The LaTeX Graphics Companion

Second Edition

Michel Goossens
Frank Mittelbach
Sebastian Rahtz
Denis Roegel
Herbert Voß

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

```
The LaTeX Graphics companion / Michel Goossens ... [et al.]. -- 2nd ed.
       p. cm.
  Includes bibliographical references and index.
  ISBN 978-0-321-50892-8 (pbk. : alk. paper)
 1.  LaTeX (Computer file) 2.  Computerized typesetting. 3.  PostScript
(Computer program language) 4.  Scientific illustration--Computer programs.
5.  Mathematics printing--Computer programs. 6.  Technical
publishing--Computer programs.  I. Goossens, Michel.
  Z253.4.L38G663 2008
  686.2'2544536-dc22                                    2007010278
```


ISBN 10: 0-321-50892-0
ISBN 13: 978-0-321-50892-8

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, July 2007

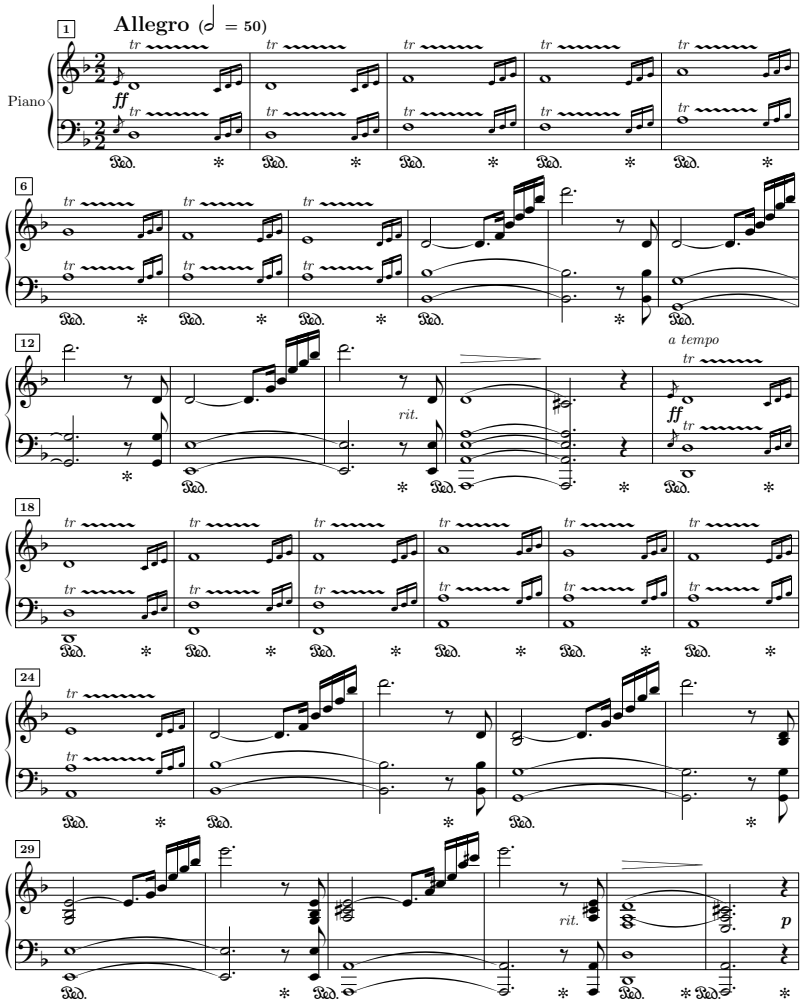
We dedicate this book to the hundreds of LaTeX developers
whose contributions are showcased in it,
and we salute their enthusiasm and hard work.

We would also like to remember with affection and thanks
Daniel Taupin, whose MusiXTeX system is described in
Chapter 9, and who passed away in 2003, a great loss to our community.

Rhapsodie

pour piano

Composé partiellement vers 1975, terminé en août 2002 — Daniel TAUPIN



Music composed by Daniel Taupin and typeset with MusiXTEX

CHAPTER 1

Graphics with LaTeX

The phrase "A picture paints a thousand words" seems to have entered the English language thanks to Frederick R. Barnard in *Printer's Ink*, 8 December 1921, retelling a Chinese proverb.[1] However, while LaTeX is quite good at typesetting words in a beautiful manner, LaTeX manuals usually tell you little or nothing about how to handle graphics. This book attempts to fill that gap by describing tools and TeXniques that let you generate, manipulate, and integrate graphics with your text.

In these days of the multimedia PC, graphics appear in various places. With many products we get ready-to-use collections of clipart graphics; in shops we can buy CD-ROMs with "the best photos" of important places; and so forth. As we shall see, all such graphics can be included in a LaTeX document as long as they are available in a suitable format. Fortunately, many popular graphic formats either are directly supported or can be converted via a program that allows transformation into a supported representation.

If you want to become your own graphic artist, you can use stand-alone dedicated drawing tools, such as the freely available dia (`www.gnome.org/projects/dia`) and xfig (`www.xfig.org/userman`) on Linux, or the commercial products Adobe Illustrator (`www.adobe.com/illustrator`) or Corel Draw (`www.corel.com/coreldraw`) on a Mac or PC. Spreadsheet programs, or one of the modern calculation tools like Mathematica

[1] Paul Martin Lester (`commfaculty.fullerton.edu/lester/writings/letters.html`) states that the literal translation of the "phony" Chinese proverb should rather be "A picture's meaning can express ten thousand words". He, rightly, emphasizes that pictures cannot and should not replace words, but both are complementary and contribute equally to the understanding of the meaning of a work.

(www.wolfram.com/mathematica), Maple (www.maplesoft.com/maple), and MATLAB (www.mathworks.com/matlab), or their freely available GNU variant Octave (www.octave.org) and its plotting complements Octaviz (octaviz.sourceforge.net) and Octplot (octplot.sourceforge.net), can also produce graphics by using one of their many graphical output representations. With the help of a scanner or a digital camera you can produce digital photos, images of hand-drawn pictures, or other graphics that can be manipulated with their accompanying software. In all these cases it is easy to generate files that can be directly referenced in the LaTeX source through the commands of the graphics package described in Chapter 2.

If needed, LaTeX can also offer a closer integration with the typesetting system than that possible by such programs. Such integration is necessary if you want to use the same fonts in text and graphics, or more generally if the "style" of the graphics should depend on the overall style of the document. Close integration of graphics with the surrounding text clearly requires generation of the graphic by the typesetting system itself, because otherwise any change in the document layout style requires extensive manual labor and the whole process becomes very error-prone.

* * *

This chapter considers graphic objects from different angles. First. we look at the requirements that various applications impose on graphic objects. Next, we analyze the types of drawings that appear in documents and the strategies typically employed to generate, integrate, and manipulate such graphics. Then, we discuss the interfaces offered by TeX for dealing with graphic objects. Armed with this knowledge, we end the chapter with a short survey of graphics languages built within and around TeX. This overview will help you select the right tool for the job at hand. In fact, the current chapter also gives some examples of languages and approaches not covered in detail elsewhere in the book. Thus this survey should provide you with enough information to decide whether or not to follow the pointers and obtain such a package for a particular application.

1.1 Graphics systems and typesetting

When speaking about "graphic objects", we should first define the term. One extreme position is to view everything put on paper as a graphic object, including the characters of the fonts used. This quite revolutionary view was, in fact, adopted in the design of the page description language PostScript, in which characters can be composed and manipulated by exactly the same functions as other graphic objects (we will see some examples of this in Chapters 5 and 6, which describe PSTricks and its support packages).

Most typesetting systems, including TeX, do not try to deploy such a general model but instead restrict their functional domain to a subset of general graphic objects—for example, by providing very sophisticated functions to place characters, resolve ligatures, etc., but omitting operators to produce arbitrary lines, construct and fill regions, and so forth. As a result the term "graphics" for most LaTeX users is a synonym for "artwork", thereby ignoring the fact that LaTeX already has a graphics language—the picture mode.

When discussing the graphical capabilities of an ideal typesetting system, we must remember that different applications have different, sometimes conflicting requirements:

- One extreme is the need for complete portability between platforms; another is to take into account even differences in the way printers put ink onto paper.
- A graphic might need to be correctly scaled to a certain size depending on factors of the visual environment created by the typesetting system, e.g., the measure of the text.
- It is also possible that parts of the graphic should not scale linearly. For example, it might be important for readability to ensure that textual parts of a graphic do not become smaller or larger than some limit. It might also be required that, when a graphic is scaled by, say, 10% to fit the line, any included text must stay the same, so as to avoid making it larger than the characters in the main document body.
- It might be required that the graphical object be closely integrated with the surrounding text, such as by using the same fonts as in other parts of the document or more generally by containing objects that should change their appearance if the overall style of the document is changed. (The latter is especially important if the document is described by its logical content rather than by its visual appearance, with the intention of reusing it in various contexts and forms.)

As LaTeX is a general-purpose typesetting system used for all types of applications, the preceding requirements and more might arise in various situations. As we will see throughout this book, a large number of them can be handled with grace, if not to perfection. In some cases an appropriate solution was anything but obvious and developing the mature macro packages and programs we now have took a decade or more of work.

1.2 Drawing types

The typology of graphics at the beginning of this chapter focused on the question of the integration with the LaTeX system, and divided the graphics into externally and internally generated ones. A different perspective would be to start from the types of graphics we might encounter in documents and discuss possible ways to generate and incorporate them.

A first class of graphics to be included are treated by LaTeX as a single object, a "black box", without an accessible inner structure. LaTeX, via its graphics package (described in Chapter 2), is interested only in the rectangular dimensions of the graphic image, its "bounding box". The graphics will be included in the output "as is", possibly after some simple manipulation, such as scaling or rotation. On top of that LaTeX can also produce a caption and legend to allow proper referencing from within the document. The main categories are as follows:

1. *Free-hand pictures* drawn without a computer, such as the drawing of a glass bead in Figure 1.1. For use in LaTeX, such a graphic must to be transformed into a digital image, using, for example, a scanner.

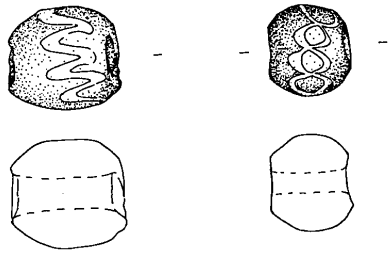


Figure 1.1: Pen and ink drawing of a bead

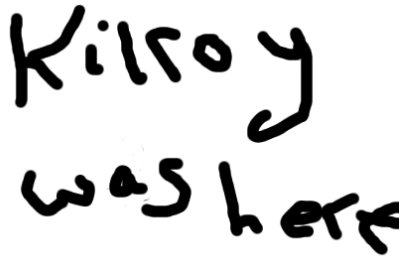


Figure 1.2: Bitmap drawing output created with GIMP

2. *"Art" graphics* drawn with bitmap tools on a computer, such as the example in Figure 1.2, which are to some extent the computer equivalents of pen and ink drawings. This drawing was created with GIMP, the GNU Image Manipulation Program (`www.gimp.org`), using a deliberately crude technique. The distinctive characteristic of this type of drawing is that the resolution chosen in the generation process cannot easily be changed without loss of quality (or alternatively without a lot of manual labor). In other respects such a picture is like a free-hand drawing: there is generally no desire to integrate the drawing with the text or to worry about conformity of typefaces.

3. *Photographs* either created directly using a digital camera or scanned like hand-drawn pictures. In the latter case the continuous tones of the photograph are converted into a distinct range of colors or gray levels (black-and-white photographs treated in this way are known as half-tones). Full-color reproduction requires sophisticated printing techniques, but this issue arises at the printing stage and does not normally affect the typesetting. Figure 1.3 shows how LaTeX can distort the image.

A second class of graphics is the "object-oriented" type, where the information is stored in the form of abstract objects that incorporate no device-dependent information (unlike bitmap graphics, where the storage format just contains information about whether a certain spot is black or white, making them resolution-dependent). This device independence makes it easy to reuse the graphic with different output devices and allows us to manipulate individual aspects of the graphic during the design process.

There are essentially three types of such graphics systems: one in which LaTeX mainly remains passive (it just takes into account the bounding box of the picture), and two others that relate to graphics that contain more complex text, in particular formulae. For the latter types it is important to use LaTeX to typeset text within the graphic because the symbols in formulae and their typeset form carry a precise semantic meaning. Therefore one must take great care to ensure that their visual representation is identical in both text and associated graphics.

1. *Self-contained object-oriented graphics.* The ducks of Figure 1.4, which was produced with Adobe Illustrator, were created by drawing one object in terms of curves and then

Figure 1.3: Digitally transformed image (vertically stretched)

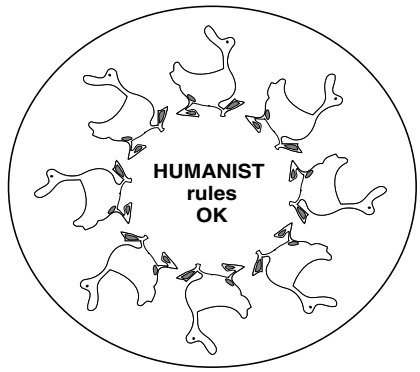


Figure 1.4: Object-oriented drawing

copying and rotating it many times. This type of drawing often also contains textual annotations comparable to typeset text. Although it is usually possible to add text to the graphic with external tools such Illustrator, it is not in general possible to use LaTeX to typeset this text (although psfrag provides a solution in some circumstances).

2. *Algorithmic display graphics* (e.g., histograms, graphs). These drawings are created without human interaction but often contain text that should match the document text. The scale and distance between elements is an essential characteristic of the drawing.
   Extensive plotting and diagram facilities are provided by many LaTeX packages building on the `picture` mode, by generic TeX packages such as PiCTeX [139], DraTex [39], and tikz [115]; and by PSTricks (see Chapters 5 and 6). All these solutions let us deploy the full power of LaTeX's typesetting functions within textual parts of the graphic and thus integrate it perfectly with surrounding document elements.

3. *Algorithmic structural graphics*, which can be derived from a textual representation. Unlike with the previous category, often merely the spatial relationship between elements is important with these graphics, not the elements' exact position or size. Examples are category diagrams, chemical formulae, trees, and flowcharts. Such graphics are natural candidates for generation by graphics languages internal to LaTeX that provide high-level interfaces which focus on objects and relationships and decide final placement and layout automatically.
   Of the general-purpose languages, the METAPOST system (Chapters 3 and 4) is perhaps the most flexible one for this type of graphics, although PiCTeX, Xy-pic (Chapter 7), PSTricks (Chapters 5 and 6), and DraTex are also suitable. They are based on different paradigms, and differ greatly in approach, focus, and user interface, but they all have found their place in the LaTeX world. We describe small specialized languages tailored for specific application domains such as physics, chemistry or electronics diagrams (Chapter 8), music (Chapter 9), and games (Chapter 10). For special applications such as tree drawing, many other LaTeX languages are available as well (see [13], for instance).

As we see, many types of graphics exist, each with its own requirements. The first three types essentially present themselves as black boxes to LaTeX and thus their use within a LaTeX document involves no more than their inclusion and in some cases their manipulation as a whole. The necessary functionality is discussed in detail in Chapter 2.

In scientific texts, the other types of graphics are by far the more common. Examples include maps [119], chemical structures, or commutative diagrams. They are for the most part based on an object-oriented approach, specifying objects and their relations in an abstract way using a suitable language. Close integration with the surrounding text can be achieved, if needed, by choosing one of the graphics languages described in this book.

In some cases interactive drawing programs can be instructed to output their results in one of the graphics languages built directly on top of LaTeX's `picture` mode. Widely used examples under Linux are dia and xfig, whose pictures, although externally produced, can be influenced by layout decisions within the document. Note, however, that such mechanically produced LaTeX code is normally not suitable for further manual editing and manipulation is practically limited to layout facilities implemented by the chosen graphics language. Nevertheless, in certain situations this approach can offer the best of two worlds.

1.3 TeX's interfaces

To understand the merits of the different approaches to graphics as implemented by various packages, it is helpful to consider yet another point of view: the interfaces provided by TeX for dealing with them. Describing the methods by which graphics can be generated, included, or manipulated will give you some feeling for such important issues as portability, quality, and resource requirements of individual solutions. We assume that the reader has a reasonable understanding of how TeX works—that is, the progression from source file to a DVI file that is processed by a driver to produce printed pages. Of course, the DVI stage can be skipped when using pdflatex, but the various ways of including the graphics material are still identical.

In the following we first look at ways of including externally generated graphics (i.e., those that appear as black boxes to TeX) and methods to manipulate them. Then we consider interfaces provided to build graphics languages within TeX.

1.3.1 Methods of integration

TeX offers two major facilities for integrating graphics as a whole: one involving the `\special` command, and the other using the font interface.

Using `\special` commands

The TeXbook [70] does not describe ways to directly include externally generated graphics. The only command available is the `\special` command, which by itself does nothing, but does enable us to access capabilities that might be present in the post-processor (DVI driver or pdflatex). To quote Knuth [70, page 229]:

> The `\special` command enables you to make use of special equipment that might be available to you, e.g., for printing books in glorious TeXnicolor.

CHAPTER 2

Standard LaTeX Interfaces

Since the introduction of LaTeX 2ε in 1994, LaTeX has offered a uniform syntax for including every kind of graphics file that can be handled by the different drivers. In addition, all kinds of graphic operations (such as resizing and rotating) as well as color support are available.

These features are not part of the LaTeX 2ε kernel, but rather are loaded by the standard, fully supported color, graphics, and graphicx extension packages. Because the TeX program does not have any direct methods for graphic manipulation, the packages must rely on features supplied by the "driver" used to print the `dvi` file. Unfortunately, not all drivers support the same features, and even the internal method of accessing these extensions varies among drivers. Consequently, all of these packages take options, such as `dvips`, to specify which external driver is being used. Through this method, unavoidable device-dependent information is localized in a single place, the preamble of the document.

In this chapter we start by looking at graphics file inclusion. LaTeX offers both a simple interface (graphics), which can be combined with the separate rotation and scaling commands, and a more complex interface (graphicx), which features a powerful set of manipulation options. The chapter concludes with a discussion of the pict2e package, which implements the driver encapsulation concept for line graphics and with a brief description of the curve2e package, which is not part of the "standard LaTeX interface" but nevertheless represents an interesting extension to pict2e. Color support is covered in Chapter 11.

2.1 Inclusion of graphics files

The packages graphics and graphicx can both be used to scale, rotate, and reflect LaTeX material or to include graphics files prepared with other programs. The difference between

Table 2.1: Overview of color and graphics capabilities of device drivers

Option	*Author of Driver*	*Features*
`dvips`	T. Rokicki	All functions (reference driver; option also used by xdvi)
`dvipdf`	S. Lesenko	All functions
`dvipdfm`	S. Lesenko	All functions
`dvipsone`	Y&Y	All functions
`dviwin`	H. Sendoukas	File inclusion
`emtex`	E. Mattes	File inclusion only, but no scaling
`pdftex`	Hàn Thế Thành	All functions for pdftex program
`pctexps`	PCTeX	File inclusion, color, rotation
`pctexwin`	PCTeX	File inclusion, color, rotation
`pctex32`	PCTeX	All functions
`pctexhp`	PCTeX	File inclusion only
`truetex`	Kinch	Graphics inclusion and some color
`tcidvi`	Kinch	TrueTeX with extra support for Scientific Word
`textures`	Blue Sky	All functions for Textures program
`vtex`	Micropress	All functions for VTeX program

the two is that graphics uses a combination of macros with a "standard" or TeX-like syntax, while the "extended" or "enhanced" graphicx package presents a key/value interface for specifying optional parameters to the `\includegraphics` and `\rotatebox` commands.

2.1.1 Options for graphics and graphicx

When using LaTeX's graphics packages, the necessary space for the typeset material after performing a file inclusion or applying some geometric transformation is reserved on the output page. It is, however, the task of the *device driver* (e.g., dvips, xdvi, dvipsone) to perform the actual inclusion or transformation in question and to show the correct result. Given that different drivers may require different code to carry out an action, such as rotation, one has to specify the target driver as an option to the graphics packages—for example, option `dvips` if you use one of the graphics packages with Tom Rokicki's dvips program, or option `textures` if you use one of the graphics packages and work on a Macintosh using Blue Sky's Textures program.

Some drivers, such as previewers, are incapable of performing certain functions. Hence they may display the typeset material so that it overlaps with the surrounding text. Table 2.1 gives an overview of the more important drivers currently supported and their possible limitations. Support for older driver programs exists usually as well—you can search for it on CTAN.

The driver-specific code is stored in files with the extension `.def`—for example, `dvips.def` for the PostScript driver dvips. As most of these files are maintained by third parties, the standard LaTeX distribution contains only a subset of the available files and not necessarily the latest versions. While there is usually no problem if LaTeX is installed as part of a full TeX installation, you should watch out for incompatibilities if you update the LaTeX graphics packages manually.

Setting a default driver

It is also possible to specify a default driver using the `\ExecuteOptions` declaration in the *configuration* file `graphics.cfg`. For example, `\ExecuteOptions{dvips}` makes the dvips drivers become the default. In this case the graphics packages pick up the driver code for the dvips TEX system on a PC if the package is called without a driver option. Most current TEX installations are distributed with a ready-to-use `graphics.cfg` file.

In addition to the driver options, the packages support some options controlling which features are enabled (or disabled):

`draft` Suppress all "special" features, such as including external graphics files in the final output. The layout of the page will not be affected, because LATEX still reads the size information concerning the bounding box of the external material. This option is of particular interest when a document is under development and you do not want to download the (often huge) graphics files each time you print the typeset result. When `draft` mode is activated, the picture is replaced by a box of the correct size containing the name of the external file.

