in the powerful `color.sty`, have greatly improved. But this has introduced both real and potential incompatibilities with the older methods. Note that `color.sty` is currently the most standard way for implementing color, and is a required part of LaTeX distributions, as part of the graphics bundle.

In the new version of axodraw, we have arranged to have compatibility with `color.sty`, while allowing as much backward compatibility as we could with the user interface from v. 1. We fully rely on `color.sty` for setting color[6]. But to keep the best of the old methods, we have defined all the named colors that were defined in the old version, together with a few extra ones. We have also defined color-setting commands in the style of `colordvi.sty`, but they now apply uniformly to both text and axodraw graphical objects, and they respect TeX and LaTeX grouping and environments.

This results in some changes in behavior in certain situations. We think the new behavior is more natural from the user's point of view; but it is a change.

There are two classes of graphics-drawing command in axodraw. One class has no explicit color argument, and uses the currently set color; the line-drawing commands are typical of these. Other commands have explicit color arguments, and these arguments are named colors. The named colors are a union of those axodraw defines, with those defined by `color.sty` together with any further ones defined by the user.

5.10.1 How to use colors

Axodraw works with named colors — see Sec. 5.10.2 — which are a standard set of 68 originally defined by the `dvips` program and the `colordvi.sty`, plus 5 extra colors defined in axodraw2. (In addition there are several named colors that are normally defined by default by `color.sty`, and that can also be used.)

---

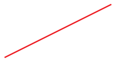
[5] The named colors corresponded to ones defined by the `dvips` program.

[6] Except for certain hard wired settings in double lines and stroked arrows.

To use them we have several possibilities to specify colors. Which to use is mostly a matter of user preference or convenience.

- The axodraw command `\SetColor{colorname}`: sets the color to be the named color for everything until the end of the current environment (or TEX group, as relevant.) The initial default color is Black, of course. An example:

  Now red is used:

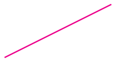
  ```
  \SetColor{Red}
  Now red is used:\\
  \begin{axopicture}(0,40)
      \Line(0,10)(40,30)
  \end{axopicture}
  ```

- Completely equivalently, one can use the command `\color{colorname}` defined by the standard `color.sty` package, with any of its options, e.g., `\color{Red}` or `\color[rgb]{1,0,0}`. In fact `\SetColor` is now a synonym for `\color`, retained for backward compatibility.

- The named colors defined by axodraw2 are listed in Sec. 5.10.2. Extra ones can be defined by axodraw2's `\newcolor` command.

- For each of the named colors defined by axodraw2 (and others defined by the use of the `\newcolor` command), there is a macro whose name is "text" followed by the color name, e.g., `\textMagenta`. This behaves just like the corresponding call to `\SetColor` or `\color`. Thus we have

  Now magenta is used:

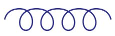
  

  ```
  \textMagenta
  Now magenta is used:\\
  \begin{axopicture}(0,40)
      \Line(0,10)(40,30)
  \end{axopicture}
  ```

  These macros correspond to macros defined by the venerable `colordvi.sty` package, but now have what is normally an advantage that their scope is delimited by the enclosing environment.

  Normal text, then

  Large, bold blue

  And normal text afterward.

  ```
  Normal text, then
  \begin{center}
      \Large \bf \color{Blue}
      Large, bold blue
      \begin{axopicture}(40,20)
        \Gluon(0,10)(40,10){4}{4}
      \end{axopicture}\\
  \end{center}
  And normal text afterward.
  ```

- A delimited section of text can be set in a color by using a macro named by the color (e.g., `\Red`):

  In the middle of black text, red text and gluon. Then continue …

```
In the middle of black text,
\Red{red text and
  \begin{axopicture}(30,10)
      \Gluon(0,5)(30,5){3}{4}
  \end{axopicture}\
  gluon%
}.
Then continue \dots
```

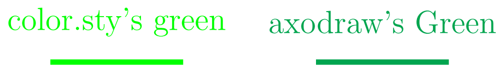
  These macros correspond to macros defined by the `colordvi.sty` package, but they now apply to axodraw objects as well.

- The same effect, for named colors, can be achieved by `color.sty`'s `\textcolor` macro. Thus `\textcolor{Red}{...}` is equivalent to `\Red{...}`.

It is also possible to define new named colors, in the CMYK system. This means that each color is defined by four numbers. New colors can be introduced with the `\newcolor{#1}{#2}` command as in `\newcolor{LightRed}{0 0.75 0.7 0}`. This use of this command defines a named color for use in axodraw, with corresponding macros `\LightRed` and `\textLightRed{#1}`, and also makes the name known to `color.sty`. (Use of `color.sty`'s `\definecolor` macro is not supported here: it will affect only normal LaTeX text, but not axodraw objects, and it will fail to define the extra macros.)