`final` The opposite of `draft`. This option can be useful when, for instance, "draft" mode was specified as a global option with the `\documentclass` command (e.g., for showing overfull boxes), but you do not want to suppress the graphics as well.

`hiresbb` In PostScript files, look for bounding box comments that are of the form `%%HiResBoundingBox` (which typically have real values) instead of the standard `%%BoundingBox` (which should have integer values).

`hiderotate` Do not show the rotated material (for instance, when the previewer cannot rotate material and produces error messages).

`hidescale` Do not show the scaled material (for instance, when the previewer does not support scaling).

With the graphicx package, the options `draft`, `final`, and `hiresbb` are also available locally for individual `\includegraphics` commands, that is, they can be selected for individual graphics.

2.1.2 The `\includegraphics` syntax in the graphics package

With the graphics package, you can include an image file by using the following command:

```
\includegraphics*[llx,lly][urx,ury]{file}
```

If the `[urx,ury]` argument is present, it specifies the coordinates of the upper-right corner of the image as a pair of TEX dimensions. The default units are big (PostScript) points; thus `[1in,1in]` and `[72,72]` are equivalent. If only one optional argument is given, the lower-left corner of the image is assumed to be located at `[0,0]`. Otherwise, `[llx,lly]` specifies the coordinates of that point. Without optional arguments, the size of the graphic is determined by reading the external *file* (containing the graphics itself or a description thereof, as discussed later).

```
%!PS-Adobe-2.0
%%BoundingBox:100 100 150 150
100 100     translate % put origin at 100 100
  0   0     moveto    % define current point
 50  50     rlineto   % trace diagonal line
 50 neg 0   rlineto   % trace horizontal line
 50  50 neg rlineto   % trace other diagonal line
stroke                % draw (stroke) the lines
  0   0     moveto    % redefine current point
/Times-Roman findfont % get Times-Roman font
 50         scalefont % scale it to 50 big points
            setfont   % make it the current font
(W) show              % draw an uppercase W
```

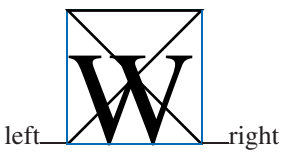
Figure 2.1: The contents of the file `w.eps`

The starred form of the `\includegraphics` command "clips" the graphics image to the size of the specified bounding box. In the normal form (without the `*`), any part of the graphics image that falls outside the specified bounding box overprints the surrounding text.

The examples in the current and next sections use a small PostScript program (in a file `w.eps`) that paints a large uppercase letter "W" and a few lines. Its source is shown in Figure 2.1. Note the `BoundingBox` declaration, which stipulates that the image starts at the point `100, 100` (in big points), and goes up to `150, 150`; that is, its natural size is 50 big points by 50 big points.

In the examples we always embed the `\includegraphics` command in an `\fbox` (with a blue frame and zero `\fboxsep`) to show the space that LaTeX reserves for the included image. In addition, the baseline is indicated by the horizontal rules produced by the `\HR` command, defined as an abbreviation for `\rule{1em}{0.4pt}`.

The first example shows the inclusion of the `w.eps` graphic at its natural size. Here the picture and its bounding box coincide nicely.

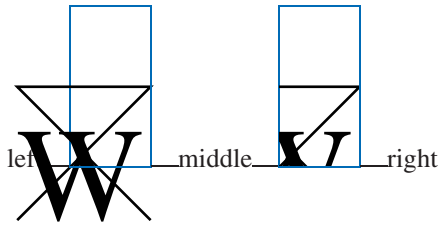



```
\usepackage{graphics,color}
\newcommand\HR{\rule{1em}{0.4pt}}
\newcommand\bluefbox[1]{\textcolor{blue}{%
   \setlength\fboxsep{0pt}\fbox{\textcolor{black}{#1}}}}
left\HR \bluefbox{\includegraphics{w.eps}}\HR right
```

Example 2-1-1

Next, we specify a box that corresponds to a part of the picture (and an area outside it) so that some parts fall outside its boundaries, overlaying the material surrounding the picture. If the starred form of the command is used, then the picture is clipped to the box (specified as optional arguments), as shown on the right.

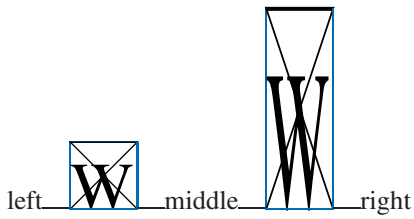
Example 2-1-2




```
\usepackage{graphics,color}
% \bluefbox and \HR as before
left\HR
 \bluefbox{\includegraphics
         [120,120][150,180]{w.eps}}%
\HR middle\HR
 \bluefbox{\includegraphics*
         [120,120][150,180]{w.eps}}%
\HR right
```

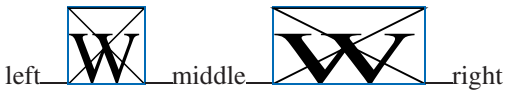
In the remaining examples we combine the `\includegraphics` command with other commands of the graphics package to show various methods of manipulating an included image. (Their exact syntax is discussed in detail in Section 2.2.) We start with the `\scalebox` and `\resizebox` commands. In both cases we can either specify a change in one dimension and have the other scale proportionally, or specify both dimensions to distort the image.

Example 2-1-3




```
\usepackage{graphics,color}
% \bluefbox and \HR as before
left\HR
  \bluefbox{\scalebox{.5}{%
    \includegraphics{w.eps}}}%
\HR middle\HR
  \bluefbox{\scalebox{.5}[1.5]{%
    \includegraphics{w.eps}}}%
\HR right
```

Example 2-1-4




```
\usepackage{graphics,color}
% \bluefbox and \HR as before
left\HR
  \bluefbox{\resizebox{10mm}{!}{%
    \includegraphics{w.eps}}}%
\HR middle\HR
  \bluefbox{\resizebox{20mm}{10mm}{%
    \includegraphics{w.eps}}}%
\HR right
```

Adding rotations makes things even more interesting. Note that in comparison to Example 2-1-1 on the facing page the space reserved by LaTeX is far bigger. LaTeX "thinks" in rectangular boxes, so it selects the smallest size that can hold the rotated image.

Example 2-1-5


```
\usepackage{graphics,color}
% \bluefbox and \HR as before
left\HR
  \bluefbox{\rotatebox{25}{%
    \includegraphics{w.eps}}}%
\HR right
```

2.1.3 The \includegraphics syntax in the graphicx package

The extended graphics package graphicx also implements \includegraphics but offers a syntax for including external graphics files that is somewhat more transparent and user-friendly. With today's TeX implementations, the resultant processing overhead is negligible, so we suggest using this interface.

\includegraphics*[*key/val-list*]{*file*}

The starred form of this command exists only for compatibility with the standard version of \includegraphics, as described in Section 2.1.2. It is equivalent to specifying the clip key.

The *key/val-list* is a comma-separated list of *key*=*value* pairs for keys that take a value. For Boolean keys, specifying just the key is equivalent to *key*=true; not specifying the key is equivalent to *key*=false. Possible keys are listed below:

bb The bounding box of the graphics image. Its value field must contain four dimensions, separated by spaces. This specification will overwrite the bounding box information that might be present in the external file.[1]

hiresbb Makes LaTeX search for %%HiResBoundingBox comments, which specify the bounding box information with decimal precision, as used by some applications. In contrast, the normal %%BoundingBox comment can take only integer values. It is a Boolean value, either "true" or "false".

viewport Defines the area of the graphic for which LaTeX reserves space. Material outside this will still be print unless trim is used. The key takes four dimension arguments (like bb), but the origin is with respect to the bounding box specified in the file or with the bb keyword. For example, to describe a 20 bp square 10 bp to the right and 15 bp above the lower-left corner of the picture you would specify viewport=10 15 30 35.

trim Same functionality as the viewport key, but this time the four dimensions correspond to the amount of space to be trimmed (cut off) at the left-hand side, bottom, right-hand side, and top of the included graphics.

natheight,natwidth The natural height and width of the figure, respectively.[2]

angle The rotation angle (in degrees, counterclockwise).

origin The origin for the rotation, similar to the origin parameter of the \rotatebox command described on page 40.

width The required width (the width of the image is scaled to that value).

[1] There also exists an obsolete form kept for backward compatibility only: [bbllx=a, bblly=b, bburx=c, bbury=d] is equivalent to [bb = a b c d], so the latter form should be used.

[2] These arguments can be used for setting the lower-left coordinate to (0 0) and the upper-right coordinate to (natwidth natheight) and are thus equivalent to bb=0 0 w h, where w and h are the values specified for these two parameters.

CHAPTER 3

METAFONT and METAPOST: TEX's Mates

In designing the TEX typesetting system, Donald Knuth soon realized that he would also have to write his own font design program. He devised METAFONT, a language for describing shapes, and a program to interpret that language and turn the shapes into a pattern of dots for a printing or viewing device. The result of Knuth's work was TEX, METAFONT, and the extensive Computer Modern font family written in METAFONT. METAFONT has also been used to create special-purpose symbol fonts and some other font families.

The development of METAFONT as a font description language paralleled to some extent that of the PostScript language, which also describes character shapes very elegantly. PostScript's strategy, however, is to leave the rendering of the shape until the final printing stage, whereas METAFONT seeks to precompute the bitmap output and print it on a fairly dumb printing device.

Font design is a decidedly specialist art, and one that most of us are ill equipped to tackle. METAFONT, however, defines a very powerful language that can cope with most graphical tasks. A sibling program, METAPOST, was developed that uses essentially the same language but generates PostScript instead of bitmaps. Together, the two provide an

excellent companion facility with which (LA)TEX users can illustrate their documents, particularly when they want pictures that graphically express some mathematical construct; this is not surprising, given that Knuth's aim was to describe font shapes mathematically. Applications vary from drawing Hilbert or Sierpiński curves (described in Section 4.4.3) to plotting data in graphs and expressing relationships in graphical form.

In this chapter we consider how to use both METAFONT and METAPOST (henceforth we use META to mean "both METAFONT and METAPOST") to draw pictures and shapes other than characters in fonts.

Our coverage of META is divided into six parts. We start with a brief look at the META language basics; our aim is to give readers new to META some ideas of its facilities and the level at which pictures can be designed. We try to explain commands as they are used, but some examples may contain META code that is not explicitly described.

We next consider in some detail the extra facilities of the METAPOST language, in particular the inclusion of text and color in figures.

The third section examines how the META programs are run and how resulting figures can be included in a LATEX document. The following section describes the general-purpose METAPOST libraries, covering in particular boxing macros and the METAOBJ package.

We then look at programs that write META commands for you, concentrating on the mfpic (LA)TEX package. We conclude with an overview of miscellaneous tools and utilities related to METAPOST.

For some applications, such as drawing of graphs, diagrams, geometrical figures, and 3-D objects, higher-level macro packages have been developed, which define their own languages for the user. These packages are described in Chapter 4.

3.1 The META language

The full intricacies of METAFONT are described in loving detail in [72]; the manual for METAPOST [47] not only describes the differences between the two systems, but is itself a good introduction to META. Alan Hoenig's book *TEX Unbound* [49] provides a wealth of material on METAFONT techniques. Articles over many years in the journal *TUGboat* are also vital reading for those who want to delve deeply into METAFONT and METAPOST.

The job of the META language is to describe shapes; these shapes can then be filled, scaled, rotated, reflected, skewed, and shifted, among other complex transformations. Indeed, META programs can be regarded as specialized equation-solving systems that have the side effect of producing pictures.

META offers all the facilities of a conventional programming language. Program flow control, for example, is provided by a `for ... endfor` construct, with the usual conditionals. You can write parameterized macros or subroutines, and there are facilities for local variables and grouping to limit the scope of value changes. Some of these features are described with more detail in the METAPOST section, although they are also available in METAFONT.

Because a lot of the work in writing META programs deals with describing geometrical shapes, the numeric support is extensive. For instance, Pythagorean addition (`++`) and subtraction (`+-+`) are directly supported. Useful numeric functions include `length` x

(absolute value of x), `sqrt` x (square root of x), `sind` x (sine of x degrees), `cosd` x (cosine of x degrees), `angle` (x, y) (arctangent of y/x), `floor` x (largest integer $\leq x$), `uniformdeviate` x (uniformly distributed random number between 0 and x), and `normaldeviate` (normally distributed random number with mean 0 and standard deviation 1).

A variety of complex data types are defined, including `boolean`, `numeric`, `pair`, `path`, `pen`, `picture`, `string`, and `transform`. Here we can look at some of these in more detail:

pair "Points" in two-dimensional space are represented in METAFONT with the type `pair`. Constants of type `pair` have the form (x, y), where x and y are both `numeric` constants. A variable p of type `pair` is equal to the `pair` expression (`xpart` p, `ypart` p).

path A `path` is a continuous curve, which is composed of a chain of *segments*. Each segment has a shape determined by four *control points*. Two of the control points, the *key* points, are the segment's end points; very often we let METAFONT determine the other two control points.

pen Pens, a distinctive feature of METAFONT, are filled convex shapes that are moved along `path`s and affect the way lines are drawn in the result. Two pens are initially present in METAFONT: `nullpen` and `pencircle`. `nullpen` is the single point $(0, 0)$; it contains no pixels and can be used to fill a region without changing its boundary. By contrast, `pencircle` is circular, with the points $(\pm 0.5, 0)$ and $(0, \pm 0.5)$ on its circumference. Other pens are constructed as convex polygons via `makepen` c, where c is a closed `path`; the key points of c become the vertices of the pen. Pens themselves can be transformed.

picture A `picture` is a data type that can be used to store a sequence of METAFONT drawing commands; the result of a complete METAFONT program is often built up from the interaction of a set of pictures. The meaning of $v + w$ in METAFONT, for example, is a picture in which each pixel is the sum of the two pixels occupying the same position in pictures v and w, respectively.

transform Affine transforms are the natural transformations of Euclidean geometry—that is, the linear transformations augmented by translation. METAFONT can construct any affine transform and provides seven primitive ones [72, p. 141]: *shifted*, *scaled*, *xscaled*, *yscaled*, *slanted*, *rotated*, and *zscaled*. The effect of most of the operations is self-evident; the last one, *zscaled*, uses a pair of numbers, interpreted as a complex number in Cartesian coordinates (i.e., complex multiplication).

Finally, METAFONT is famous for its ability to solve linear equations, including equations that involve points. In particular, you can define a point in terms of other points. For example, `z3=1/2[z1,z2]` defines `z3` as the point in the middle of the line from `z1` to `z2`.

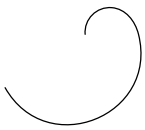
3.1.1 First examples of METAFONT programs

Let us first look at some examples of METAFONT code, all drawn using METAPOST. You should have little difficulty making these examples run under METAFONT as well, except that

you may encounter problems with high-resolution output devices, as METAFONT can run out of memory when composing large pictures—remember that METAFONT generates a bitmap output. This book was typeset at 2400 dpi, and some METAFONT examples were impossible to run at this resolution. Your only recourse is to work at a lower resolution (e.g., 300 dpi) or to break your picture into separate "characters" in a font and join them together in LaTeX. It is almost certainly easier to use METAPOST, as it generates PostScript that can be rendered directly by many printers or turned into PDF.

We do not show the "wrapper" code that is always necessary to turn these examples into a self-contained document. See the notes in Section 3.3.1 on page 68 for information on how METAFONT creates a character and Section 3.3.2 on page 71 for more on how METAPOST creates a figure.

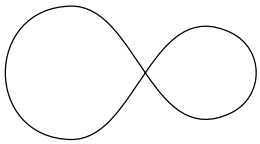
The simplest statement in META is `draw`, which takes a sequence of points separated by `..` and connects them with curves:



```
draw (0,0)..(50,20)..(40,30)..(30,20);
```

Example 3-1-1

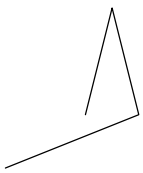
The default unit here is a PostScript point (1/72 inch, TEX's "big point"). To close a object smoothly between its last and first points, the sequence can be terminated by `cycle`:



```
draw (0, 50)..(0,0)..
  (60,40)..(60,10)..cycle;
```

Example 3-1-2

Straight lines are drawn by putting `--` instead of `..` between the points (the lines are actually implemented as specially constrained curves):



```
draw (0,0)--(50,20)--(40,60)--(30,20);
```

Example 3-1-3

There are several ways of controlling curves: one can vary the angles at the start and end of the curve with `dir`, the points that are to be the extremes (the upmost, the leftmost, and so forth), and the inflection of the curve (with `tension` and `curl`). Thus the following

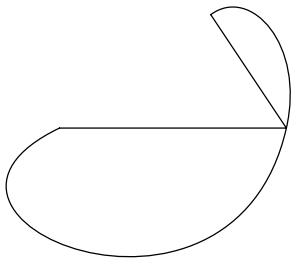
code draws a crude coil by judicious use of `dir`. Instead of the default units, we express all dimensions in terms of a unit of 2.5 cm, defined at the start:

Example 3-1-4

```
u=2.5cm;
path p;
p= (0,0) {dir 130}..
   {dir -130}(0.25u,0){dir 130}..
   {dir -130}(0.5u,0){dir 130}..
   {dir -130}(0.75u,0){dir 130}..
   {dir -130}(u,0);
draw p rotated -90;
```

The next example shows the effect of `curl`. Here a straight line is drawn between three points and then a curve is drawn between the same points, with `curl` values:

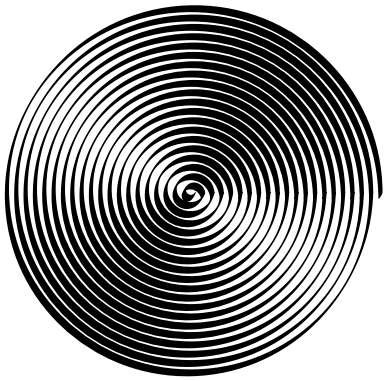
Example 3-1-5



```
path p,q;
u=.5cm;
q=(0u,0u)--(6u,0u)--(4u,3u);
draw q;
p=(0u,0u){curl 4000}..(6u,0u)
  ..{curl 4000}(4u,3u);
draw p;
```

To demonstrate METAFONT's unusual "pens", we approximate a spiral drawn with a strange "nib". A colored version of this drawing appears in Color Plate I(a).

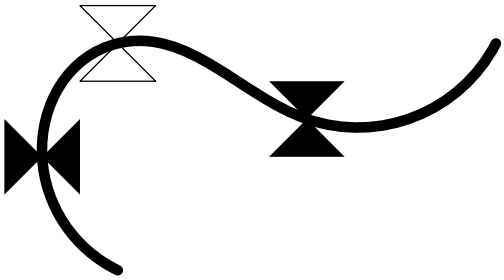
Example 3-1-6



```
pickup pencircle scaled 3pt
   yscaled .2pt rotated 60;
n:=5;
for i := (n*20) step -(n) until (n):
 draw ((i,0)..(0,i)..(-i,0)
      ..(0,-(i-n))..(i-n,0)) scaled 0.7;
endfor
```

A very characteristic technique with METAFONT is creating a path and then using it several times with different transformations. The following code is an extract from a drawing of a

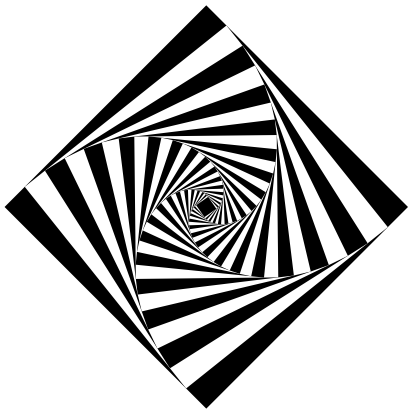
kite's tail. Note that shapes can be made solid by using `fill` instead of `draw`:



```
u=1cm;
path p[];
p1:=(.5u,.5u)--(1.5u,.5u)--(.5u,1.5u)
 --(1.5u,1.5u)--(.5u,.5u)--cycle;
fill (p1 shifted (0,2.5u))
 rotatedaround ((u,3.5u),90);
draw p1 shifted (u,4u);
fill p1 shifted (3.5u,3u);
p2 =(2u,2u)..(u,3.5u)..(2u,5u)
 ..(4.5u,4u)..(7u,5u);
pickup pencircle scaled 4pt;
draw p2;
```

Example 3-1-7

A more complicated picture, courtesy of Alan Hoenig from his book *TEX Unbound* [49], demonstrates looping commands. Boxes of gradually decreasing size are drawn alternately white and black, with each one being rotated slightly with respect to the previous box.



```
boolean timetofillbox; timetofillbox := true;
partway := 0.9; l := .45in; u := 1.05in;
n := 4; theta := 360/n; z1 = (0,u);
for i := 2 upto n:
  z[i] = z1 rotated ((i-1)*theta);
endfor
forever:
  path p;    p := z1
  for j := 2 upto n: --z[j] endfor  --cycle;
  if timetofillbox:
    fill p;  timetofillbox := false;
  else:
    unfill p;  timetofillbox := true;
  fi
  pair Z[];
  for j := 1 upto n:
    Z[j] := partway[z[j-1],z[j]];
  endfor
  Z1 := partway[z[n],z1];
  for j := 1 upto n:
    x[j] := xpart Z[j]; y[j] := ypart Z[j];
  endfor
  if not timetofillbox: l := abs(z1); fi
  exitif l < .05u;
endfor
```

Example 3-1-8

CHAPTER 4

METAPOST Applications

Chapter 3 gave a general overview of METAFONT and METAPOST, as well as an extensive description of two multipurpose structuring packages, boxes and METAOBJ. However, as is the case for LaTeX, solutions to many problems can often be found by using existing high-level packages. Sometimes several different METAPOST packages are aimed at the same tasks, and these packages come with both advantages and drawbacks.

Unfortunately, the perfect package is seldom at hand. It is therefore useful to have a general idea of what can be achieved in METAPOST, and to have some kind of toolbox for problem solving. Understanding a number of basic tricks will enable the beginner to supplement existing packages and achieve the desired results.

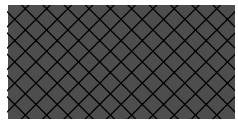
In this chapter, we start with a review of a number of basic problems and show how these problems can be solved. Then we describe some standard applications of METAPOST, ranging from geometry to physics.

4.1 A drawing toolkit

This section is devoted to a number of advanced features, which are located somewhere between low-level METAPOST code and full application packages. We like to consider all these features as a kind of toolkit, which can be used with benefit in wider applications.

Bogusław Jackowski's hatching package provides a more elaborate way to achieve hatching patterns, by redefining the `withcolor` primitive in such a way that it represents hatching parameters when the blue component of the color is negative. The following examples illustrate this principle.

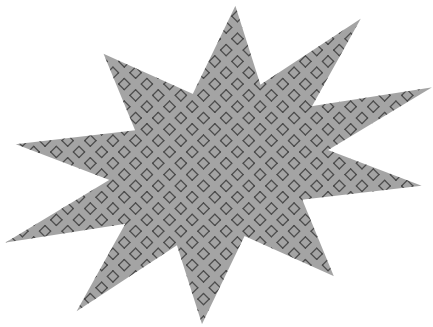
Example 4-1-10



```
input hatching;
path p;
p:=unitsquare xscaled 30mm yscaled 15mm;
hatchfill p withcolor red
           withcolor (45,2mm,-.5bp)
           withcolor (-45,2mm,-.5bp);
```

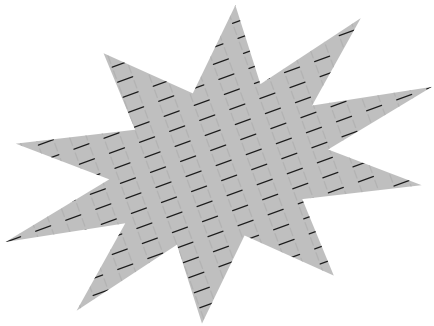
The next three examples use a special closed path shaped as a star, defined by the `star` macro:

Example 4-1-11



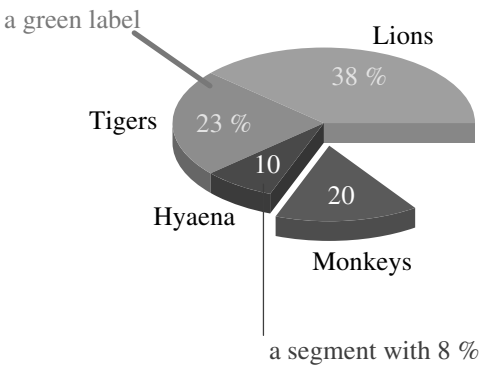
```
input hatching;
vardef star(expr n) =
 for i_:=0 upto 2n-1:
  if odd i_: 1/2 fi (right rotated (180*(i_/n))) --
 endfor cycle
enddef;
interim hatch_match:=0;
path p;
p:=star(10) xscaled 30mm
            yscaled 20mm
            rotated 20;
hatchfill p withcolor (0,1,.5);
draw image(hatchfill p
           withcolor (45,3bp,-.5bp)
           withcolor (-45,3bp,-.5bp);
     ) withcolor red dashed evenly;
```

Example 4-1-12



```
input hatching;
% star macro defined as above
path p;
p:=star(10) xscaled 30mm
            yscaled 20mm
            rotated 20;
interim hatch_match:=0;
hatchoptions(withcolor blue
             dashed evenly scaled 2);
hatchfill p withcolor .75white
           withcolor (20,6bp,-.5bp);
hatchoptions(withcolor (blue+green)
             dashed evenly
             shifted (3/2bp,0));
hatchfill p withcolor (110,6bp,-.5bp);
```

A more elaborate example appears below. The 8% corresponds to 10 being 8% of $50 + 30 + 10 + 20 + 20$.




```
input piechartmp
SetupColors((.7,.7),this,this);
SetupPercent(this, " %");
Segment(50,"Lions");  Segment(30,"Tigers");
Segment(10,"Hyaena"); Segment(20,"Monkeys");
Segment(20,"Warthogs");
SegmentState(4,this,0.3);
SegmentState(5,invisible,this);
PieChart(2cm,0.15,60,0,0);
Label.auto(0)(name)(outwards,0);
Label(3,4,5)(value)(inwards,0) withcolor white;
Label(1,2)(percent)(inwards,0) withcolor (1,1,0);
Label.lrt(3)("a segment with ",percent)
            ((0.9,0.8),(0,-2cm)) withcolor .8red;
pickup pencircle scaled 2pt;
Label.auto(2)("a green label")
        ((0.9,0.1),(-1cm,7mm)) withcolor .8green;
```

Example 4-2-26

This example has labels with spaces and needs a font with spaces—hence the `defaultfont` declaration. This is not a problem when we are using TEX labels.

`SetupNumbers`(*precision*,*delimiter*)

Setup commands

In addition to the `SetupPercent` commands, several other setup commands are available. The first, `SetupNumbers`, sets the accuracy and delimiter used. `SetupNumbers(2,",")` will, for instance, round at two places and use a comma delimiter.

`SetupColors`(*auto-SV*,*shading-SV*,*grayscale*)

This command specifies the colors used for segments. The three arguments are as follows:

auto-SV is a pair (S, V), where S is the saturation and V is the value in the HSV model. The hue H is taken from the position of the segment.

shading-SV is a pair giving the maximum values of (S, V) for shaded areas in segments. The default is (0.4,0.3).

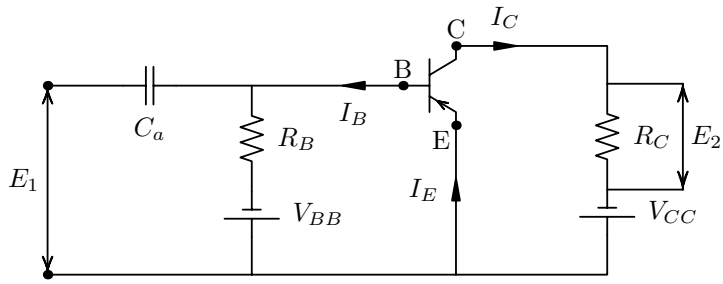
grayscale is a Boolean that, when set to `true`, switches the colors to grayscale.

`SetupText`(*Mode*,*TeXFormat*,*TeXSettings*)

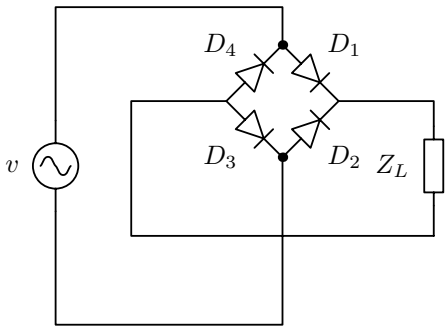
This command sets up how text is handled, using three arguments:

Mode is an integer specifying the way labels are typeset: 0 is for string-based typesetting (default); 1 is for external TEX-based typesetting using *TeXFormat* and *TeXSettings*; 2 is

```
ctext.rt(R.C.l+(1cm,0),R.C.r+(1cm,0),"$E_2$",witharrow);
```



Example 4-5-7

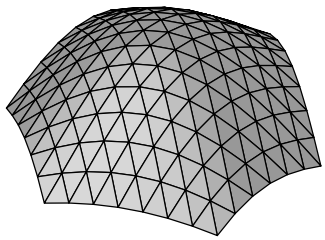



```
input makecirc;
initlatex("\usepackage{amsmath,amssymb}");
source.a(origin,AC,90,"v","");
junction.a(S.a.p+(3cm,1cm),"")(top);
diode.a(J.a,normal,-45,pinA,"D_1","");
diode.b(D.a.K,normal,-135,pinK,"D_2","");
diode.c(D.b.A,normal,135,pinK,"D_3","");
diode.d(D.c.A,normal,45,pinA,"D_4","");
junction.b(D.b.A,"")(bot);
centerto.A(S.a.n,S.a.p)(5cm,imp);
impedance.a(A,90,"Z_L","");
wireU(S.a.p,D.a.A,1.5cm,udsq);
wireU(S.a.n,D.b.A,-1.5cm,udsq);
wire(D.a.K,Z.a.r,rlsq);
wire(Z.a.l,Z.a.l+(0,-4mm),nsq);
wireU(Z.a.l+(0,-4mm),D.d.A,-4cm,rlsq);
```

Example 4-5-8

```
input makecirc;
initlatex("\usepackage{amsmath,amssymb}");
transformer.a(origin,mid,0);
diode.a(tf.a.ss+(5mm,1cm),normal,0,pinA,"D_1","");
diode.b(tf.a.si+(5mm,-1cm),normal,0,pinA,"D_2","");
impedance.a(D.a.K+(2cm,-4mm),-90,"Z_L","300\ohm");
wire(tf.a.ss,D.a.A,udsq);wire(tf.a.si,D.b.A,udsq);
wire(D.a.K,Z.a.l,rlsq);wire(Z.a.r,tf.a.m,udsq);
wire(D.b.K,D.a.K+(5mm,0),rlsq);
junction.a(D.a.K+(5mm,0),"")(top);
centerto.A(tf.a.pi,tf.a.ps)(-15mm,sac);
source.a(A,AC,90,"220 V","v");
wire(S.a.p,tf.a.ps,udsq);wire(S.a.n,tf.a.pi,udsq);
centreof.A((xpart S.a.p,ypart tf.a.ps),tf.a.ps,cur);
current.a(c.A,phi.A,"i(t)","5 A");
imesh(tf.a.ss+(1cm,0),15mm,1cm,cw,0,"I_{cc}");
```

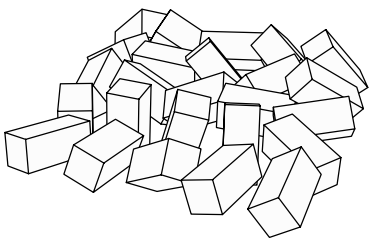
hexagonal meshes Given a function $z = f(x, y)$, a hexagonal mesh can be obtained with the `hexagonaltrimesh` macro.



```
input featpost3Dplus2D
def zsurface( expr xc, yc ) =
        cosd(xc*57)*cosd(yc*57)
        +4*mexp(-(xc**2+yc**2)*6.4)
enddef;

f := 7*(4,1,5);
Spread := 35;
LightSource := 10*(4,-3,4);
SubColor := 0.4background;

numeric np, ssize;
path chair;
np = 20;
ssize = 5;

hexagonaltrimesh( true,np,ssize,zsurface);
```

Example 4-6-2

cubes The `kindof cube` macro produces a cube in an orientation depending on its parameters. In this example, each cube erases what has been drawn under it, so that it gives the illusion of the removal of hidden parts.

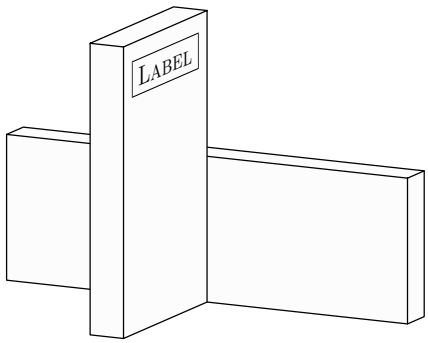


```
input featpost3Dplus2D
Spread := 30;
f := 5.4*(1.5,0.5,1);
numeric gridstep, sidenumber,
        i, j, coord, aa, ab, ac;
color pa;
gridstep = 0.7;
sidenumber = 4;
coord = 0.5*sidenumber*gridstep;
for i=0 upto sidenumber:
  for j=0 upto sidenumber:
    pa := (-coord+j*gridstep,-coord+i*gridstep,0);
    aa := uniformdeviate(360);
    ab := uniformdeviate(180);
    ac := uniformdeviate(90);
    kindofcube(false, false,
               pa, aa, ab, ac, 0.4, 0.4, 0.9 );
  endfor;
endfor;
```

Example 4-6-3

labels in space The next example shows how labels can be drawn in space using the `labelinspace` macro.

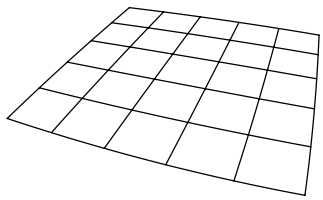
Example 4-6-4




```
input featpost3Dplus2D
verbatimtex
%&latex
\documentclass{article}
\begin{document}
etex

f := 1.1*(2,1,0.5);
ParallelProj := true;
kindofcube(false,true,(0,-0.5,0),
           90,0,0,1.2,0.1,0.4);
kindofcube(false,true,(0,0,0),
           0,0,0,0.5,0.1,0.8);
labelinspace(false,(0.45,0.1,0.65),
             (-0.4,0,0),(0,0,0.1),
    btex \framebox{\textsc{Label}} etex);
```

projected segments The last example shows how points can be defined in space, and `pathofstraightline` used to draw a segment joining the projections of these points.

Example 4-6-5



```
input featpost3Dplus2D
SphericalDistortion := true;
Spread := 50;
f := 0.4*(1.5,0.5,1);
numeric gridstep, sidenumber, i, coord;
color pa, pb, pc, pd;
gridstep = 0.1;
sidenumber = 5;
coord = 0.5*sidenumber*gridstep;
for i=0 upto sidenumber:
  pa := (-coord,-coord+i*gridstep,0);
  pb := (coord,-coord+i*gridstep,0);
  pc := (-coord+i*gridstep,-coord,0);
  pd := (-coord+i*gridstep,coord,0);
  draw pathofstraightline( pa, pb );
  draw pathofstraightline( pc, pd );
endfor;
```

3DLDF

Laurence D. Finston's ambitious extension to METAPOST, 3DLDF (`http://www.gnu.org/software/3dldf/LDF.html`) is written in C++ using CWEB. 3DLDF (the author's initials) takes an input similar to METAPOST and outputs pure METAPOST code. The package currently computes the intersections of various projected curves, and the author plans to implement the removal of hidden parts.

CHAPTER 5

Harnessing PostScript Inside LaTeX: PSTricks

As we saw in Chapter 1, one way of drawing graphics with LaTeX is to embed low-level picture drawing primitives for the target device into LaTeX macros, so that full typesetting information is available and we can work in a familiar macro programming environment. When the target device is something as rich as the full PostScript language, this can result in a very powerful system. While many macro packages have implemented access to some parts of PostScript for this purpose, the most complete is undoubtedly **PSTricks**. In the next two chapters, we survey its capabilities and demonstrate some of the power that results from combining LaTeX and PostScript.

We do not attempt to describe absolutely every PSTricks-related macro, nor do we give examples of all the possible combinations and tricks, as this would require a large book of its own, e.g., [135]. We have, however, tried to describe and give examples of all the important features of the basic packages. You'll find a lot of useful information on the official PSTricks Web site at `http://PSTricks.tug.org/`.

Because there are a great many commands and especially keywords in PSTricks, we provide a summary description at the end of the next chapter (Section 6.8 on page 459). PSTricks and its related packages are extremely powerful, and their facilities may take some time to understand. It is also documented in the individual packages and [127, 135], and its implementation is described in [126].

5.1 The components of PSTricks

The PSTricks project was started by Timothy Van Zandt a long time ago and is one of the oldest TeX packages still in use.

> I started in 1991. Initially I was just trying to develop tools for my own use. Then I thought it would be nice to package them so that others could use them. It soon became tempting to add lots of features, not just the ones I needed. When this became so interesting that it interfered with my "day job", I gave up the project "cold turkey", in 1994.
>
> [Timothy Van Zandt]

After Timothy Van Zandt stopped working on the project, Denis Girou took over the task to care for PSTricks, mainly fixing bugs and writing some more new packages; nowadays this job is done by Herbert Voß. Several developers are working on existing and new packages, which is the reason why the number of these additional packages, which depend on the basic PSTricks, is still increasing. A selection of them is discussed in Chapter 6, and the full list is available at the official Web site at `http://PSTricks.tug.org`.

5.1.1 The kernel

The basic PSTricks package file is `pstricks.tex`, which provides the basic unit handling, and basic graphic macros like dots, lines, frames, and so on. For some historical reason the packages pstricks, pst-plot, pst-node, and pst-tree build the core of PSTricks and are all available on CTAN in the directory `CTAN:/graphics/pstricks/base/generic/`. Each PSTricks package has a corresponding LaTeX style file, and the basic ones are stored in `CTAN:/graphics/pstricks/base/latex/`. In general, the style files do nothing other than load the TeX file via the `\input` macro.

The basic PSTricks packages consist of a core of picture-drawing primitives implemented by `\special` commands that pass PostScript code to a driver, mainly dvips. The packages also contain a set of higher-level macros for particular applications, like pst-plot or pst-node. With it you can

- Draw lines, polygons, circles, and curves.
- Place and manipulate TeX text.

\psgrid [*settings*] (x_0, y_0) (x_1, y_1) (x_2, y_2)

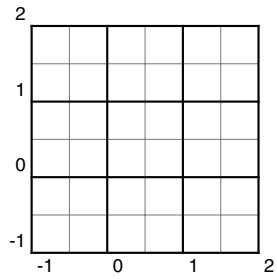
The \psgrid macro is a very powerful tool for drawing coordinate grids. The syntax is easy to use, but is valid only for Cartesian coordinate systems.

When no coordinates have been specified, \psgrid takes the ones defined by the enclosing pspicture environment or, if not inside such an environment, a 10×10 rectangle in the current units is assumed. If only one coordinate pair is given, it is taken to denote one corner and (0,0) is established as the opposite corner. When using two coordinate pairs, any two opposite corners of the grid should be specified. With three coordinate pairs given, the first pair determines the intersection point of the lines to be labeled and the other two pairs are interpreted as in the previous case.

In short: (x_0, y_0) defaults to (x_1, y_1); the default for the latter is (0,0), and (outside of a pspicture environment) the default for (x_2, y_2) is (10,10).

The labels are positioned along the two lines that intersect at (x_0, y_0), on the side of the line pointing away from (x_2, y_2), and shifted slightly horizontally or vertically towards the latter coordinate so they won't interfere with other lines. In the next example, \psgrid has no arguments, so it takes all coordinates from the surrounding pspicture environment. The keywords used in this and the following examples are discussed in detail in Section 5.5.1 on the following page.

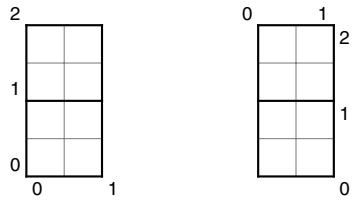
Example 5-5-1




```
\usepackage{pstricks}
\psset{griddots=0,gridlabels=7pt,subgriddiv=2}
\begin{pspicture}(-1,-1)(2,2)
  \psgrid
\end{pspicture}
```

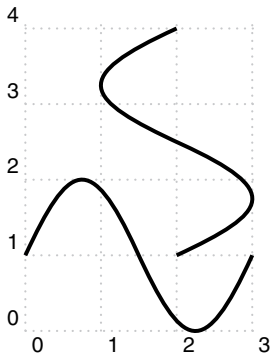
With only one pair of coordinates, \psgrid assumes that (0,0) is the opposite corner. Exchanging the order of the coordinate pairs, as in the second figure, changes the position of the labels from the left and bottom sides to the right and top sides of the rectangle, respectively. (See also the last example below with three pairs of coordinates.)

Example 5-5-2




```
\usepackage{pstricks}
\begin{pspicture}(-1,-1)(2,2)
  \psgrid[griddots=0,gridlabels=7pt,subgriddiv=2](1,2)
\end{pspicture}
\begin{pspicture}(-1,-1)(2,2)
  \psgrid[griddots=0,gridlabels=7pt,
          subgriddiv=2](1,2)(0,0)
\end{pspicture}
```

This is also demonstrated in the next example.



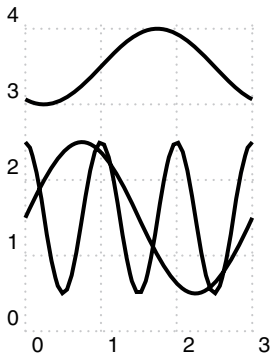

```
\usepackage{pstricks,pst-plot}
\begin{pspicture}[showgrid=true](3,4)
  \pscustom[linewidth=1.5pt]{%
    \translate(0,1)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \translate(2,0)
    \swapaxes
    \psplot[liftpen=2]{0}{3}{x 180.0 mul 1.5 div sin}}
\end{pspicture}
```

Example 5-13-18

\msave \mrestore

With this pair of macros, the currently valid coordinate system may be saved and restored, respectively. In contrast to what happens with \gsave and \grestore pairs, all other values such as line type, thickness, etc., will remain unaffected. The \msave and \mrestore commands must be used in pairs! They can be nested arbitrarily both with themselves and with \gsave and \grestore. Care must be taken to ensure that this nesting is pairwise balanced.

The next example plots the first sine function with the origin of ordinates set by \translate(0,1.5). Thereafter, the state of the coordinate system is saved, a new origin is set with \translate(1,2)[1], and another sine function is plotted. Following that, the old state is restored with \mrestore and the origin of ordinates is back at (0,1.5) again. The later cosine function is plotted with this origin.




```
\usepackage{pstricks,pst-plot}
\begin{pspicture}[showgrid=true](3,4)
  \pscustom[linewidth=1.5pt]{%
    \translate(0,1.5)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \msave
      \translate(1,2)
      \scale{1 0.5}
      \psplot[liftpen=2]{-1}{2}{x 180.0 mul 1.5 div sin}
    \mrestore
    \psplot[liftpen=2]{0}{3}{x 180.0 mul 0.5 div cos}}
\end{pspicture}
```

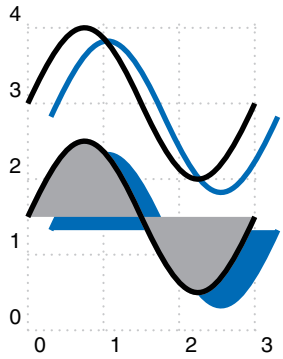
Example 5-13-19

[1] Referring to the current origin (0,1.5) a \translate(1,2) corresponds to the absolute coordinates (1,3.5).