We define the CMYK values for the named colors in the `axodraw2.sty` file. These override the definitions provided by `color.sty` (in its file dvipsnam.def), which are the same (at least currently).

There can be differences in how colors render on different devices. In principle, there should be compensations made by the driver to compensate for individual device properties. Our experience is however that such compensations are not always implemented well enough. Most notorious are differences between the shades of green on the screen, on projectors, and on output from a printer. These colors are usually much too light on a projector and one way to correct this is to redefine those colors when the output is prepared for a projector, e.g., by use of axodraw's `\newcolor{#1}{#2}` macro. An example is illustrated by



coded by

```
\color{green}
\begin{axopicture}(100,20)
```

```
  \Text(25,15){color.sty's green}
  \Line[width=2](0,0)(50,0)
  \end{axopicture}
%
  \color{Green}
  \begin{axopicture}(100,20)
  \Text(25,15){axodraw's Green}
  \Line[width=2](0,0)(50,0)
  \end{axopicture}
```

On a typical screen or projector, we find that the two greens are quite distinct, the "green" being much lighter than the "Green"[7]. But on the paper output from our printers, they give close results.

5.10.2 Defined named colors

The first set of predefined colors are those defined by dvips (and defined in `colordvi.sty`, or in `color.sty` with the use of both of its usenames and dvipsnames options). They are

> GreenYellow, Yellow, Goldenrod, Dandelion, Apricot, Peach, Melon, YellowOrange, Orange, BurntOrange, Bittersweet, RedOrange, Mahogany, Maroon, BrickRed, Red, OrangeRed, RubineRed, WildStrawberry, Salmon, CarnationPink, Magenta, VioletRed, Rhodamine, Mulberry, RedViolet, Fuchsia, Lavender, Thistle, Orchid, DarkOrchid, Purple, Plum, Violet, RoyalPurple, BlueViolet, Periwinkle, CadetBlue, CornflowerBlue, MidnightBlue, NavyBlue, RoyalBlue, Blue, Cerulean, Cyan, ProcessBlue, SkyBlue, Turquoise, TealBlue, Aquamarine, BlueGreen, Emerald, JungleGreen, SeaGreen, Green, ForestGreen, PineGreen, LimeGreen, YellowGreen, SpringGreen, OliveGreen, RawSienna, Sepia, Brown, Tan, Gray, Black, White.

In addition `axodraw2.sty` defines the following extra colors:

> LightYellow, LightRed, LightBlue, LightGray, VeryLightBlue.

Note that `color.sty` by default also defines a set of other named colors: black, white, red, green, blue, cyan, magenta, and yellow (with purely lower-case names). Depending on properties of your screen, projector or printer, these may or may not agree with the similarly named axodraw colors (which have capitalized names). These names can also be used in the `\SetColor` and `\color` commands and for color names to those axodraw commands that take named colors for arguments.

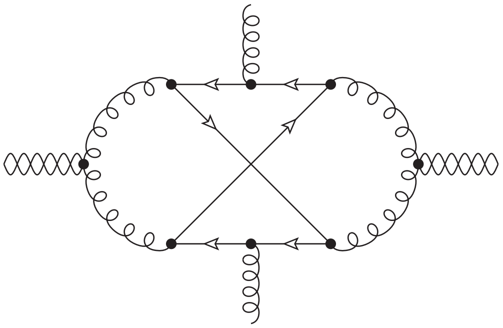
---

[7] The "green" is defined in the RGB scheme from the values $(0, 1, 0)$, while "Green" is defined in the CMYK scheme from the values $(1, 0, 1, 0)$.

6 Some examples

6.1 A Feynman diagram

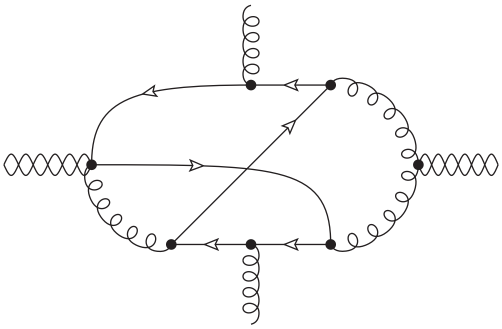
When computing the singlet part of structure functions in polarized Deep Inelastic Scattering one approach is to use spin two currents to determine all anomalous dimensions. At the three loop level this can give diagrams like the following:



for which the code is:

```
\begin{center} \begin{axopicture}{(200,140)(0,0)}
\SetArrowStroke{0.5} \SetArrowScale{0.8}
\Photon(7,70)(37,70){4}{3}
\Photon(7,70)(37,70){-4}{3}
\GluonArc(70,70)(30,90,270){3}{10}
\Line[arrow](100,100)(70,100) \Line[arrow](130,100)(100,100)
\Line[arrow,arrowpos=0.25](70,100)(130,40)
\Line[arrow](100,40)(70,40) \Line[arrow](130,40)(100,40)
\Line[arrow,arrowpos=0.75](70,40)(130,100)
\GluonArc(130,70)(30,270,450){3}{10}
\Photon(163,70)(193,70){4}{3}
\Photon(163,70)(193,70){-4}{3}
\Gluon(100,100)(100,130){3}{4}
\Gluon(100,40)(100,10){3}{4}
\Vertex(37,70){2} \Vertex(163,70){2} \Vertex(70,100){2}
\Vertex(70,40){2} \Vertex(130,100){2} \Vertex(130,40){2}
\Vertex(100,100){2} \Vertex(100,40){2}
\end{axopicture} \end{center}
```

The diagrams can become a bit more complicated when more lines meet in a single vertex. One could compose some lines from straight lines and arcs, but in this case we selected some Bézier curves. The result is



for which the code is:

```
\begin{center}
\begin{axopicture}{(200,140)(0,0)}
\SetArrowStroke{0.5} \SetArrowScale{0.8}
\Photon(7,70)(40,70){4}{3}
\Photon(7,70)(40,70){-4}{3}
\GluonArc(70,70)(30,180,270){3}{5}
\Bezier[arrow](100,100)(55,100)(40,95)(40,70)
\Line[arrow](130,100)(100,100)
\Bezier[arrow,arrowpos=0.37](40,70)(100,70)(130,70)(130,40)
\Line[arrow](100,40)(70,40) \Line[arrow](130,40)(100,40)
\Line[arrow,arrowpos=0.75](70,40)(130,100)
\GluonArc(130,70)(30,270,450){3}{10}
\Photon(163,70)(193,70){4}{3}
\Photon(163,70)(193,70){-4}{3}
\Gluon(100,100)(100,130){3}{4} \Gluon(100,40)(100,10){3}{4}
\Vertex(40,70){2} \Vertex(163,70){2} \Vertex(70,40){2}
\Vertex(130,100){2} \Vertex(130,40){2} \Vertex(100,100){2}
\Vertex(100,40){2}
\end{axopicture}
\end{center}
```

6.2 A diagrammatic equation

This example is from ref [6]. The equations in that paper were rather untransparent, because each Feynman diagram represents a complicated two loop integral and to solve these integrals one needed many different recursion relations in terms of the powers of the propagators. We defined a number of macro's for the diagrams, each containing one picture. Here are three of them:

```
\def\TAA(#1,#2,#3,#4,#5,#6){
  \raisebox{-19.1pt}{ \hspace{-12pt}
```

```
    \begin{axopicture}{(50,39)(0,-4)}
    \SetScale{0.5}\SetColor{Blue}%
    \CArc(40,35)(25,90,270) \CArc(60,35)(25,270,90)
    \Line(40,60)(60,60) \Line(40,10)(60,10) \Line(50,10)(50,60)
    \Line(0,35)(15,35) \Line(85,35)(100,35)
    \SetColor{Black}\SetPFont{Helvetica}{14}%
    \PText(55,39)(0)[lb]{#5} \PText(55,36)(0)[lt]{#6}
    \PText(35,62)(0)[rb]{#1} \PText(65,62)(0)[lb]{#2}
    \PText(65,8)(0)[lt]{#3} \PText(35,8)(0)[rt]{#4}
    \SetColor{Red} \SetWidth{3}
    \Line(50,35)(50,60) \Line(40,60)(50,60)
    \CArc(40,35)(25,90,180) \Vertex(50,60){1.3}
    \end{axopicture}
    \hspace{-12pt}
  }
}

\def\TABs(#1,#2,#3,#4,#5){
  \raisebox{-18.1pt}{ \hspace{-12pt}
    \begin{axopicture}{(50,39)(0,-4)}
    \SetScale{0.5}\SetColor{Blue}%
    \CArc(40,35)(25,90,270) \CArc(60,35)(25,270,90)
    \Line(40,60)(60,60) \Line(40,10)(60,10) \Line(50,10)(50,60)
    \Line(0,35)(15,35) \Line(85,35)(100,35)
    \SetColor{Black}\SetPFont{Helvetica}{14}%
    \PText(55,38)(0)[l]{#5}
    \PText(35,62)(0)[rb]{#1} \PText(65,62)(0)[lb]{#2}
    \PText(65,8)(0)[lt]{#3} \PText(35,8)(0)[rt]{#4}
    \SetColor{Red} \SetWidth{3}
    \Line(50,10)(50,60) \Vertex(50,60){1.3}
    \Line(40,60)(50,60) \CArc(40,35)(25,90,180)
    \end{axopicture}
    \hspace{-12pt}
  }
}

\def\TACs(#1,#2,#3,#4,#5){
  \raisebox{-19.1pt}{ \hspace{-12pt}
    \begin{axopicture}{(50,39)(0,-4)}
    \SetScale{0.5}\SetColor{Blue}%
    \CArc(40,35)(25,90,270) \CArc(60,35)(25,270,90)
    \Line(40,60)(60,60) \Line(40,10)(60,10) \Line(50,10)(50,60)
    \Line(0,35)(15,35) \Line(85,35)(100,35)
```

```
      \SetColor{Black}\SetPFont{Helvetica}{14}%
      \PText(53,38)(0)[l]{#5}
      \PText(35,62)(0)[rb]{#1} \PText(65,62)(0)[lb]{#2}
      \PText(65,8)(0)[lt]{#3} \PText(35,8)(0)[rt]{#4}
      \SetColor{Red} \SetWidth{3}
      \Line(40,60)(50,60) \CArc(40,35)(25,90,180)
      \end{axopicture}
      \hspace{-12pt}
    }
  }
```