`\openshadow` [*settings*]

The `\openshadow` command creates a copy of the current path, using the specified shadow key values (see page 239). Whether the shadow path thus obtained is stroked or filled depends on the parameter settings supplied with `\openshadow` itself and/or `\pscustom`, as can be seen in the example.

Example 5-13-20

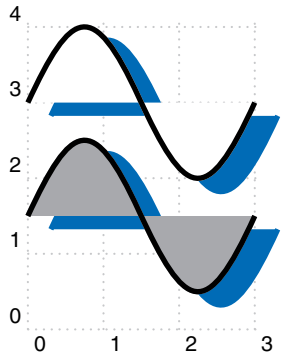



```
\usepackage{pstricks,pst-plot}
\begin{pspicture}[showgrid=true](3,4)
  \pscustom[linewidth=2pt]{%
    \translate(0,3)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \openshadow[shadowsize=10pt,shadowangle=-30,
                shadowcolor=blue]}
  \pscustom[linewidth=2pt,fillcolor=red,
            fillstyle=solid]{%
    \translate(0,1.5)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \openshadow[shadowsize=10pt,shadowangle=-30,
                shadowcolor=blue]}
\end{pspicture}
```

`\closedshadow` [*settings*]

The `\closedshadow` command *always* creates a filled shadow of the region enclosed by the current path, as if it were a non-transparent environment.

Example 5-13-21

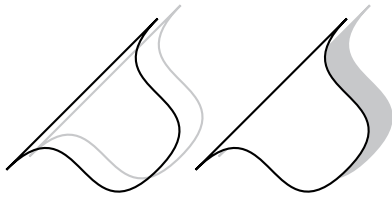



```
\usepackage{pstricks,pst-plot}
\begin{pspicture}[showgrid=true](3,4)
  \pscustom[linewidth=2pt]{%
    \translate(0,3)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \closedshadow[shadowsize=10pt,shadowangle=-30,
                  shadowcolor=blue]}
  \pscustom[linewidth=2pt,fillcolor=red,
            fillstyle=none]{% <-- no effect!
    \translate(0,1.5)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \closedshadow[shadowsize=10pt,shadowangle=-30,
                  shadowcolor=blue]}
\end{pspicture}
```

The method used for producing the shadow should be noted. PSTricks simply creates a copy of the closed path, translates it according to the demands of `shadowsize` and `shadowangle`, fills it with `shadowcolor`, and then refills the original path with `fillcolor`, which is `white` by default. The `\openshadow` macro doesn't fill the original

path with the current `fillcolor`, so that the underlying shadow copy is visible (and in this example, not filled). The `\closedshadow` fills the original path, so that the underlying copy looks like a real shadow.

```
\usepackage{pstricks}
\begin{pspicture}(0,-0.25)(5,2)
\pscustom[fillstyle=none,shadowcolor=lightgray,fillcolor=blue]{%
  \psbezier(0,0)(1,1)(1,-1)(2,0) \psbezier(2,0)(3,1)(1,1)(2,2)
  \closepath
  \openshadow[shadowsize=10pt,fillcolor=white,shadowangle=30]}
\rput(2.5,0){%
\pscustom[fillstyle=none,shadowcolor=lightgray,fillcolor=blue]{%
  \psbezier(0,0)(1,1)(1,-1)(2,0) \psbezier(2,0)(3,1)(1,1)(2,2)
  \closepath
  \closedshadow[shadowsize=10pt,fillcolor=white,shadowangle=30]}}
\end{pspicture}
```

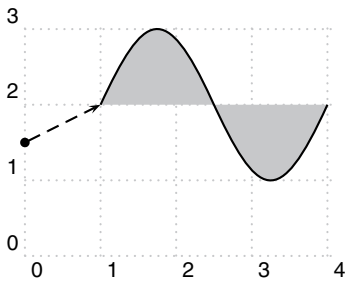


Example 5-13-22

This strategy is to be kept in mind when specifying, with the keyword `\pscustom`, a `fillcolor` that differs from `white`: in such cases the macro `\closedshadow` has to be given the correct fill color.

`\movepath(`dx, dy`)`

The `\movepath` command shifts the current path by (dx, dy). If the original path is needed later on, the `\movepath` operation has to be encapsulated within a `\gsave`/`\grestore` pair.




```
\usepackage{pstricks,pst-plot}
\begin{pspicture}[showgrid=true](4,3)
  \pscustom[fillcolor=lightgray,fillstyle=solid]{%
    \translate(0,1.5)
    \psplot{0}{3}{x 180.0 mul 1.5 div sin}
    \movepath(1,0.5)}
  \psline[linestyle=dashed]{*->}(0,1.5)(1,2)
\end{pspicture}
```

Example 5-13-23

CHAPTER 6

The Main PSTricks Packages

The "main" packages of PSTricks nowadays have this name only for historical reasons. PSTricks is used for those packages listed in the pst-all package. We do not follow this list here. Instead, we describe the most common ones (e.g., pst-plot, pst-node) in some detail. Section 6.7 then gives an overview of other packages, showing at least one characteristic example to help you understand the purpose of each package and approach that it takes.

6.1 pst-plot—Plotting functions and data

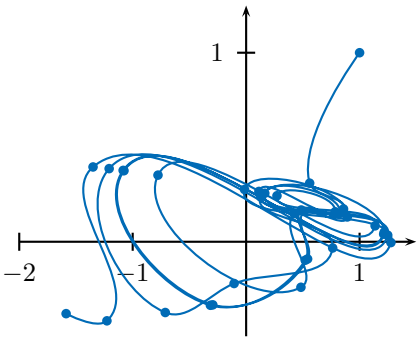
The base package pstricks provides some macros to plot function values and coordinates, as listed in Table 6.1. All of these macros accept an arbitrary number of coordinate pairs as arguments.

The pst-plot package provides improved commands for plotting external data and functions as well as coordinate axes [59, 60, 131]. It supports only two-dimensional data pairs. For plotting (x, y, z) data triplets or three-dimensional functions, you can use the pst-3dplot package discussed in Section 6.6, which supports a parallel projection of 3-D objects [132, 134].

\listplot

In contrast to the preceding plot commands, the argument of \listplot is first expanded if it contains TEX macros; otherwise, it is passed to PostScript without change. In the process, TEX macros are replaced with their corresponding replacement text. It is possible to include entire PostScript programs in the argument to \listplot, as shown in Example 6-1-33.

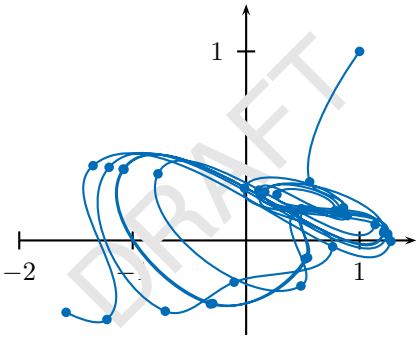
The first example illustrates the Hénon attractor.




```
\usepackage{pstricks,pst-plot}
% definition of \henon with data points like this:
% \newcommand\henon{ 1.00000000  1.00000000
%                    0.56000000  0.31000000
%                    ... many more ...}

\psset{xunit=1.5cm, yunit=2.5cm}
\begin{pspicture}(-2,-0.5)(1.5,1.25)
  \psaxes{->}(0,0)(-2,-0.5)(1.5,1.25)
  \listplot[showpoints=true,plotstyle=curve,
    linecolor=blue]{\henon}
\end{pspicture}
```

Example 6-1-32

The second example includes the watermark "DRAFT", which was added to the original data with additional PostScript code.



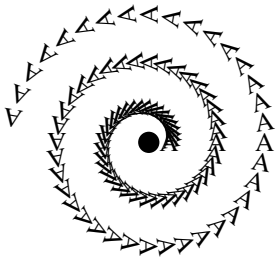

```
\usepackage{pstricks,pst-plot}
% \henon as in previous example
\newcommand{\dataA}{\henon
  gsave
  /Helvetica findfont 40 scalefont setfont
  45 rotate
  0.9 setgray
  -60 10 moveto (DRAFT) show
  grestore }
\psset{xunit=1.5cm, yunit=2.5cm}
\begin{pspicture}(-2,-0.5)(1.5,1.25)
  \psaxes{->}(0,0)(-2,-0.5)(1.5,1.25)
  \listplot[showpoints=true,linecolor=blue,
        plotstyle=curve]{\dataA}
\end{pspicture}
```

Example 6-1-33

Instead of modifying the data set passed to \listplot, you can redefine the \ScalePoints macro in pst-plot. For example, if you wanted to exchange the x and y val-

It works only in conjunction with the \nput command (see page 359).

Example 6-2-54



```
\usepackage{pstricks,pst-node,multido}
\begin{pspicture}(4.5,4.5)
  \cnode*(2,2){4pt}{A}
  \multido{\nA=0+10,\rB=0+0.5}{90}{%
     \nput[rot=\nA,%
        labelsep=\rB pt]{\nA}{A}{A}}
\end{pspicture}
```

6.2.5 Putting labels on node connections

In Section 5.11 on page 265, we already discussed several commands that allow arbitrary placement of marks with respect to labels. In the context of connections, there are some special commands to consider. After a connection has been drawn, the coordinates of two points are stored temporarily until a new connection is drawn. This data may prove very useful for positioning the labels to be attached to such a connection. Of course, it also implies that label commands should come immediately after connection commands.

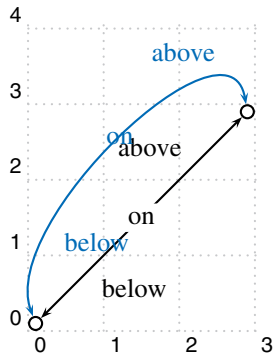
In Section 6.2.4 on page 348, which discussed the allowed keywords, you will find many examples of the placement of labels. In this section we will review the various commands once again.

\ncput* [*settings*] {*object*} \naput* [*settings*] {*object*} \nbput* [*settings*] {*object*}

n labels

The n label commands are always based on the visible length of a connection, without attention to the actual node centers. By default, the label is placed in the middle of this visible connection, which can be changed with the appropriate keyword. The letter c indicates *connected* (on the line), and a and b indicate *above* and *below* the line, respectively. The starred versions produce opaque material, which means you can overwrite lines with a label to gain increased visibility.

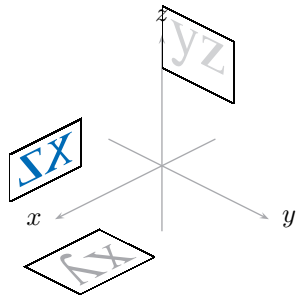
Example 6-2-55




```
\usepackage{pstricks,pst-node}
\begin{pspicture}[showgrid=true](3,4)
  \cnode(0.1,0.1){0.1cm}{A}  \cnode(2.9,2.9){0.1cm}{B}
  \ncline{<->}{A}{B} \ncput*{on}
  \naput[npos=0.75]{above} \nbput[npos=0.25]{below}
  \nccurve[angleA=110,angleB=100,
    linecolor=blue]{<->}{A}{B}
  \ncput{\textcolor{blue}{on}}
  \naput[npos=0.75]{\textcolor{blue}{above}}
  \nbput[npos=0.25]{\textcolor{blue}{below}}
\end{pspicture}
```

The `pOrigin` *key*

The keyword `pOrigin` is the positioning key, which is passed to the command `\rput`. Its effects concern only `\pstThreeDPut`, and the default value is based on the defaults for `\rput` (see Section 5.11.1 on page 266).




```
\usepackage{pstricks,pst-3dplot}
\begin{pspicture}(-2,-1)(1,2.5)
  \pstThreeDCoor[xMin=-1,xMax=2,yMin=-1,
                 yMax=2,zMin=-1,zMax=2]
  \pstPlanePut[pOrigin=c](0,0,-1){\fbox{\Huge\red xy}}
  \pstPlanePut[plane=xz,pOrigin=rb](0,0,0)
              {\fbox{\Huge\blue xz}}
  \pstPlanePut[plane=yz,pOrigin=lb](0,0,1.5)
              {\fbox{\Huge\green yz}}
\end{pspicture}
```

Example 6-6-28

The `hiddenLine` *key*

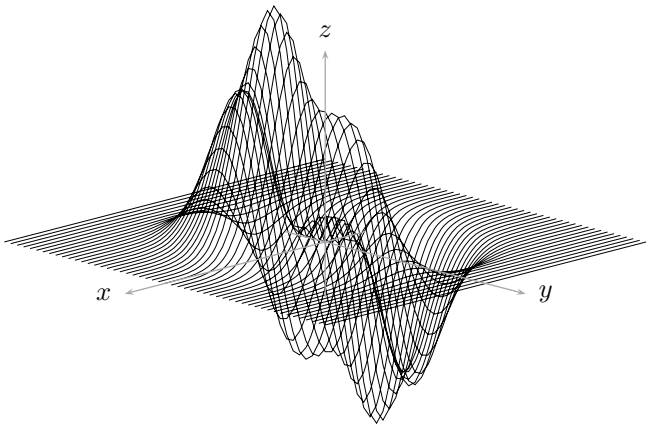
The keyword `hiddenLine` enables a very simple "hidden-line algorithm": the lines are plotted with the command `\pscustom` and then filled with the predefined fill style `hiddenStyle`.

```
\newpsstyle{hiddenStyle}{fillstyle=solid,fillcolor=white}
```

You can overwrite this style as required. Just keep in mind that the curves must be built from the end to the beginning; otherwise, the hidden lines will be visible. For examples, see Section 6.6.2 on page 406.

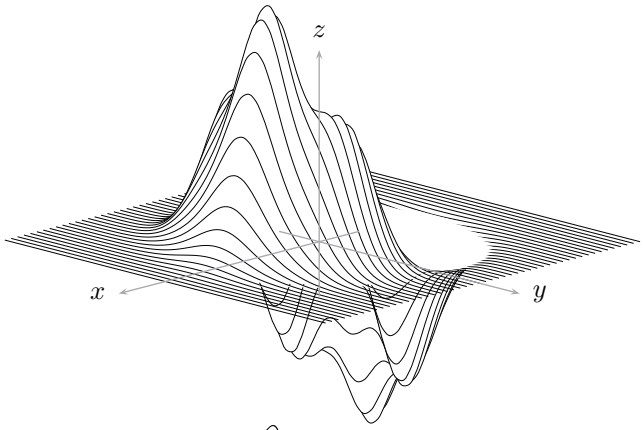
The `drawStyle` *key*

The keyword `drawStyle` defines the manner in which the function is plotted. Possible key values are `xLines`, `yLines`, `xyLines`, and `yxLines`. The values refer to the plotting sequence; that is, `xLines` has the lines drawn in the x direction, whereas `yxLines` means that they are first drawn in the y direction and then in the x direction.



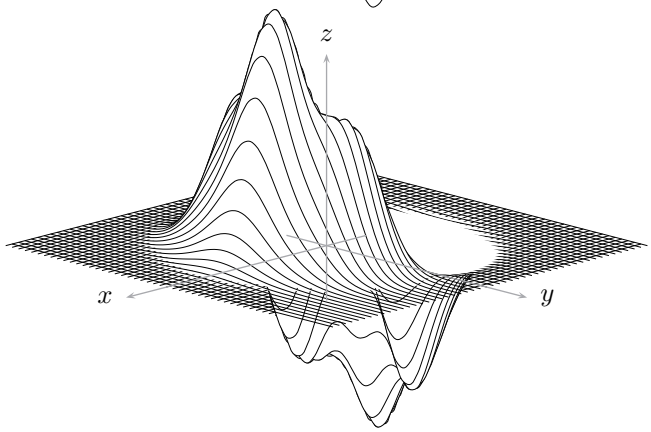

```
\usepackage{pstricks,pst-3dplot}
% \func as defined in Example 6-6-13
\begin{pspicture}(-6,-3)(6,4)
  \psset{Beta=15,unit=0.75}
  \psplotThreeD[plotstyle=line,
     drawStyle=xLines,
     yPlotpoints=50,xPlotpoints=50,
     linewidth=0.2pt](-4,4)(-4,4)
    {\func}
  \pstThreeDCoor[xMax=5,yMax=5,
                 zMax=3.5]
\end{pspicture}
```

Example 6-6-29



Example 6-6-30

```
\usepackage{pstricks,pst-3dplot}
% \func as defined in Example 6-6-13
\begin{pspicture}(-6,-3)(6,4)
  \psset{Beta=15,unit=0.75}
  \psplotThreeD[plotstyle=curve,%
     drawStyle=yLines,%
     hiddenLine=true,%
     yPlotpoints=50,xPlotpoints=50,%
     linewidth=0.2pt](-4,4)(-4,4){\func}
  \pstThreeDCoor[xMax=5,yMax=5,zMax=3.5]
\end{pspicture}
```

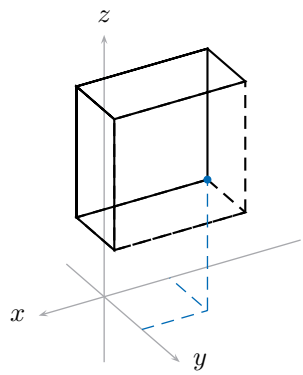


Example 6-6-31

```
\usepackage{pstricks,pst-3dplot}
% \func as defined in Example 6-6-13
\begin{pspicture}(-6,-3)(6,4)
  \psset{Beta=15,unit=0.75}
  \psplotThreeD[%
    plotstyle=curve,drawStyle=xyLines,%
    hiddenLine=true,%
    yPlotpoints=50,xPlotpoints=50,%
    linewidth=0.2pt](-4,4)(-4,4){\func}
  \pstThreeDCoor[xMax=5,yMax=5,zMax=3.5]
\end{pspicture}
```

The `visibleLineStyle` *and* `invisibleLineStyle` *keys*

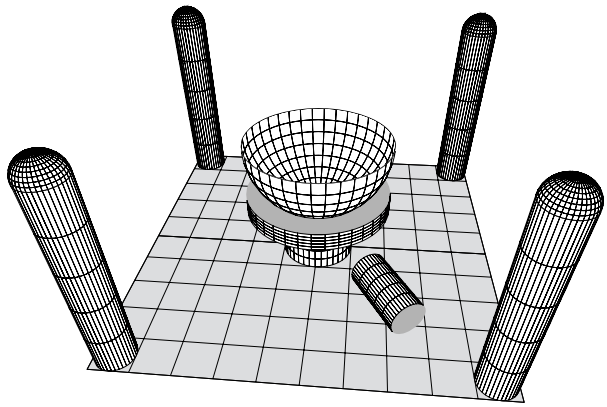
The keywords `visibleLineStyle` and `invisibleLineStyle` refer to the drawing of bodies: the macro tries to identify hidden lines and draws them with the line style `invisibleLineStyle`, while drawing the visible ones with the style `visibleLineStyle`.



Example 6-6-32

```
\usepackage{pstricks,pst-3dplot}
\begin{pspicture}(-1,-1)(3,3.25)
  \psset{Alpha=30}
  \pstThreeDCoor[xMin=-3,xMax=1,yMax=2,zMax=4]
  \pstThreeDBox(-1,1,2)(0,0,2)(2,0,0)(0,1,0)
  \pstThreeDDot[drawCoor=true,linecolor=blue](-1,1,2)
\end{pspicture}
```

```
\CylindreThreeD(0,0,0){10}{15}          \CylindreThreeD(0,0,15){20}{5}
\DemiSphereThreeD[RotX=180](0,0,35){20}
\SphereCreuseThreeD[RotX=180](0,0,35){20}
{ \psset{RotY=90,RotX=0,RotZ=30}  \CylindreThreeD(15,15,5){5}{20}  }
\multido{\iCY=-45+90}{2}{\CylindreThreeD(45,\iCY,0){5}{50}
                         \DemiSphereThreeD(45,\iCY,50){5}}
\end{pspicture}
```

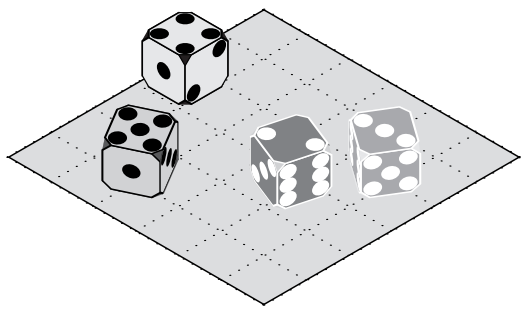


Example 6-7-39

The pst-ob3d package

This package allows you to draw basic three-dimensional objects such as cubes (which can be deformed to rectangular parallelepipeds) and dies. The package author is Denis Girou.

```
\usepackage{pst-ob3d}
\ThreeDput{\psframe[fillstyle=solid,fillcolor=black!15](6,6)
           \psgrid[subgriddiv=0,gridlabels=0,griddots=5](6,6)}
\psset{fillstyle=solid,dotscale=2,RandomFaces=true,Corners=true}
\randomi=123456 \PstDie[fillcolor=black!10](1,3,0)
\randomi=271354 \PstDie[fillcolor=black!20,viewpoint=1 0.3 1,
                        CornersColor=black!80](0.3,1.5,0)
\psset{linecolor=white}
\randomi=93850516 \PstDie[fillcolor=black!60,viewpoint=1 -0.5 1,
                          CornersColor=black!20](3,3,0)
\randomi=8873165  \PstDie[fillcolor=black!40,viewpoint=1 -0.2 1,
                          CornersColor=black!10](2,5,0)
```



Example 6-7-40

CHAPTER 7

The Xy-pic Package

Xy-pic is a general-purpose drawing package based on TEX. It works smoothly with most formats, including LATEX, AMS-LATEX, AMS-TEX, and plain TEX. It has been used to typeset complicated diagrams from numerous application areas, including category theory, automata, algebra, geometry, neural networks, and knot theory. Xy-pic's generic syntax lets you use a consistent mnemonic notation system that is based on the *logical* construction of diagrams by the combination of various elementary *visual* components. You can also write macros by combining these basic elements consistently to form higher-level structures specific to the intended application.

Xy-pic was originally written by Kristoffer Høgsbro Rose [105]. Later Ross Moore joined the development effort and the ensuing collaboration resulted in extensive revisions and extensions [104, 106].

7.1 Introducing Xy-pic

The Xy-pic system is built around an object-oriented drawing language called the *kernel*: this is a notation for composing "objects" with "methods" that correspond to the meaningful drawing operations on the object.

The kernel supports the following basic graphic notions (see Section 7.2):

- *Positions* can be specified in various formats. In particular, user-defined coordinates can be absolute or relative to previous positions, objects, object edges, or points on connections.

- *Objects* can have several forms—e.g., circular, elliptic, and rectangular—and can be adjusted in several ways, even depending on the *direction* of other objects. In particular, an object can be used to *connect* two other objects.

Enhancements to the kernel, called "options", have two main varieties: *extensions* (see Section 7.3) add more objects and methods to the repertoire (such as "curving" and "framing"), while *features* (see Section 7.4) provide notations for particular application areas (e.g., "arrows", "matrices", "polygons", "lattices", "knots"). In general, extensions provide visual components, whereas features add domain-specific notations for their logical composition.

This chapter gives examples of Xy-pic's use in various application areas. Through this "teach by example" approach, it serves as a complement to the *Xy-pic User's Guide* [106], which introduces the most used features, and the *Xy-pic Reference Manual* [104], which describes the syntax of all Xy-pic commands and their arguments. A study of our examples should put you in an excellent position to start drawing your own diagrams; we hope it will also convince you of the beauty, power, and flexibility of the Xy-pic package.

A first example of Xy-pic code

Xy-pic consists of various modules. If you are not sure which ones to load, it is probably best to load "a large set", as follows:[1]

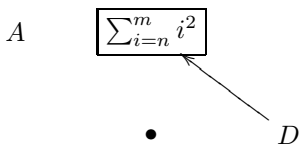
```
\usepackage[all]{xy}
```

Once you know enough about Xy-pic to identify which functions you want to use, then you can specify only the extensions or features that are actually needed. For instance,

```
\usepackage[curve,arrow,cmactex]{xy}
```

loads the `curve` extension and `arrow` feature, which are tuned to produce `\special` commands understood by Thomas Kiffe's CMacTeX Macintosh port of TEX programs.

To get an idea of the philosophy on which Xy-pic is based, let us first look at how we "construct" an Xy-picture. To make things relatively easy, we consider a matrix-like diagram. As explained in more detail in Section 7.4.2, the principal way to create a diagram is with the command `\xymatrix{`*spec*`}`, where *spec* is the specification of the *matrix entries*, which, in general, are aligned in *rows* and *columns*. Just as in a `tabular` environment, entries inside a row are separated by ampersands and successive rows are separated by `\\`.




```
\usepackage[all]{xy}
\[
\xymatrix{
 A & *+[F]{\sum_{i=n}^m {i^2}} \\
   & {\bullet} & D \ar[ul]
}\]
```

Example 7-1-1

[1] For formats other than LaTeX, use the command `\input xy` followed by `\xyoption{all}`. The `all` option loads the `curve`, `frame`, `tips`, `line`, `rotate`, and `color` extensions as well as the `matrix`, `arrow`, and `graph` features. Any other features or extensions needed must be loaded separately.

This example has two rows of three columns and shows a good deal about how XY-pic interprets commands.

- By default, entries inside XY-pic environments are typeset in mathematics mode, using "text style", and are centered.
- In many cases you may not start entries with a bare macro name—such names must be enclosed in braces or be otherwise "protected".
- As in a `tabular` environment, empty entries at the end of rows can be omitted if not referred to.
- Elements can be addressed by their *relative* ("logical") position in the diagram; thus `\ar[ul]` draws an arrow from the "current" position to the matrix cell "one up and one to the left".
- The *format* and *shape* of an element can be customized by specifying an "entry modifier" (e.g., "`[F]`" tells XY-pic to frame the entry).

If you have questions or need some help, you can address the XY-pic mailing list `xy-pic@tug.org`, to which you can subscribe by visiting the Web site `http://tug.org/mailman/listinfo/xy-pic`.

7.2 Basic constructs

A thorough knowledge of how XY-pic interprets the various commands will let you exploit its many functions fully. It will also help you understand the subtleties of the various extensions and features introduced in later sections.

A kernel XY-picture is enclosed in an `xy` environment:[1]

```
\begin{xy}...\end{xy}
```

The location at which an XY-pic object is being "dropped" is called its "position". In fact, in most cases only the coordinates or shape of the "current position" is set.

7.2.1 Initial positions

The simplest form of XY-pic position is called *absolute*, written <*X*,*Y*>. The coordinates *X* and *Y* are the offsets *right* and *above* the origin of the picture, which thus lies at `<0cm,0cm>`. Simple arithmetic operators can be used to position the current point. A comma is used to separate one position from another:

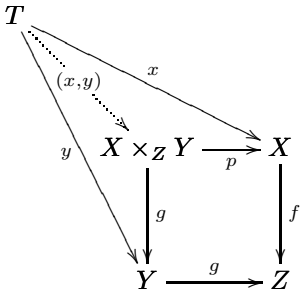
Example 7-2-1

$$\begin{array}{ccc} UL & & UR \\ & 5,5 & \\ DL & & DR \end{array}$$

```
\usepackage{xy}
\[\begin{xy}
  0*{DL} ,+/r1cm/*{DR}
 ,<0cm,1cm>*{UL} ,<1cm,1cm>*{UR}
 ,(5,5)*{5,5}
\end{xy}\]
```

[1] When using XY-pic with formats other than LaTeX, use `\xy...\endxy`.

Squares and triangles can be easily combined to create more complex diagrams. A special kind of diagram is the "pullback", which is created as follows.

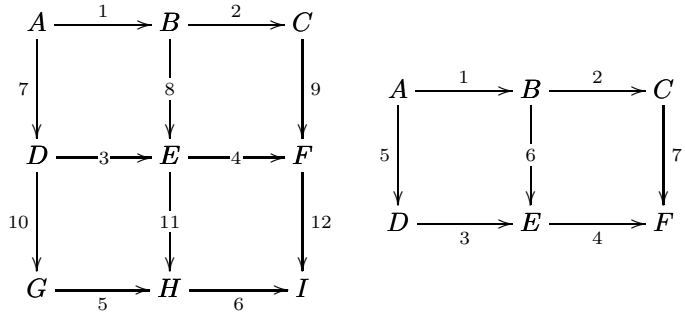



```
\usepackage{diagxy}
\[\bfig
  \pullback|brra|
           [X\times_ZY‘X‘Y‘Z;p‘g‘f‘g]%
            />‘{.>}‘>/[T;x‘(x,y)‘y]
\efig\]
```

Example 7-4-9

In homology one often encounters 3×3 and 3×2 diagrams. They are typeset with the `\iiixiii` and `\iiixii` commands, respectively, whose default behavior is displayed in the following examples. The usual order for the arrow parameters is first all horizontal arrows and then all vertical ones, left to right, and then top to bottom.

```
\usepackage{diagxy}
$\bfig \iiixiii[A‘B‘C‘D‘E‘F‘G‘H‘I; 1‘2‘3‘4‘5‘6‘7‘8‘9‘10‘11‘12] \efig$
\quad
$\bfig \iiixii[A‘B‘C‘D‘E‘F; 1‘2‘3‘4‘5‘6‘7] \efig$
```



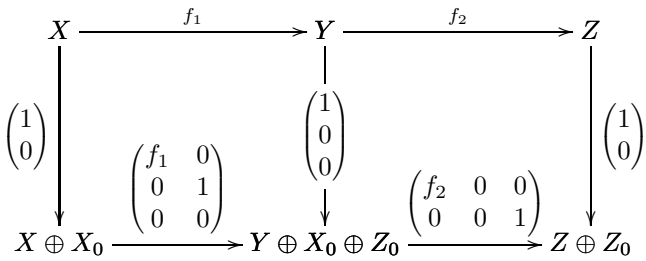
Example 7-4-10

A more interesting example of a 3×2 diagram is the following, where we add annotations (text and matrices) to the arrows. The placement of the arrow labels is specified with the first argument. Recall the order in which the arrow characteristics should be specified (see Example 7-4-10). We also load the amsmath package since we use the `pmatrix` environment.

```
\usepackage{diagxy,amsmath}
\[\bfig
 \iiixii|aaaalmr|<1000,800>
        [X‘Y‘Z‘X\oplus X_0‘Y\oplus X_0\oplus Z_0‘Z\oplus Z_0;
         f_1‘f_2‘\begin{pmatrix}f_1&0\\0&1\\0&0\end{pmatrix}‘
                 \begin{pmatrix}f_2&0&0\\0&0&1\end{pmatrix}‘
```

```
                \begin{pmatrix}1\\0\end{pmatrix}‘
                \begin{pmatrix}1\\0\\0\end{pmatrix}‘
                \begin{pmatrix}1\\0\end{pmatrix}]
\efig\]
```

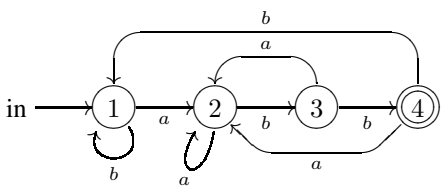
Example 7-4-11



Finite-state and stack diagrams

Finite-state diagrams can also be typeset in a straightforward way:

Example 7-4-12

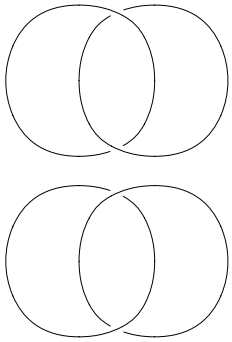



```
\usepackage[matrix,curve,arrow,tips,frame]{xy}
\[\UseTips
\entrymodifiers={++[o][F]}
\xymatrix @-1mm {
 *+\txt{in} \ar[r]
& 1 \ar@(dr,dl)[]^b \ar[r]_a
& 2 \ar@(d,dl)[]^a  \ar[r]_b
& 3 \ar ‘u[l] ‘^d[l]_a [l] \ar[r]_b
& *+++[o][F=]{4}
  \ar ‘dl_l[ll]+/d6mm/‘l_ul[ll]^a [ll]
  \ar ‘u^l[lll]+/u1cm/‘l^d[lll]_b [lll]
}\]
```

In this kind of diagram,[1] all states (elements) are enclosed in circles; here we use the `\entrymodifiers` command to specify the default modifier to realize this goal. To get nice arrowheads on the end of curves, we use Computer Modern tips. To keep the diagram a little more compact, we reduce the interelement spacing by 1 mm (`@-1mm` before the opening brace of the `\xymatrix` command). Starting an entry with an asterisk (i.e., using the form `*`⟨*object*⟩) overrides the default settings from `\entrymodifiers`; this feature is used in the leftmost cell to eliminate the frame and in the rightmost cell to typeset a double circle. Note that in the latter case the complete modifier specification had to be given. The only other tricky bit is the use of displacements towards the exterior, which add 6 mm (for a) and 1 cm (for b) in establishing the locations of the turns.

[1] We based our example on the deterministic finite automaton diagram in [7, p. 136]; another representation of the same diagram can be found in [106, Section 3.4], and we also used it for Example 3-4-10 on p. 79.

Note the use of the ^ character in the first position of the label "5", which places the label "above" the arrow while the (default) _ character places it "below".

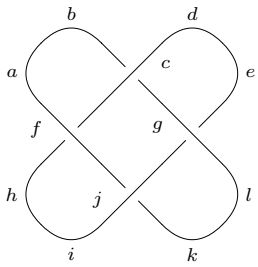


```
\usepackage[curve,knot,graph,dvips]{xy}
\[\xygraph{!{0;/r10mm/:}
          !{\vover}
      [u] !{\hcap[-2]}
      [d] !{\vover-}
    [ruu] !{\hcap[2]}
}\]
\[\begin{xy} 0;/r10mm/:
  ,\hcap[-2]\vunder\vunder-
  ,+(1,2),\hcap[2]
\end{xy}\]
```

Example 7-4-39

Since all knot crossings are, by default, bounded by a rectangle of one coordinate unit, and since loop and cap commands do not change the current point, it is convenient to use the graph feature to put together the various pieces of knot crossings and joins. This is shown in the top part of Example 7-4-39, where the \vover and \hcap commands position the elements by using "turtle" movements (up, down, left, right). The bottom part presents a variant diagram in which an explicit coordinate move was used to place the final \hcap. Note the use of the scaling factors, [2] or [-2].

Commands are also available to combine pieces in which the strings are basically at angles of 45 degrees, as in this next example.




```
\usepackage[curve,knot,arrow,dvips]{xy}
\[\renewcommand{\labelstyle}{\scriptstyle}
\begin{xy} 0;/r8mm/:
 ,{\xcapv-|{a}}
 , +(0,1) ,{\xcaph|{b}\xunderh|{c}%
          \xcaph|{d}\xcapv|{e}}
 ,-(3,0),{\xoverh|{f}}
 ,+(1,0),{\xoverh|{g}}
 ,-(3,1),{\xcapv-|{h}\xcaph-|{i}}
 ,+(0,1),{\xunderh-|{j}}
 ,+(0,-1),{\xcaph-|{k}}
 ,+(0,1),{\xcapv|{l}}
\end{xy}\]
```

Example 7-4-40

The placement of the various pieces in this construction is easy to follow by looking at the labels.

CHAPTER 8

Applications in Science, Technology, and Medicine

Because of its unsurpassed mathematical typesetting, TEX is widely used in the area of science, technology, and medicine (STM). It is not surprising, therefore, that the STM community has developed a number of packages to typeset the diagrams and schematics needed in their various disciplines. Chapter 8 of *The LATEX Companion, Second Edition* [83], describes in detail the $\mathcal{AMS}$-LATEX package, which makes marking up (higher) mathematics rather more convenient than with TEX's basic commands. Chapter 10 of that book mentions a few simple packages, such as epic, eepic, and pspicture (or the recently released pict2e), which complement LATEX's `picture` environment for drawing "simple" generic graphics. Of course, the general packages, such as METAPOST (Chapters 3 and 4) and PSTricks (Chapters 5 and 6), or even the slightly more directed XY-pic package (Chapter 7) may provide all the functionality you need to typeset even the most complex graphics. Nevertheless, the specific needs of a given user community are often better served by a more targeted approach; the packages covered in this chapter address such problem areas.

In scientific texts, precision and consistency are of the utmost importance. Therefore we start with a brief discussion of typographic conventions in scientific texts. The next two sections describe packages for typesetting chemical structures and complex biological protein topologies. Section 8.4 explores various ways of constructing Feynman diagrams, an

important tool used by physicists. The last two sections turn to electronics and describe dedicated packages for drawing timing and circuit diagrams.

8.1 Typographical rules for scientific texts

In scientific texts the typographic representation of a symbol carries a semantic meaning. Authors working in these areas should, therefore, be aware of and adhere to these typographical conventions. A brief summary of the most important rules for composing scientific texts follows (see also [52, 53, 56, 69]).

The most important rule in all circumstances is *consistency*: a given symbol should always be presented in the same way, whether it appears in the text body, a title, a figure, a table, or a formula; on the main line or as a superscript or subscript. An important corollary for LaTeX users is this: always typeset a symbol in either math or text mode—never mix the two, even if the results appear to be the same. Indeed, with LaTeX, the final visual appearance may change substantially when using a different class file or after adding a new package. For example, when using PostScript fonts, digits in text are taken from the PostScript text face and can look quite different from those in formulae. Therefore, it is good practice to always typeset numbers that refer to a result or part of a formula in math mode—i.e., surrounded by $.

In scientific texts, many symbols are traditionally typeset as *Roman* (upright) characters[1] and may not be understood properly otherwise. The most important such symbols are described here:[2]

- *Units*—for example, g, cm, s, keV. Note that physical *constants* are usually set in italics, so that units involving constants are mixed Roman–italics, e.g., keV/*c* (where *c* is the speed of light, a constant). Unit symbols are never followed by a period (see Section 8.1.1).
- *Chemical elements*—for example Ne, O, Cu—and *elementary particle names*—for example, p, K, q, H. To help the typist produce typographically correct texts, packages that contain commands representing the various names have been developed. In particular, chemists can use chemsym (see Section 8.1.2), while the PEN (Particle Entity Notation) scheme has been proposed for high-energy physics [34].[3]
- Standard mathematical functions ($\sin$, $\det$, $\cos$, $\tan$, $\Re$, $\Im$, etc.), for which the built-in LaTeX functions should be used.
- Numbers.

[1] With LaTeX, Roman type in mathematics mode can be achieved by the `\mathrm` command.

[2] See `http://physics.nist.gov/Document/typefaces.pdf` for a convenient two-page overview.

[3] Andy Buckley's heppennames package is an implementation of the PEN notation. He also wrote hepnicenames, which complements heppennames by providing more "user-friendly" names for often-occurring particles. These packages do, however, allow you too much freedom by offering the possibility to define the output style for the particle names. For instance, you can typeset their symbols in italic, a style still often (wrongly) used in American physics journals, rather than in Roman, as mandated by the IUPAP rules [56] described here. See Section 8.4.2 for an example of how these packages are used in practice.

Table 8.1: The importance of typographic rules in scientific texts

Roman Type		*Italic Type*	
A	ampere (electric unit)	A	atomic number (variable)
e	electron (particle name)	e	electron charge (constant)
g	gluon (particle name)	g	gravitational constant
l	liter (volume unit)	l	length (variable)
m	meter (length unit)	m	mass (variable)
p	proton (particle name)	p	momentum (variable)
q	quark (particle name)	q	electric charge (variable)
s	second (time unit)	s	c.m. energy squared (variable)
t	tonne (weight unit)	t	time (variable)
V	volt (electric unit)	V	volume (variable)
Z	Z boson (particle name)	Z	atomic charge (variable)

- Names of waves or states (p-wave) and covariant couplings (A for axial, V for vector); names of monopoles (E for electric, M for magnetic).
- Abbreviations that are pieces of words (exp for experimental; min for minimum).
- The "d" in integrands (e.g., $\mathrm{d}p$).

Obeying these typesetting conventions helps the reader understand at first glance the meaning of a symbol. Table 8.1 shows a few examples in which the meaning of a symbol depends on its typographic representation.

8.1.1 Getting the units right

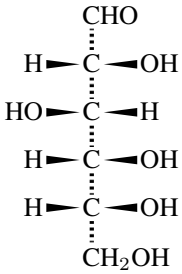
The importance of correctly typesetting units was recognized early, and several authors have developed packages to help users in this respect. Axel Reichert made a first step with his units and nicefrac packages. More recent and complete approaches are Patrick Happel's unitsdef package and Danie Els's SIstyle package. Both contain useful rules for expressing values of quantities.[1] SIstyle can be used together with Marcel Heldoorn's SIunits package. This package, which we shall describe next, is by far the more complete and provides full support for all units defined by the International System of Units (abbreviated SI[2]), the modern form of the metric system. It is the world's most widely used system of units, both in everyday com-

[1] The requirements for formatting and typesetting of SI units and numbers are described in the NIST (National Institute of Standards and Technology) document `http://physics.nist.gov/Document/sp811.pdf`. A very handy checklist for reviewing compuscripts is available from `http://physics.nist.gov/cuu/Units/rules.html`.

[2] From the French name *Système International d'Unités*. The SI was adopted by the "General Conference on Weights and Measures", which is also known under its French acronym CGPM (*Conférence Générale des Poids et Mesures*; see `http://www.bipm.fr/en/convention/cgpm/`). The CGPM meets in Paris once every four years, and the last CGPM was held in October 2003. The SI is a coherent system based on seven base units as defined in the CGPM 1960 and subsequent conferences. An overview of the SI system is available in the brochure `http://www1.bipm.org/utils/common/pdf/si_brochure_8_en.pdf` (eighth edition, 2006).

Configurations, conformations, and reaction schemes

Numerous configurations of tetrahedral molecules with wedged bonds can be drawn using variants of the command `\tetrahedral`. For instance, the following Fischer diagram, which shows the absolute configuration of the sugar D-glucose, uses four nested `\tetrahedral` commands.