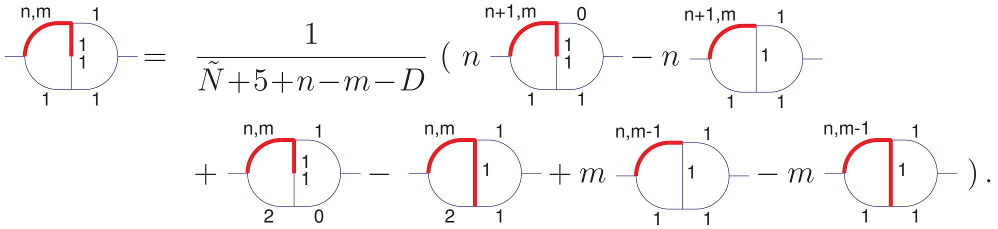
and together with two extra little macro's

```
\def\plus{\!+\!}
\def\minus{\!-\!}
```

the equations became rather transparent and easy to program. This is the code

```
   \begin{eqnarray}
     \TAA({n,m},1,1,1,1,1) & = & \frac{1}{\tilde{N}\plus 5\plus n\minus
     m\minus D}\ (\ n\ \ \TAA({n+1,m},0,1,1,1,1)
       \ \ -n\ \ \TACs({n+1,m},1,1,1,1)  \\ & &
       +\ \ \TAA({n,m},1,0,2,1,1)
       \ \ -\ \ \TABs({n,m},1,1,2,1)
       \ \ +m\ \ \TACs({n,m-1},1,1,1,1)
       \ \ -m\ \ \TABs({n,m-1},1,1,1,1)\ \ \ ) \, .\nonumber
   \end{eqnarray}
```

and the equation becomes



(5)

The diagrams are actually four-point diagrams. A momentum P flows through the diagram (the fat red line), but because the method of computation involves an expansion in terms of this momentum the remaining diagrams are like two-point functions. Details are in the paper.

Acknowledgements

JAMV's work is part of the research program of the "Stichting voor Fundamenteel Onderzoek der Materie (FOM)", which is financially supported by the "Nederlandse organisatie voor Wetenschappelijke Onderzoek (NWO)” and is also supported by the ERC Advanced Grant no. 320651, HEPGAME. JCC is supported in part by the U.S. Department of Energy under Grant No. DE-SC0008745.

We like to thank Lucas Theussl for discussions during the development of axodraw2.

A The `axohelp` program: Information for developers

This appendix provides some details on how the `axohelp` program works. Most of the information is only relevant to people who wish to modify or extend axodraw2 and therefore may need to modify `axohelp` as well.

The reason for `axohelp`'s existence is that axodraw needs to perform substantial geometric calculations. When axodraw is used with pdflatex to produce pdf output directly, suitable calculational facilities are not available, neither within the PDF language nor within LaTeX itself. Therefore when axodraw is used under pdflatex, we use our program `axohelp` to perform the calculations.

The mode of operation is as follows. Let us assume that the `.tex` file being compiled by the `pdflatex` program is called `paper.tex`. When one issues the command

```
pdflatex paper
```

the reaction of the system is of course to translate all TeX related objects into a PDF file. Most (but not all) axodraw objects need non-trivial calculations and hence their specifications are placed inside a file called `paper.ax1`. At the end of the processing `pdflatex` will place a message on the screen that mentions that the user should run the command

```
axohelp paper
```

for the processing of this graphical information. In principle it is possible to arrange for `axohelp` to be invoked automatically from within pdflatex. But for this to be done, the running of general external commands from pdflatex would have to be enabled. That is a security risk, and is therefore normally disabled by default for pdflatex.

When run, `axohelp` reads the file `paper.ax1`, processes the contents, and produces a file `paper.ax2`. For each axodraw object, it contains both the code to be placed in the pdf file, and a copy of the corresponding specification that was in `paper.ax1`.

When pdflatex is run again, it sees that the file `paper.ax2` is present and reads it in to give essentially an array of objects, one for each processed axodraw object. Then during the processing of the document, whenever axodraw runs into an axodraw object in need of external calculation, it determines whether an exactly corresponding specification was present in the file `paper.ax2`. If not, it means that the graphical information in the file `paper.tex` has changed since the last run of `axohelp` and the graphics information is invalidated. In that case, at the end of the program the message to run `axohelp` will be printed again. But if instead there is an exact match between an axodraw object in the current `paper.tex` and its specification in `paper.ax2`, then the corresponding pdf code will be placed in the PDF file. If all axodraw commands have a proper match in the `paper.ax2` file, there will be no message in the paper.log file and on the screen about rerunning `axohelp`; then the PDF file should contain the correct information for drawing the axodraw objects (at least if there are no TeX errors).

In a sense the situation with `axohelp` is no different from the use of makeindex when one prepares a document that contains an index. In that case one also has to run LaTeX once to prepare a file for the makeindex program, then run this program which prepares another file and finally run LaTeX again. Note that if you submit a paper to arXiv.org, it is likely that their automated system for processing the file will not run `axohelp`. So together with `paper.tex`, you one should also submit the `.ax2` file.

The complete source of the `axohelp` program can be found in the file `axohelp.c`. This file contains a bit less than 4000 lines of C code but should translate without problems with any C compiler — see Sec. 3.2.2 for an appropriate command line on typical Unix-like systems.

The `axohelp` program functions as follows:

1. The `.ax1` file is located, space is allocated for it and the complete file is read and closed again.
2. The input is analysed and split in individual object specifications, of which a list is made.
3. The list of object specifications is processed one by one. Before the processing of each object specification, the system is brought to a default state to avoid that there is a memory of the previous object.
4. In the `.ax2` file, for each object is written both the corresponding pdf code and a copy of the specification of the object as was earlier read from the `.ax1` file. Before the output for an object is written to the `.ax2` file it is optimized a bit to avoid superfluous spaces and linefeeds.

Processing an object from the input involves finding the proper routine for it and testing that the number of parameters is correct. Some objects have a special input (like the Curve, DashCurve, Polygon and FilledPolygon commands). All relevant information is stored in an array of double precision numbers. Then some generic action is taken (like setting the linewidth and the color) and the right routine is called. The output is written to an array of fixed (rather large) length. Finally the array is optimized and written to file.

A user who would like to extend the system with new objects should take the above structure into account. There is an array that gives the correspondence between axodraw object names and the corresponding routine in `axohelp`. For each object, this array also gives the number of parameters and whether the stroking or non-stroking color space should be used.

Naturally, when adding new kinds of object, it is necessary to add new items to the just-mentioned array, and to add a corresponding subroutine. One should also try to do all the writing of PDF code by means of some routines like the ones sitting in the file in the section named “PDF utilities”. This is important from the viewpoint of future action. When new graphical languages will be introduced and it will be needed to modify axodraw2 such that it can produce code for those languages, it should be much easier if code in the supporting `axohelp` program needs to be changed in as few places as possible. They form a set of graphics primitives used by other subroutines. Some of these subroutines in the “PDF utilities” section of `axohelp.c` have names similar to operators in the postscript language that perform the same function.

References

[1] J.A.M. Vermaseren, Comput. Phys. Commun. **83** (1994) 45–58

[2] D. Binosi and L. Theussl, Comput. Phys. Commun. **161** (2004) 76–86.

[3] D. Binosi, J. Collins, C. Kaufhold, L. Theussl, Comput. Phys. Commun. **180** (2009) 1709–1715

[4] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.

[5] J.C. Collins, “Foundations of Perturbative QCD” (Cambridge University Press, 2011).

[6] S. Moch and J.A.M. Vermaseren, Nucl. Phys. **B573** (2000) 853.