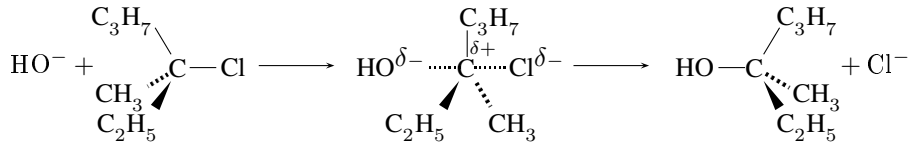



```
\usepackage{xymtexps}
\changeunitlength{0.09pt}
\tetrahedral{0==C;1A==CHO;%
             2B==H;4B==OH;3A==%
 \tetrahedral{0==C;1==(yl);%
              2B==HO;4B==H;3A==%
  \tetrahedral{0==C;1==(yl);%
               2B==H;4B==OH;3A==%
   \tetrahedral{0==C;1==(yl);%
                2B==H;4B==OH;3A==CH$\sb{2}$OH}}}}
```

Example 8-2-39

Finally, reaction schemes containing tetrahedral molecules with wedged bonds can also be handled. For instance, consider the Walden inversion reaction, which is drawn with the help of the `chemeqn` environment and the `\reactrarrow` command, both of which are defined in the chemist package (part of the XeLaTeX distribution).

```
\usepackage{xymtexps,chmst-ps}
\begin{chemeqn}
HO\sp{-}~+~
\raisebox{-28pt}{\ltetrahedralS{0==C;1==Cl;%
                                2==C$\sb{3}$H$\sb{7}$;%
                                3A==CH$\sb{3}$;4B==C$\sb{2}$H$\sb{5}$}}
\reactrarrow{0pt}{1cm}{}{}\qquad
\raisebox{-28pt}{\dtrigpyramid[{0{~~$\delta+$}}]%
                              {0==C;4A==HO$\sp{\delta-}$;%
                               5A==Cl$\sp{\delta-}$;%
                               1==C$\sb{3}$H$\sb{7}$;%
                               2A==CH$\sb{3}$;%
                               3B==C$\sb{2}$H$\sb{5}$}}
\quad\reactrarrow{0pt}{1cm}{}{}\quad
\raisebox{-28pt}{\rtetrahedralS{0==C;1==HO;%
                                2==C$\sb{3}$H$\sb{7}$;%
                                3A==CH$\sb{3}$;4B==C$\sb{2}$H$\sb{5}$}}
~+~Cl\sp{-} \label{myeqn}
\end{chemeqn}
```



(1)

Example 8-2-40

8.3.2 Membrane protein topology plots

Eric Beitz also wrote the textopo package, which provides a LaTeX interface to generate shaded membrane protein topology plots. This package provides two new environments, `textopo` and `helicalwheel`.

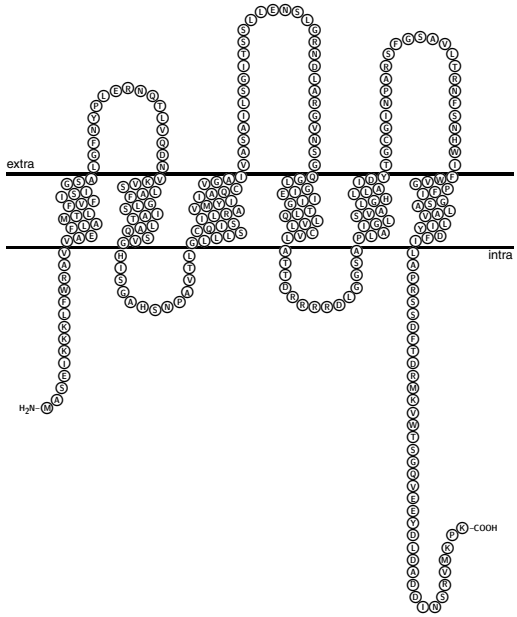
The `textopo` environment displays schematic topology plots of membrane proteins. It allows you to import sequence and topology data or alignment files in various formats. You can also manually enter the sequence and the positions of the membrane spanning domains within the environment. The package implementation will generate a basic layout from these data, which can be further adjusted by adding labels, special styles for the presentation of residues, automatic or manual shading, and annotations.

```
\begin{textopo}[parameterfile]
    textopo commands
\end{textopo}
```

The parameter file *parameterfile*, which is optional, can contain any command defined by the textopo package to specify user parameter settings. The `textopo` environment itself must contain at least one command to load the sequence and topology data for the protein that must be plotted (i.e., `\getsequence` or `\sequence` and `\MRs`, which specify the positions of the membrane regions).

The following example, which uses the file `AQP1.PHD`, comes with the distribution.

Example 8-3-6




```
\usepackage[]{textopo}
\begin{textopo}
 \getsequence{PHD}{AQP1.phd}
% no transmembrane labels
 \hideTMlabels
% small font size (range 1-10)
 \scaletopo{2}
\end{textopo}
```

The second environment, `helicalwheel`, is in its functionality quite similar to `textopo`, but produces output that shows helical transmembrane spans as helical wheels

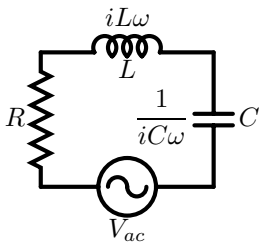
command sequence for this precedure on a Unix machine would be similar to the following (depending on where the m4 files are stored):

```
m4 /usr/local/lib/m4/libcct.m4 cirexa.m4 > cirexa.pic
gpic -t cirexa.pic > cirexa.tex
```

This leaves us with a TEX file `cirexa.tex`, which contains only the tpic code for the example. To process it further, we could include it into a LATEX source using `\input`. This stores the picture in a box register named `\graph`, so we have to add a `\usebox{\graph}` statement into the document at the spot where we want it to appear.

Customizing the diagram

To show the flexibility of the `circuit_macros` approach, let us modify our example slightly to see how it behaves with an alternating current.

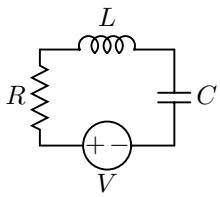



```
.PS
cct_init
linethick=1.6
define('dimen_',0.6)
loopwid = 0.9; loopht = 0.7
  source(left_ loopwid,AC); llabel(,V_{ac},)
  resistor(up_ loopht,5); llabel(,R,)
  inductor(right_ loopwid,W); rlabel(,L,); llabel(,iL\omega,)
  capacitor(down_ loopht,); llabel(,C,)
                  rlabel(,\displaystyle\frac{1}{iC\omega},)
.PE
\usebox{\graph}
```

Example 8-6-11

After specifying thick lines, we draw an alternating current (AC) source. The resistor is made a little bigger, and we specify a complex value for the impedance of the self and the capacitor. Note how we place text at either side of the element with the `llabel` and `rlabel` commands. As the label text is set in mathematics mode, you can freely use math symbols and other specific commands for math mode (e.g., `\displaystyle` to choose a larger type size for the capacitor's numerator and denominator).

Some authors prefer to draw their circuit elements using a grid. We can write an m4 macro `grid`, which has two arguments `$1` and `$2` that define the x and y coordinates at which the element is to be drawn.




```
.PS
cct_init
gridsize = 0.1
define('grid','(gridsize*'$1',gridsize*'$2')')
  source(left_ from grid(7,0) to grid(0,0),V); llabel(,V,)
  resistor(up_ from grid(0,0) to grid(0,5),4); llabel(,R,)
  inductor(right_ from grid(0,5) to grid(7,5),W); llabel(,L,)
  capacitor(down_ from grid(7,5) to grid(7,0)); llabel(,C,)
.PE
\usebox{\graph}
```

Example 8-6-12

CHAPTER 9

Preparing Music Scores

Preparing music scores of high quality is a complex task, since music notation can represent a huge amount of information about the structure and performance of a musical piece.[1] While reading a score for performing a music piece, musicians must gather all the information they need, including the pitch and the length of the notes, the rhythm, and the articulation. Depending on the instrument, the musical notation may span more than a single stave (e.g., three or more for the organ), so the amount of data to be processed concurrently can be quite large. This makes great demands on the musician's ability, especially when sight-reading a piece. The quality of the typeset score plays an important role in this process since it must clearly show the structure of the piece.

High-quality music typesetting requires a good eye and much experience. Until recently, this type of work has been done by highly trained music engravers who manage, according to Helene Wanske [136], no more than one or two pages per day. As in typesetting of text, a criterion of high quality is the overall look of the page, especially the distribution of black and white. Several texts about music notation practice have been published, but they cannot replace a practitioner when it comes to ensuring the aesthetic form of the score as a whole. The Production Committee of the Music Publisher's Association has pub-

[1] The Web site `http://www.music-notation.info/` provides a set of pointers to music notation languages, programs, fonts, etc.

lished a text that outlines a series of standards for music notation (`http://www.mpa.org/notation/notation.pdf`). *The Big Site of Music Notation and Engraving* (`http://www.coloradocollege.edu/dept/MU/Musicpress/`) intends to provide a helpful source for musicians, typesetters, students, publishers, and anyone else who is interested in music notation and engraving. See also Jean-Pierre Coulon's *Essay on the true art of music engraving* (`http://icking-music-archive.org/lists/sottisier/sottieng.pdf`).

In recent years several computer systems for writing scores have been developed. Encore (`www.encoremusic.com`), Finale (`www.finalemusic.com`), and Sibelius (`www.sibelius.com`) are examples of commercial products, while Rosegarden (`http://www.rosegardenmusic.com/`) and noteedit (`http://developer.berlios.de/projects/noteedit`) are freely available developments. All of these programs are of the WYSIWYG (What You See Is What You Get) type, and most of them have reached a genuine state of perfection. However, they cannot yet replace an experienced music engraver. All they can do to ensure high-quality typesetting is to create a "nice" draft: they contribute to a high-quality score only if they leave the aesthetic decisions to the *experienced* user.

This role is even more evident when one considers nonstandard situations, which are encountered in modern music, for which notational requirements are hard to standardize at all. Indeed, music, as a live art form, evolves continuously, and its current practice is often quite distinct from that of the 18th and 19th centuries, when the "standard" music notation was consolidated. Whereas standard notational practices are quite sufficient for popular and commercial music (and thus the favored target for commercial software), "modern" music goes well beyond this traditional form, in particular in its graphic representation. Moreover, musicology has notational needs (e.g., symbols for highlighting certain notes, unusual ties, superposition of staves) for the analysis of all kinds of music—classical and contemporary, western and oriental, ethnic from various peoples of the world—that go well beyond the possibilities of current professional typesetting applications. What is needed is a programmable system, and here TEX can be an important player.

In this chapter, after a short historical introduction (Section 9.1), we first consider MusiXTEX, a set of TEX macros that build a very powerful and flexible tool for typesetting scores. As MusiXTEX makes no aesthetic decisions—these choices must all be made by the typesetter—it is quite complex to use. Therefore several preprocessors have been developed to provide an easier interface. In Section 9.3, we introduce the abc language, which is in widespread use for folk tunes. In Section 9.5, we describe the PMX language, which makes entering polyphonic music more convenient. In Section 9.6, we have a look at the M-Tx language, an offspring of PMX, which adds, among other features, support for dealing with multi-voice lyrics in scores. In Section 9.7, we introduce LilyPond, a music typesetter written in C++, while Section 9.8 says a few words about TEX*muse*.

The *Werner Icking Music Archive* (`http://icking-music-archive.org`) contains a lot of material related to music software. In particular, it is the definitive archive of software related to MusiXTEX, including pointers to the latest developments of abc, PMX, M-Tx, and their brethren. It also contains hundreds of freely available music scores typeset with MusiXTEX, often with accompanying input files, so that it is an ideal source of examples.

This chapter is somewhat unusual as it contains little LATEX: MusiXTEX is essentially low-level TEX, albeit with a LATEX interface; some of the programs discussed to translate musical languages, such as abc, even bypass TEX altogether. We nevertheless believe that it is appro-

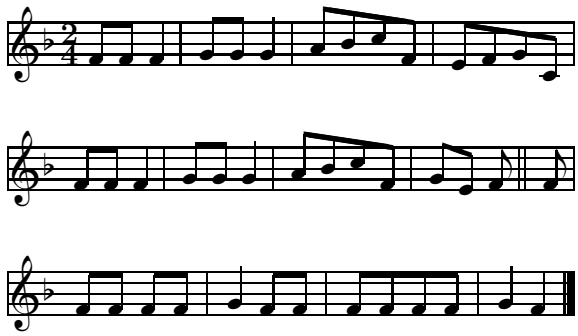
a little practice, most users can play a tune directly from the abc notation (without generating sheet music output). Moreover, the simplicity and clarity of the notation make it a straightforward matter to notate tunes that are stored in a computer file. In addition, these files can be easily exchanged by e-mail, thus enabling dissemination and discussion of the music. In fact, the abc language has become the de facto standard among folk musicians, and thousands of tunes in abc notation are now available on the Internet (see, e.g., `http://abcnotation.org.uk/tunes.html`).

9.3.1 Writing an abc source

To see how an abc source is built up, consider the following example:

Example 9-3-1

1. Sur le pont d'Avignon

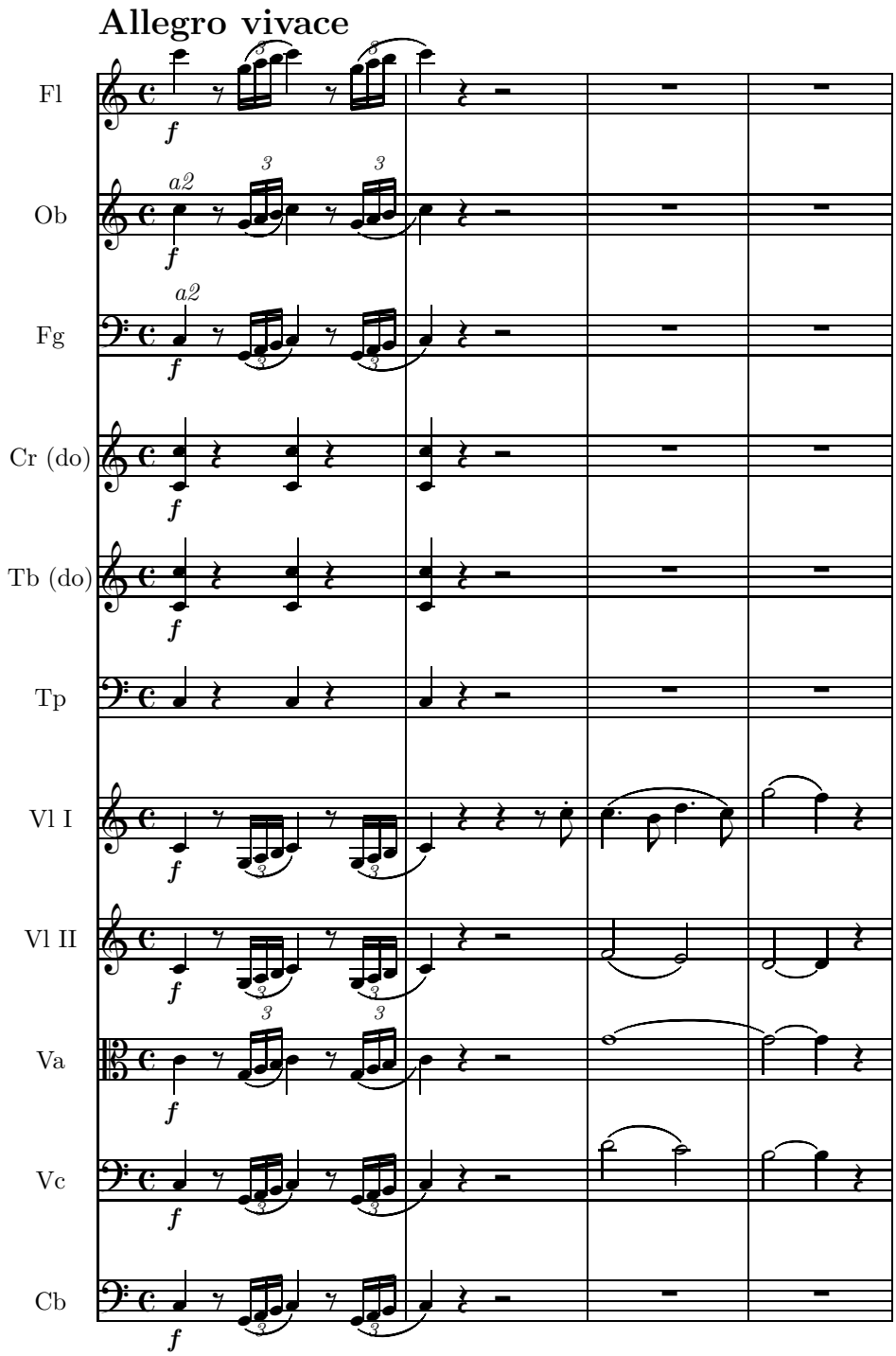


```
X:1
T: Sur le pont d'Avignon
M:2/4
L:1/8
K:F
FF F2 | GG G2 | ABcF | EFGC |
FF F2 | GG G2 | ABcF | GE F || F |
FF FF | G2 FF | FFFF | G2 F2 |]
```

An abc source consists of two parts: a header and a body. The header (shown in blue in the examples) contains information fields, each starting with an uppercase letter to denote the kind of information, followed by a colon. The body consists of the music piece itself. Within the body, additional information fields can be inserted that are used for changes to the header information (e.g., the key, meter, or tempo).

Table 9.3 shows all possible information fields, most of which are optional. A few words about the more important ones follow.

- Musical information:
  - `K:` the key, consisting of a capital letter possibly followed by a `#` or `b` for sharp or flat, respectively. You can use major keys (e.g., `K:Emaj`) or minor keys (`K:gmin`), or specify other modes, such as Mixolydian (`K:AMix`) and Dorian modes (`K:EDor`).
  - `L:` the default note length (i.e., `L:1/4` for a quarter note, `L:1/8` for an eighth note, etc.). The default note length is also set automatically by the meter field `M:`.
  - `M:` the meter, such as `M:3/4`, `M:C` (common time), or `M:C|` (cut time).



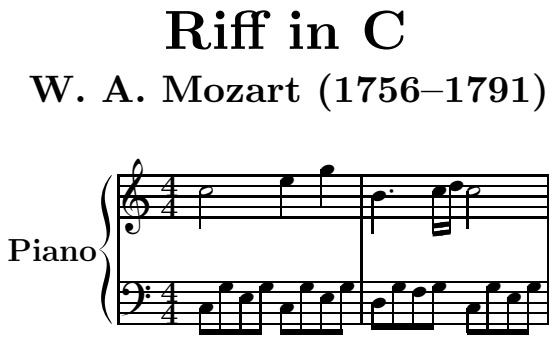
Example 9-5-36

9.6 M-Tx—Music from TeXt

After describing the PMX language we now turn to Dirk Laurie's M-Tx language,[1] which adds a layer of convenience to PMX, making entering information—in particular, in the preamble—more intuitive. By its very conception, it offers also a straightforward way for adding words (lyrics) to the music.

Let us first have another look at Section 9.4 on page 615, especially the example comparing the coding of the first bars of the Mozart piece. One large difference between PMX and M-Tx coding is that, with M-Tx voice (instrument) lines are input *as they are printed* (i.e., from top to bottom), whereas with PMX they are entered last line first (i.e., from bottom to top).

Example 9-6-1




```
Title: Riff in C
Composer: W. A. Mozart (1756--1791)
Style: piano
Name: Piano
Meter: 4/4
Size: 16
Indent: 0.18

%% w70m
c2+       e4    g   | b4d-  c1 d c2        |
c8 g+ e g c- g+ e g | d g f g    c- g+ e g |
```

Example 9-6-1 was compiled by the M-Tx processor prepmx, which transforms the M-Tx input file into a PMX file to be run through the pmxab processor.

```
> prepmx 9-6-1
==> This is M-Tx 0.60 (Music from TeXt) <16 March 2005>
==>> Input from file 9-6-1.mtx
Writing to 9-6-1.pmx
instrumentNames = TRUE
PrePMX done.  Now run PMX.

> pmxab 9-6-1
 This is PMX, Version 2.506, 14 Nov 04
 Opening 9-6-1.pmx
 Starting first PMX pass
  Bar 1  Bar 2
 Done with first pass
 Starting second PMX pass
  Bar 1  Bar 2
 Writing ./9-6-1.tex
 Done with second PMX pass.
```

The prepmx processor has several options, all of which are described in the M-Tx manual.

[1] The M-Tx entry on the home page `http://icking-music-archive.org/software/indexmt6.html` of the Icking Music Archive provides pointers to the latest version of the distribution, manual, examples, and related utilities.

9.7 The music engraver LilyPond

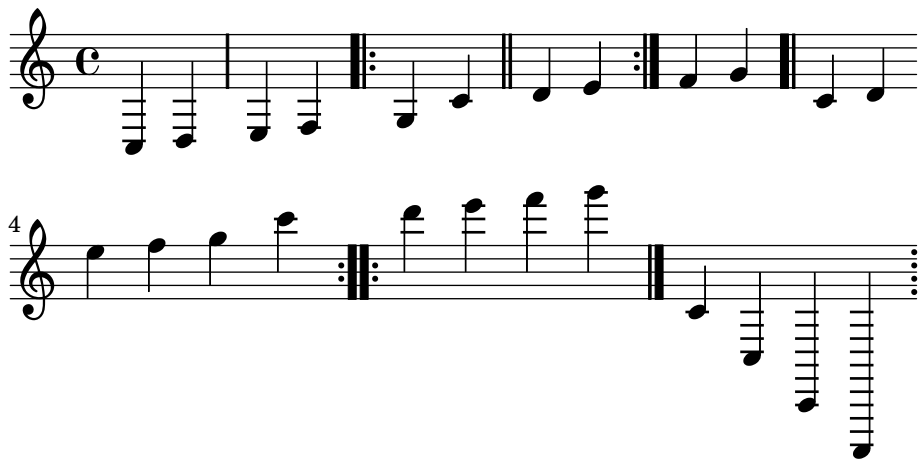
In 1996, in the previous edition of this book, we described Jan Nieuwenhuizen's MPP MusiXTEX preprocessor [89]. Since then, Jan and his colleague Han-Wen Nienhuys have abandoned that system and developed LilyPond,[1] an "automated engraving system that formats music beautifully and automatically and has a friendly syntax for its input files". They no longer use TEX as the basic typesetting engine but have developed a large C++ program (more than 6000 lines of code); they also use Python and Scheme code, as well as a specially designed font family (*feta*), which is available in various formats (PostScript Type 1, OpenType, and SVG).

9.7.1 The LilyPond source language

To typeset one note, four kinds of information can be specified: *notename*, *octave*, *duration*, and *features*. Only the *notename* is mandatory. All this information is coded in the given order with no intervening spaces; a blank separates two notes.

Notes are denoted by lowercase letters. A comma (`,`) following the letter transposes the note one octave deeper, while a right quote (`'`) makes it an octave higher. To generate different clefs, use the command `\clef` followed by either `treble`, `alto`, `tenor`, or `bass`. The following example shows some pitches and ways to generate different kinds of *bar lines*.

```
{c d   \bar "|" e f    \bar "|:"  g c'        \bar "||"
 d' e' \bar ":|" f' g' \bar ".|"  c' d'       \bar ".||." \break
 e'' f'' g'' c'''      \bar ":|:"
 d''' e''' f''' g'''   \bar "|." c' c c, c,,  \bar ":"     }
```



Example 9-7-1

[1] The LilyPond home page is at `www.lilypond.org`, where you can download the latest version of the system. There is also a tutorial, the reference guide, and much more. Of particular interest is the essay "What is behind LilyPond?", which explains the authors' views on problems in music notation (software) and their approach to solving them.

CHAPTER 10

Playing Games

Board and card games have a long history, and thousands of books in many languages have been dedicated to chess, Go, cards, and the like. These books almost always use diagrams to explain the rules or show the evolution of a game. In the present chapter we look at a number of examples showing how to prepare such graphical presentations with LaTeX.

Most game packages are concerned with making available either a special font for typesetting the right symbols or macros for producing nice examples of the state of play. The highly developed field of chess notation, however, lends itself well to an algorithmic typesetting system like LaTeX. The chess packages, with which we begin, keep track of the state of moves and allow various forms of output.

We move next to the rather similar games of Chinese chess and Go, followed by backgammon. We then look at cards, where the classic game of bridge has a special package, before concluding the chapter with the esoteric subject of crossword and Sudoku puzzles. Although crossword design is not a game, it has some similar typesetting problems, and LaTeX-using crossword makers will enjoy using the sophisticated package to help them. In the case of Sudoku, there is even a package that generates new puzzles or solves existing ones.

\ahead	\dummy	\ddummy

It is, of course, also possible to talk about the next move in a commentary started with \[or [: simply prefix the first move inside with \ahead.

If certain moves are irrelevant for the analysis you can use \dummy or \ddummy to advance the game state by one or two half-moves, respectively. This means that skak can't follow the position on the board any longer, so texmate immediately disables this functionality with \SkakOff upon encountering these commands for the remainder of the variation.

French Defense analysis:
1. e4 e6 2. d4 d5 3. ♘c3 ♗d4 4. e×d5 e×d5 5. ♗d3 ♘c6 6. a3 ♗e7 7. ♗f4! [7...♘×d4?! 8. ♗b5+! ♘c6 9. ♘×d5 ♗d6 10. ♕e2+ ♘ge7 11. ♖d1 ♗d7 12. ♗×c6 ♗×c6 13. ♘×c7+!+−] **7...a6!** [7...♗e6 8. ♘f3 ♘f6 (8...♗g4 9. h3! ♗h5 10. ♘b5!♖c8 ♗f5!+−) 9. ♘b5! ♖c8 10. ♘e5! ♘×e5 11. d×e5... 12. ♘×a7] **8. ♘f3!**

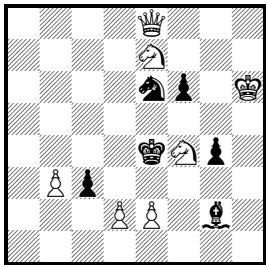
```
\usepackage{texmate}
\setchessfontfamily{leipzig}
French Defense analysis:\\
| e4 e6 ; d4 d5 ; Nc3 Bd4 ; exd5 exd5 ;
Bd3 Nc6 ; a3 Be7 ; Bf4! [ \ahead Nxd4?! ;
Bb5+! Nc6 ; Nxd5 Bd6 ; Qe2+ Nge7 ;
Rd1 Bd7 ; Bxc6 Bxc6 ; Nxc7+!\wdecisive ]
a6! [ Be6 ; Nf3 Nf6 [ Bg4 ; h3! Bh5 ; Nb5!
Rc8 ; Bf5!\wdecisive] ; Nb5! Rc8 ;
Ne5! Nxe5 ; dxe5 \dummy\,\dots Nxa7 ] Nf3! |
```

Example 10-1-19

If there are multiple variations to discuss as alternatives at a certain point in the game, you can use the variations environment or its starred form.

\begin{variations}\var *variation*$_1$ \var *variation*$_2$ ... \end{variations}

Inside the variations environment, each variation is introduced with a \var command. This will typeset the first move of a variation in boldface and separate variations by a semicolon. Alternatively, you can use \var*, in which case no special formatting is applied. The starred form variations* of the environment is equivalent to using \var* for all variations.

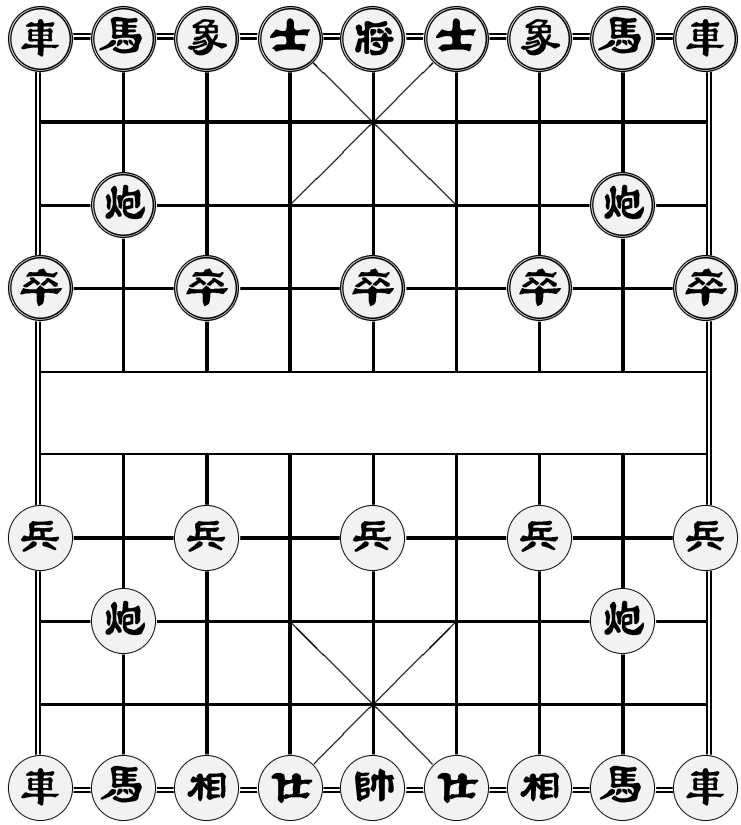


Mate in 3 moves by Bayersdorfer, 1888

1. ♘d3!△2. ♕a8+ ♔d4 3. ♕a4# [**1...♘d4** 2. ♘c5+ ♔e5 (2...♔f4 3. ♕b8#) 3. ♕b8# ; **1...c×d2** 2. ♘f5! △♕×e6# ♔d5 (2...♔×f5 3. ♕g6#) 3. ♕a8#]

```
\usepackage{texmate}
\setchessfontfamily{leipzig}

\position{4Q3/4N3/4np1K/8/4kNp1/1Pp5/3PP1b1/8}
\shortstack{\showboard\\
   Mate in 3 moves by Bayersdorfer, 1888}

| Nd3! \Threat<\withidea Qa8+ Kd4 Qa4 \#>
  [\ahead\begin{variations}
     \var  Nd4 Nc5+ Ke5 [Kf4 Qb8 \#] Qb8 \#
     \var cxd2 Nf5! \threat<Qxe6 \#>
          Kd5 [Kxf5 Qg6 \#] Qa8 \#
   \end{variations}] |
```

Example 10-1-20



Example 10-2-2

Figure 10.1: Initial setup of Chinese chess game (xiangqi)

The following listing, a mate situation after four moves, gives an example of the use of this command. The board situation after these four moves is shown in Example 10-2-4 on the following page.

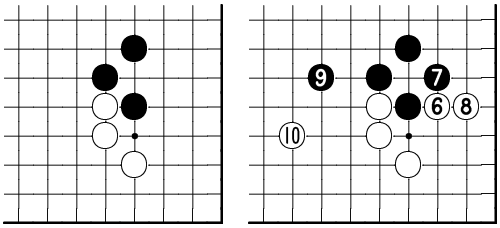
1. 炮 h3–e3 馬 b0–a8
2. 炮 e3×e7 車 a0–a9
3. 炮 b3–b5 馬 h0–g8
4. 炮 b5–e5 mates!

Example 10-2-3

```
\usepackage{cchess}
\newcommand\x{$\times$} % a shortcut to denote capture
\begin{tabbing}
1. \= \textpiece{c}h3--e3 \qquad
                          \=\textpiece{N}b0--a8 \\
2. \> \textpiece{c}e3\x e7 \>\textpiece{R}a0--a9 \\
3. \> \textpiece{c}b3--b5  \>\textpiece{N}h0--g8 \\
4. \> \textpiece{c}b5--e5 mates!
\end{tabbing}
```

The `position` environment draws a complete board. Within its body, the `\piece` command is used to place the individual pieces.

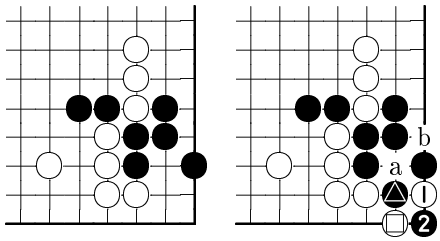
indicates the color of the first stone being placed. This method is most suitable to record games or longer sequences where the order of play needs to be indicated.



```
\usepackage{igo}
\white[\igonone]{q3,q5,p5,p6,p4,q7}
\showgoban[m1,t8]
\white[6]{r5,r6,s5,n6,m4}
\showgoban
```

Example 10-3-2

If `\white` or `\black` is used without an optional argument or if the optional argument is `\igotriangle`, `\igosquare`, `\igocircle`, or `\igocross`, then all stones typeset are of the same color and decorated with the respective glyph as specified by the optional argument. This input method is most suitable for documenting Go problems, where the order of stones placed previously is unimportant.

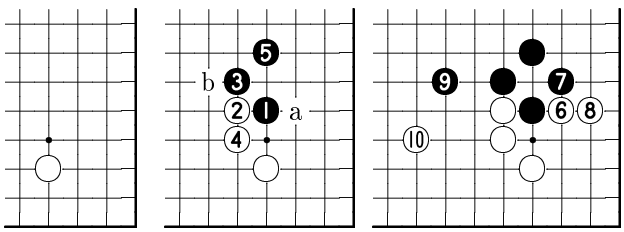


```
\usepackage{igo}
\white{o3,q2,q3,q4,r2,r5,r6,r7}
\black{p5,q5,r3,r4,s4,s5,t3}
\showgoban
\black[\igotriangle]{s2}
\white[\igosquare]{s1}
\gobansymbol{s3}{a}\gobansymbol{t4}{b}
\white[1]{t2,t1}
\showgoban
```

Example 10-3-3

`\cleargobansymbols`

Once the progress in a game has been shown in a diagram, it is customary to show the already placed stones in later diagrams without numbers, achieved by issuing a `\cleargobansymbols` command. This helps in identifying newly placed stones and makes the diagrams more readable. Whether numbering is continued is a matter of taste. Although igo supports sequentially numbered stones for a full game, for readability it is usually better to restart numbering when three-digit numbers are reached and you can afford to typeset more than a single diagram.



```
\usepackage{igo}
\white{q3}
\showgoban[p1,t8]
\black[1]{q5,p5,p6,p4,q7}
\gobansymbol{r5}{a}\gobansymbol{o6}{b}
\showgoban[n1,t8]\cleargobansymbols
\white[6]{r5,r6,s5,n6,m4}
\showgoban
```

Example 10-3-4

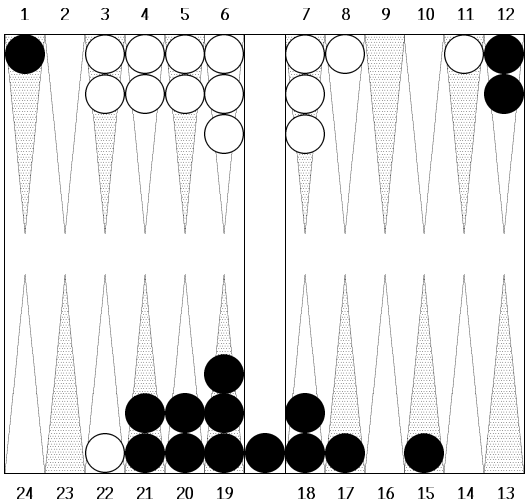
10.4 Backgammon

Jörg Richter's package **bg** defines two LaTeX environments, `position` and `game`, to display backgammon games. The `position` environment draws a single board and is thus convenient for discussing a problem, while with the `game` environment you can enter each move individually. In the latter case the board positions are stored internally, allowing the "current" status to be drawn at any time.

By convention, the homes of both players are on the left-hand side, with white's home at the top and black's home at the bottom. Unlike in the other packages discussed so far, positions on the board are not denoted with absolute coordinates but rather are numbered as viewed by the party whose move is being placed (e.g., white's 24 corresponds to black's 1, and so on). Moves are always performed from high to low numbers, and the cube is always on the right-hand side of the board.

`\begin{position}...\end{position}`

The `position` environment initializes an empty board into which stones are placed by the commands described below. Some of these commands also allow you to customize some aspects of the board's layout. The board is printed when the `\end{position}` command is encountered. Example 10-4-1 shows the use of various commands of the `position` environment.



White to play 3–2

1

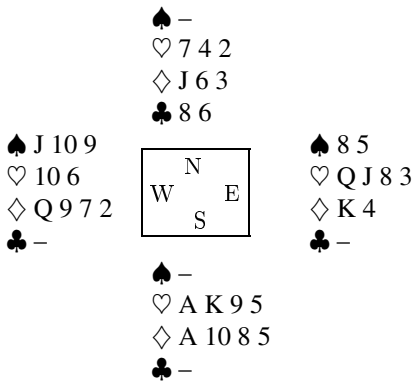
```
\usepackage{bg}
\begin{position}
\normalboard
\whitepoint{3}{2}   \whitepoint{4}{2}
\whitepoint{5}{2}   \whitepoint{6}{3}
\whitepoint{7}{3}   \whitepoint{8}{1}
\whitepoint{11}{1}  \whitepoint{22}{1}
\blackpoint{24}{1}  \blackpoint{13}{2}
\blackpoint{10}{1}  \blackpoint{8}{1}
\blackpoint{7}{2}   \blackpoint{6}{3}
\blackpoint{5}{2}   \blackpoint{4}{2}
\blackbar{1}
\shownumbers \middlecube{1} \showcube
\whiteonmove
\boardcaption{White to play 3--2}
\end{position}
```

Example 10-4-1

`\blackpoint{`*p*`}{`*n*`}` `\whitepoint{`*p*`}{`*n*`}`

These two commands are used to place stones on the board; *n* denotes the number of stones to place and *p* denotes the point where they are positioned. It is important to remember that these points are numbered downwards from 24 relative to the home position of each player.

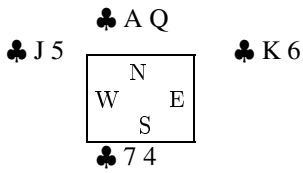
Example 10-5-4




```
\usepackage{bridge}
\crdima{}{}
 {\hand{--}{7 4 2}{J 6 3}{8 6}}
 {\hand{J 10 9}{10 6}{Q 9 7 2}{--}}
 {\hand{8 5}{Q J 8 3}{K 4}{--}}
 {\hand{--}{A K 9 5}{A 10 8 5}{--}}
```

In discussing certain techniques of play, often only the card distribution in a single suit is shown. In that case it would be nice not to use the `\hand` command in the arguments of `\crdima`, but unfortunately the result is not quite what we would expect.

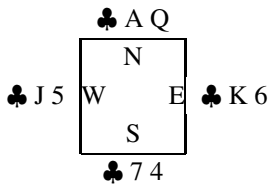
Example 10-5-5




```
\usepackage{bridge}
\crdima{}{}
  {\club{} A Q}
  {\club{} J 5}{\club{} K 6}
  {\club{} 7 4}
```

In this case a solution using the `tabular` environment gives better results. The first argument specifies the suit of interest, and the other arguments correspond to the four players (with the same order as in the `\crdima` command). Note the use of the `\multicolumn` command to suppress the vertical lines in the first and last rows.

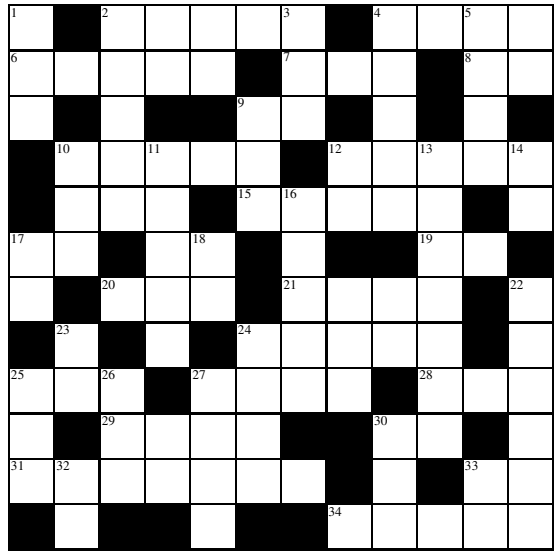
Example 10-5-6




```
\usepackage{bridge}
\newcommand{\Crdexa}[5]{{\renewcommand\arraystretch{1.2}%
   \begin{tabular}{l|@{}c@{}|l}
     \multicolumn{1}{c}{} & \multicolumn{1}{c}{#1 #2} \\
     \cline{2-2}
                              &    N      &               \\
                    #1 #3 &W\hfill\hfill E& #1 #4      \\
                              &    S      &               \\
     \cline{2-2}
     \multicolumn{1}{c}{} & \multicolumn{1}{c}{#1 #5}
   \end{tabular}}}
\Crdexa{\club}{A Q}{J 5}{K 6}{7 4}
```

Bidding

An important part of the bridge game is the initial bidding phase, in which the players decide who plays the contract. To document such a bidding sequence, Kees van der Laan introduced a `bidding` environment as an application of LaTeX's standard `tabbing` environment.



ACROSS

2 Gap between tree node labels and the node in PSTricks (5)
4 Modern replacement for scissors and glue (4)
6 A Unicode TEX variant (5)
7 . . . you always wanted to know but never dared to ask (3)
8 A graphics key that needs four numbers (2)
10 Called bb in Karl Berry's font-naming schemes (5)
12 A way to make your pages into thumbnails (5)
15 You can do it to a box but it isn't proper LATEX (5)
19 In LATEX denotes $\wp$; in other circumstances might mean a word processor (2)
20 Result of a TEX run (3)
21 A language whose name should probably have five letters, but then it was developed for Unix (4)
24 It's not Intel (5)
25 A pointer misspelled (3)
27 Testing your LATEX knowledge: $\prec$ (4)
28 Label for a signal line (3)
29 Another name for the LATEX3 project team on c.t.t. (4)
30 One way to get a sharp in MusiXTEX (2)
31 A figure or plan intended to explain rather than represent actual appearance (7)
Example 10-6-1 **33** 72.27 to an inch (2)
34 see **1d** (5)

DOWN

1 & a34 Grand wizard of TEX (3,5)
2 A ready-to-run TEX for Unix (5)
3 A novice golfer's dream (3)
4 LATEX 2_ε name for document style (5)
5 Double beam above notes in MusiXTEX (4)
9 Either/or—mathematically speaking (3)
10 German beer (3)
11 Save your coordinates (PSTricks) (5)
12 Approximation of TEX's version number (2)
13 A PostScript operator (7)
14 Probability function (2)
16 A divine messenger misspelled (5)
17 How do you get an Å? (2)
18 ξ (2)
22 LATEX has rigid and rubber ones (6)
23 Amor uses them and Xy-pic calls them (2)
24 Length of the line segment where the connector joins the first node (4)
25 Files containing LATEX font-definition documentation (3)
26 η—don't say this is all Greek to you (3)
27 $\perp$, also the first letters of everlasting (4)
30 We plot it in Chapter 4 (3)
32 TEX's name for inch (2)
33 Lula is chief of (2)

Figure 10.2: A sample crossword for you to fill in (done with crosswrd)

The size of the grid can be adjusted by setting `\sudokusize` (the default value is `10cm`), and the size and font for the numbers can be manipulated by redefining `\sudokuformat` as shown in Example 10-7-1. The default definition uses `\Huge` to fit the larger grid size. The package also offers the environment `sudoku`, which is simply an abbreviation for `sudoku-block` inside a `center` environment.

10.7.2 sudokubundle—Solving and generating Sudokus

In 2006, Peter Wilson published a bundle of three packages that not only typeset but also attempt to solve existing Sudokus or generate new ones. In contrast to the sudoku package, with Wilson's bundle the puzzles have to be stored in external files and require a somewhat different input syntax.

In these external files, only the first nine lines are relevant. Each must consist of nine characters, either a dot (representing an empty cell) or one of the numbers 1 to 9 (indicating prefilled cells). Any further lines can be used for comments and will not be read by LaTeX.

The printsudoku package provides the command `\sudoku` for typesetting such files. It also offers a `\writepuzzle` command to write external Sudokus into separate files, but for this purpose a `filecontents*` environment, as used in the next example, or a simple text editor is equally or even more suitable.

		9					6	4
4								
1			3	6			7	2
		4	6					9
			9		3			
2					5	4		
9	2			5	7			8
								5
3	4					6		

```
\usepackage{printsudoku}
\begin{filecontents*}{sample.sud}
..9....64
4........
1..36..72
..46....9
...9.3...
2....54..
92..57..8
........5
34....6..
A moderate challenge
\end{filecontents*}
\cluefont{\small}
\cellsize{1.2\baselineskip}
\sudoku{sample.sud}
```

Example 10-7-2

As seen in the previous example, the size of the puzzle and the numbers inside are controlled through `\cluefont` (default `\Huge`) and `\cellsize` (default `2.5\baselineskip`), respectively. Note that compared to the sudoku package these are declarations, rather than length registers or macros, and thus are changed in a different way. For example, to get sans serif numbers, we would need to use `\sffamily` instead of using `\textsf`.

The solvesudoku package attempts to solve a given puzzle and prints the solution as far as it was able to produce it. Given that TeX isn't the best language in which to implement complicated algorithms, it does a surprisingly good job and is able to fully resolve most

CHAPTER 11

The World of Color

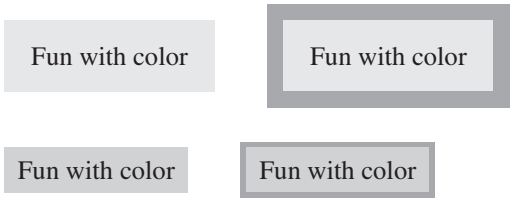
For many people, color is indispensable for effective graphics. All of the modern interactive drawing packages support coloring of lines, filling objects with color, etc., and all of the standard bitmap file formats such as GIF (Graphics Interchange Format), PNG (Portable Network Graphic), JPEG (Joint Photographic Experts Group), PBM (Portable Bitmap), TIFF (Tagged Image File Format), BMP (Windows Bitmap), SVG (Scalable Vector Graphic), and Encapsulated PostScript support color. Thus, if you generate a picture with a drawing package, and then import it into your LaTeX document using the packages described in Chapter 2, you should have no problems if your printing or viewing device supports color. However, you do have to know something about how color is represented and which color model you are using. We discuss these issues in the first part of this chapter.

If you prepare your graphics using LaTeX itself or simply want colored text, you need some special support from both LaTeX and your driver. The main body of this chapter describes the extended LaTeX xcolor package, which we believe is powerful enough to meet almost all needs and is capable of working with most other packages. xcolor extends the old color package with features such as color mixing, color sequences, and tabular shading.

LaTeX users often request color for use in presentations. The xcolor package can, of course, be used with old LaTeX slides classes, but we devote some space to explaining a more sophisticated class, beamer, and give lots of examples of its facilities.

As the book is printed in two colors, it is possible to show some color effects in examples. All other colors will appear in grayscale throughout the text. However, we repeat selected examples in the color plates. We indicate when the reader should refer to the full-color version. You can also take the example source code, run it through LaTeX or pdfLaTeX, and view the PostScript or PDF output.

Some further examples (also in Color Plate XIII b) show how to control the exact form of the box with the `\fbox` parameters `\fboxrule` and `\fboxsep`, which specify the thickness of the rule and the size of the shaded area respectively.




```
\usepackage{color}
\setlength{\fboxrule}{6pt}%
\setlength{\fboxsep}{10pt}%
\colorbox{yellow}{Fun with color}\qquad
\fcolorbox{red}{yellow}{Fun with color}
\par\bigskip\par
\setlength{\fboxrule}{2pt}%
\setlength{\fboxsep}{5pt}%
\colorbox{green}{Fun with color}\qquad
\fcolorbox{red}{green}{Fun with color}
```

Example 11-2-6

Combining the use of PostScript fonts and color, you can construct lists with colorful elements; the `\ding` command is part of the pifont package described in [83, p. 378].

- On the first day of Christmas my true love sent to me
  - a partridge in a pear tree
- On the second day of Christmas my true love sent to me
  - two turtle doves
  - and a partridge in a pear tree
- On the third day of Christmas my true love sent to me
  - three French hens
  - two turtle doves
  - and a partridge in a pear tree

```
\usepackage{pifont,color}
\newenvironment{coldinglist}[1]
        {\begin{list}{\textcolor{blue}{\ding{#1}}}{}}
        {\end{list}}
\newcommand\OnThe[1]{On the \textcolor{blue}{#1} day of
                     Christmas my true love sent to me}
\begin{coldinglist}{113}
 \item \OnThe{first}
 \begin{coldinglist}{42}
    \item a partridge in a pear tree
 \end{coldinglist}
 \item \OnThe{second}
 \begin{coldinglist}{42}
    \item two turtle doves
    \item and a partridge in a pear tree
 \end{coldinglist}
 \item \OnThe{third}
 \begin{coldinglist}{42}
    \item three French hens
    \item two turtle doves
    \item and a partridge in a pear tree
 \end{coldinglist}
\end{coldinglist}
```

Example 11-2-7

More complicated color support can be obtained in the framework of the colortbl package, which allows you to produce colored tables (see Section 11.3) or the beamer class, which makes color slides (see Section 11.4).

followed by a number. This number describes the percentage of this color to use in the mix, with the remainder being white.

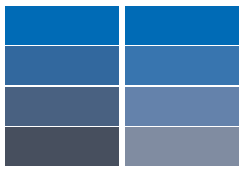
Example 11-2-13

```
\usepackage{xcolor}
\newcommand\blob[1]{{\color{#1}\rule{1.5cm}{5mm}}}
\blob{blue} \blob{blue!75} \\ \blob{blue!50} \blob{blue!25}
```

Tone and shade

What we see in this example is actually an abbreviation of the more general syntax for mixing colors: if the second color in the mix is not white, you have to specify it as well by adding it to the right, again separated by an exclamation mark. The next example shows the mixing of blue with black (called adding tone) and gray (called shading).

Example 11-2-14



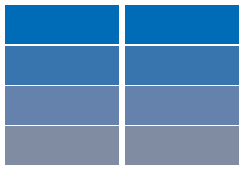
```
\usepackage{xcolor}
\newcommand\blob[1]{{\color{#1}\rule{1.5cm}{5mm}}}
\blob{blue}           \blob{blue}\\
\blob{blue!75!black} \blob{blue!75!gray}\\
\blob{blue!50!black} \blob{blue!50!gray}\\
\blob{blue!25!black} \blob{blue!25!gray}
```

Colorful mix

It is also possible to mix more than two colors in this way, but you have to understand how the algorithm works to do it successfully. Assume you have the three colors in individual buckets and some empty buckets for mixing. You mix the first two colors according to the specified percentage into a free bucket. That gives you a new color in that bucket. Then you use this color and mix it with the third color again into a free bucket, etc.

If you want to mix several colors with a specific percentage in the final mix, that can still be quite tricky. The next example reimplements the mix of blue and gray (which is a 50% mix of black and white) from the previous example. Here it is clearly simpler to first mix black and white and then blue to obtain the same results as before.

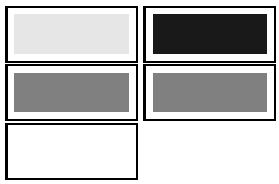
Example 11-2-15



```
\usepackage{xcolor}
\newcommand\blob[1]{{\color{#1}\rule{1.5cm}{5mm}}}
\blob{blue}                    \blob{blue}        \\
\blob{white!50!black!25!blue} \blob{blue!75!gray}\\
\blob{white!50!black!50!blue} \blob{blue!50!gray}\\
\blob{white!50!black!75!blue} \blob{blue!25!gray}
```

It is also possible to specify the complement of a color or color mix with this syntax, by putting a minus sign before the specification. The complement is the color that, if combined with the original color, yields white. However, in the example below, mixing the colors `test` and `anti` yields gray due to the fact that each of the colors in the mix consists of 50% white. Only the extended specification in the third row (explained afterwards) allows us to use 100% of each color, i.e., combine them.

Example 11-2-16



```
\usepackage{xcolor}
\colorlet{test}{yellow!90} \colorlet{anti}{-test}
\newcommand\blob[1]{\fbox{\color{#1}\rule{1.5cm}{5mm}}}
\blob{test}                \blob{anti}         \\
\blob{test!50!anti}        \blob{gray}         \\
\blob{rgb,1:test,1;anti,1}
```

To draw attention to individual rows of a table, we can put a band of color behind them (Color Plate XVI e):

Table title

Description	Column 1	Column 2
Row one	mmmmm	mmmm
Row two	mmmm	mmm
Row three	mmmmm	mmmmm
Row four	mmmmm	mmmm
Totals	mmmmm	mmmmm

```
\usepackage{colortbl}
\newcommand\panel[1]{\multicolumn{1}%
            {>{\columncolor{magenta}}#1}}
\begin{tabular}{lrr}
  \large\textbf{Table title}\\[2mm]
  \textbf{Description}
          & \textbf{Column 1}
             & \textbf{Column 2}\\[1mm]
  Row one & mmmmm & mmmm \\
  Row two &  mmmm & mmm  \\
  \panel{l}{Row three}
          & \panel{r}{mmmmm}
             & \panel{r}{mmmmm} \\
  Row four& mmmmm & mmmm  \\ \cline{2-3}
  Totals  & mmmmm & mmmmm
\end{tabular}
```

Example 11-3-13

But we can do even better: color the whole table, and leave the row to be emphasized with a white background (Color Plate XVI f):

Table title

Description	Column 1	Column 2
Row one	mmmmm	mmmm
Row two	mmmm	mmm
Row three	mmmmm	mmmmm
Row four	mmmmm	mmmm
Totals	mmmmm	mmmmm

```
\usepackage{colortbl}
\newcommand\panel[1]{\multicolumn{1}%
            {>{\columncolor{white}}#1}}
\colorbox{magenta}{%
\arrayrulecolor{black}
\begin{tabular}{lrr}
  \large\textbf{Table title}\\[2mm]
  \textbf{Description}
         & \textbf{Column 1}
             & \textbf{Column 2}\\[1mm]
  Row one & mmmmm & mmmm \\
  Row two &  mmmm & mmm  \\
  \panel{l}{Row three}
          & \panel{r}{mmmmm}
             & \panel{r}{mmmmm} \\
  Row four& mmmmm & mmmm \\ \cline{2-3}
  Totals  & mmmmm & mmmmm
\end{tabular}}
```

Example 11-3-14

This is completely analogous to the previous example except that the `\columncolor` command now uses the color white, while the `\colorbox` at the beginning makes the whole table magenta.

Now we look at ways to highlight columns rather than rows. We use the `\columncolor` command to specify the color of the columns (Color Plate XVI g):

Example 11-3-15

Table title

Description	Column 1	Column 2
Row one	mmmmm	mmmm
Row two	mmmm	mmm
Row three	mmmmm	mmmmm
Row four	mmmmm	mmmm
Totals	mmmmm	mmmmm

```
\usepackage{colortbl}
\definecolor{Bluec}{cmyk}{.60,0,0,0}
\begin{tabular}{l>{\columncolor{Bluec}}rr}
\large\textbf{Table title}\\[2mm]
\textbf{Description} & \textbf{Column 1}
            & \textbf{Column 2} \\[1mm]
Row one  & mmmmm & mmmm \\
Row two  &  mmmm &  mmm \\
Row three& mmmmm & mmmmm \\
Row four & mmmmm & mmmm \\
Totals   & mmmmm & mmmmm
\end{tabular}
```

Colored panels of this type are often used to highlight connected regions in a table. The blue shade (`Bluec`) is defined at the beginning with the standard `\definecolor` command, although we could also have combined it with `\columncolor` as

```
\columncolor[cmyk]{.60,0,0,0}
```

Another feature often encountered in color work is the color gradient (Color Plate XVI h). Here we use various levels of cyan defined at the start for successive rows. We use the extended mixing possibilities of xcolor to achieve this effect:

Example 11-3-16

Table title

Description	Column 1	Column 2
Row one	mmmmm	mmmm
Row two	mmmm	mmm
Row three	mmmmm	mmmmm
Row four	mmmmm	mmmm
Totals	mmmmm	mmmmm

```
\usepackage[table]{xcolor}
\definecolor{Cyan}{cmyk}{1,0,0,0.3}
\begin{tabular}{l rr}
\rowcolor{Cyan}\multicolumn{3}{l}
    {\large\textbf{\strut Table title}}\\[2mm]
\rowcolor{Cyan}
\textbf{Description} & \textbf{Column 1}
             & \textbf{Column 2} \\[1mm]
\rowcolor{Cyan!20}Row one  & mmmmm & mmmm \\
\rowcolor{Cyan!40}Row two  &  mmmm &  mmm \\
\rowcolor{Cyan!60}Row three& mmmmm & mmmmm\\
\rowcolor{Cyan!80}Row four & mmmmm & mmmm \\
\rowcolor{Cyan}   Totals   & mmmmm & mmmmm
\end{tabular}
```

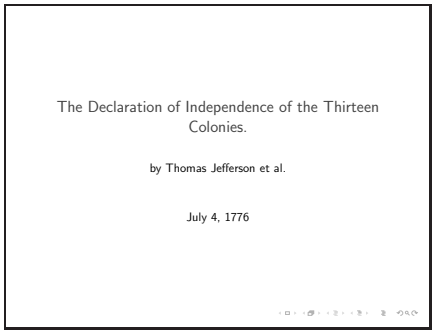
Although this task requires specifying colors for each row, the result can be quite pleasing. This technique is certainly one of those most often used to produce attractive and easily readable tabular material.

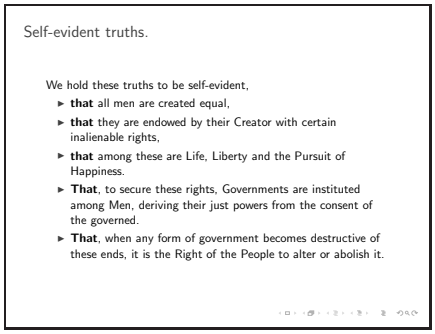
One might expect to be able to achieve the same effect by defining a color series and stepping it through each row. However, as it turns out, this approach results in the color changing for every cell: due to the implementation, the color expression is evaluated each

11.4.2 Your first slides

The beamer class comes with lengthy documentation, example files, and a lot of ready-made templates for different colors and layouts. The following example shows the default output. It is difficult to choose the right layout for the presentation—when people are more impressed by the fancy layout than by the contents, then there is something wrong! For a first-time user, it is sensible to use some of the predefined themes of beamer, and to attempt to write your own only after gaining some experience with this class.

Let us start with a simple pair of slides:






```
\documentclass{beamer}
\title{The Declaration of Independence of
       the Thirteen Colonies.}
\author{by Thomas Jefferson et al.}
\date{July 4, 1776}
\frame{\maketitle}

\section{The unanimous Declaration}
\begin{frame}
 \frametitle{Self-evident truths.}
 We hold these truths to be self-evident,
 \begin{itemize}
  \item \textbf{that} all men are created equal,
  \item \textbf{that} they are endowed by their
        Creator with certain inalienable rights,
  \item \textbf{that} among these are Life,
        Liberty and the Pursuit of Happiness.
  \item \textbf{That}, to secure these rights,
 Governments are instituted among Men, deriving
 their just powers from the consent of the governed.
  \item  \textbf{That}, when any form of government
 becomes destructive of these ends, it is the Right
 of the People to alter or abolish it.
 \end{itemize}
\end{frame}
```

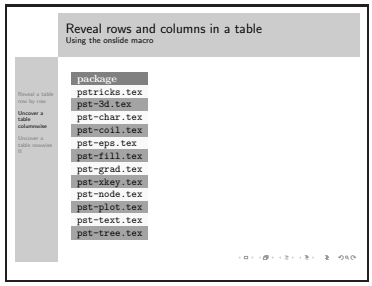
Example 11-4-1

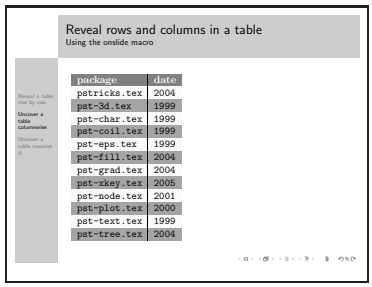
We can change appearance of the slides by choosing variants in five style levels for beamer: the theme, the outer layout, the inner layout, the color theme, and the font theme. In each case you can use the standard LaTeX `\usepackage` mechanism by preceding the style name with the word `beamertheme`, `beameroutertheme`, `beamerinnertheme`, `beamercolortheme`, or `beamerfonttheme` respectively.

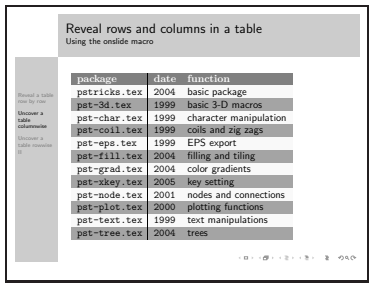
Table 11.4 lists the predefined styles that come with beamer. These themes are not official, and their contents and layout depend on what users have contributed to the community.

In the next step we choose the Malmoe main theme; this is just a name for the theme and not the official layout of the Swedish university!

the end of the last column, the use of \onslide without a specification ensures that the first column on the next row is once more shown normally, so that the whole first column is seen (the last slide is also shown in Color Plate XVI x).







Example 11-4-11

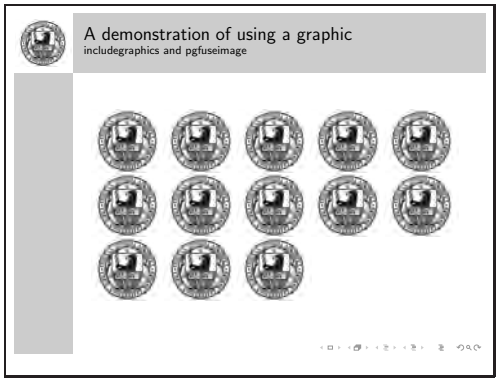
```
\documentclass[xcolor=table]{beamer}
\usetheme{Malmoe}
\useoutertheme{sidebar}
\usecolortheme{dove}
\newcommand\bfrm[1]
           {\textbf{\textrm{\textcolor{white}{#1}}}}
\section{Reveal a table row by row}
\begin{frame}
  \frametitle{Reveal rows and columns in a table}
  \framesubtitle{Using the pause macro}
  ...
\end{frame}
\section{Uncover a table columnwise}
\begin{frame}
 \frametitle{Reveal rows and columns in a table}
 \framesubtitle{Using the onslide macro}
 \rowcolors[]{1}{blue!40}{yellow!20}
 \begin{tabular}{>{\ttfamily}l<{\onslide<2->}|%
   >{\ttfamily}l<{\onslide<3->}l<{\onslide}@{}}
 \rowcolor{gray}
   \bfrm{package}&\bfrm{date}&\bfrm{function} \\
  pstricks.tex & 2004 & basic package          \\
  pst-3d.tex   & 1999 & basic 3-D macros       \\
  pst-char.tex & 1999 & character manipulation\\
  pst-coil.tex & 1999 & coils and zig zags     \\
  pst-eps.tex  & 1999 & EPS export             \\
  pst-fill.tex & 2004 & filling and tiling     \\
... further code omitted ...
```

\onslide can also be used to show specific rows of a table, as we saw earlier with \pause. The following example shows the third and fifth slides of the frame. Note that in the example the \onslide commands are added at the end of the rows (affecting the next) and not at the beginning, as that would trigger the coloring of the row.

```
\documentclass[xcolor=table]{beamer}
\usetheme{Malmoe}  \useoutertheme{sidebar}  \usecolortheme{dove}
\newcommand\bfrm[1]{\textbf{\textrm{\textcolor{white}{#1}}}}
\section{Reveal a table row by row}     \begin{frame}  ... \end{frame}
\section{Uncover a table columnwise}    \begin{frame}  ... \end{frame}
\section{Uncover a table rowwise II}
\begin{frame}
  \frametitle{Reveal rows and columns in a table}
```

```
\includegraphics<overlay spec.> [key/vals] {file name}
\pgfdeclareimage{key/vals}{beamer name }{file name}
\pgfuseimage{key/vals}{beamer name}
```

The following example shows both ways of using a graphic. The screenshot is the thirteenth slide, which is easy to control because each line has five pictures. The automatic slide control is done by the option <+-> together with the \only and \includegraphics macros.




```
\documentclass{beamer}  \usetheme{Malmoe}
\useoutertheme{sidebar} \usecolortheme{dove}
\pgfdeclareimage[width=2cm]{fu}{fu-berlin}
\newcommand\FU{\only<+->{\pgfuseimage{fu}}}
\newcommand\fu
   {\includegraphics<+->[width=2cm]{fu-berlin}}
\logo{\includegraphics[width=1.5cm]{fu-berlin}}
\begin{frame}
 \frametitle{A demonstration of using a graphic}
 \framesubtitle{includegraphics and pgfuseimage}
 \FU \fu \FU \fu \FU\par \fu \FU \fu \FU \fu\par
 \FU \fu \FU \fu \FU
\end{frame}
```

Example 11-4-34

Often a full-screen graphic is needed, which is possible with an empty frame (keyword plain) and filling the background canvas with the graphic.

```
\documentclass{beamer}  \usetheme{Malmoe}
\useoutertheme{sidebar} \usecolortheme{dove}
\setbeamertemplate{background canvas}{%
     \includegraphics[width=\paperwidth]%
                     {fu-berlin-air}}
\begin{frame}[plain]
\end{frame}
```

Example 11-4-35

This image shows the main campus of the Free University of Berlin and is courtesy of Foster & Partners.

11.4.8 Managing your templates

The beamer class is totally driven by templates, and nearly everything can be overwritten or simply defined by the user. In general there are three kinds of templates: