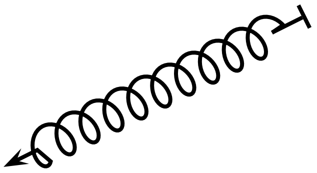


```
\pscoil[coilarm=.5cm,linewidth=1.5pt,coilwidth=.5cm]{<-|}(4,2)
```

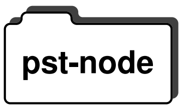
Here is an example of **\pszigzag**:

```
\pszigzag[coilarm=.5,linearc=.1]{<->}(4,0)
```

Note that **\pszigzag** uses the **linearc** parameters, and that the beginning and ending segments may be longer than **coilarm** to take up slack.

\psCoil just draws the coil horizontally from *angle1* to *angle2*. Use **\rput** to rotate and translate the coil, if desired. **\psCoil** does not use the **coilarm** parameter. For example, with **coilaspect=0** we get a sine curve:

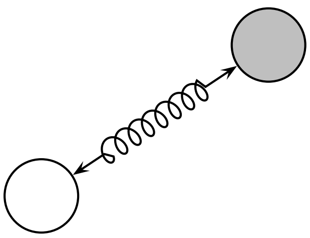
```
\psCoil[coilaspect=0,coilheight=1.33,
  coilwidth=.75,linewidth=1.5pt]{0}{1440}
```



pst-coil.tex also contains coil and zigzag node connections. You must also load pst-node.tex / pst-node.sty to use these. The node connections are:

\nccoil*[*par*]{*arrows*}**{*nodeA*}{*nodeB*}**
\nczigzag*[*par*]{*arrows*}**{*nodeA*}{*nodeB*}**
\pccoil*[*par*]{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**
\pczigzag*[*par*]{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**

The end points are chosen the same as for **\ncline** and **\pcline**, and otherwise these commands work like **\pscoil** and **\pszigzag**. For example:



```
\cnode(.5,.5){.5}{A}
\cnode[fillstyle=solid,fillcolor=lightgray](3.5,2.5){.5}{B}
\nccoil[coilwidth=.3]{<->}{A}{B}
```

34 Special coordinates

The command

\SpecialCoor

enables a special feature that lets you specify coordinates in a variety of ways, in addition to the usual Cartesian coordinates.[16] Processing is slightly slower and less robust, which is why this feature is available on demand rather than by default, but you probably won't notice the difference.

Here are the coordinates you can use:

(*x*,*y*) The usual Cartesian coordinate. E.g., (3,4).

(*r*;*a*) Polar coordinate, with radius r and angle a. The default unit for r is **unit**. E.g., (3;110).

(*node*) The center of *node*. E.g., (A).

([*par*]*node*) The position relative to *node* determined using the **angle**, **nodesep** and **offset** parameters. E.g., ([angle=45]A).

(!*ps*) Raw PostScript code. *ps* should expand to a coordinate pair. The units **xunit** and **yunit** are used. For example, if I want to use a polar coordinate $(3;110)$ that is scaled along with **xunit** and **yunit**, I can write

```
(!3 110 cos mul 3 110 sin mul)
```

(*coor1*|*coor2*) The x coordinate from *coor1* and the y coordinate from *coor2*. *coor1* and *coor2* can be any other coordinates for use with **\SpecialCoor**. For example, (A|1in;30).

\SpecialCoor also lets you specify angles in several ways:

num A number, as usual, with units given by the **\degrees** command.

[16] There is an obsolete command **\Polar** that causes coordinates in the form (*r*,*a*) to be interpreted as polar coordinates. The use of **\Polar** is not recommended because it does not allow one to mix Cartesian and polar coordinates the way **\SpecialCoor** does, and because it is not as apparent when examining an input file whether, e.g., (3,2) is a Cartesian or polar coordinate. The command for undoing **\Polar** is **\Cartesian**. It has an optional argument for setting the default units. I.e.,

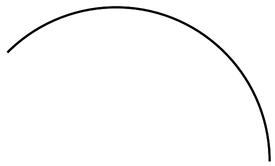
```
\Cartesian(<x>,<y>)
```

has the effect of

```
\psset{xunit=<x>,yunit=<y>}
```

\Cartesian can be used for this purpose without using **\Polar**.

(*coor*) A coordinate, indicating where the angle points to. Be sure to include the (), in addition to whatever other delimiters the angle argument uses. For example, the following are two ways to draw an arc of .8 inch radius from 0 to 135 degrees:



```
\SpecialCoor
\psarc(0,0){.8in}{0}{135}
\psarc(0,0){.8in}{0}{(-1,1)}
```

!*ps* Raw PostScript code. *ps* should expand to a number. The same units are used as with *num*.

The command

\NormalCoor

disables the **\SpecialCoor** features.

35 Overlays

Overlays are mainly of interest for making slides, and the overlay macros described in this section are mainly of interest to TEX macro writers who want to implement overlays in a slide macro package. For example, the seminar.sty package, a LATEX style for notes and slides, uses PSTricks to implement overlays.

Overlays are made by creating an \hbox and then outputting the box several times, printing different material in the box each time. The box is created by the commands

\overlaybox *stuff*\endoverlaybox

LATEX users can instead write:

```
\begin{overlaybox}  <stuff>  \end{overlaybox}
```

The material for overlay *string* should go within the scope of the command

\psoverlay{*string*}

string can be any string, after expansion. Anything not in the scope of any **\psoverlay** command goes on overlay main, and material within the scope of **\psoverlay{all}** goes on all the overlays. **\psoverlay** commands can be nested and can be used in math mode.

The command

> **\putoverlaybox{*string*}**

then prints overlay *string*.

Here is an example:

```
\overlaybox
  \psoverlay{all}
  \psframebox[framearc=.15,linewidth=1.5pt]{%
    \psoverlay{main}
    \parbox{3.5cm}{\raggedright
     Foam Cups Damage Environment {\psoverlay{one} Less than
       Paper Cups,} Study Says.}}
\endoverlaybox
  \putoverlaybox{main} \hspace{.5in} \putoverlaybox{one}
```

Foam Cups Damage
Environment

Study Says.

Less
than Paper Cups,

Driver notes: Overlays use **\pstVerb** and **\pstverbscale**.

36 The gradient fill style

gradient

The file gradient.tex/gradient.sty, along with the PostScript header file gradient.pro, defines the gradient **fillstyle**, for gradiated shading. This **fillstyle** uses the following parameters:

gradbegin=*color* **Default: gradbegin**

> The starting and ending color.

gradend=*color* **Default: gradend**

> The color at the midpoint.

gradlines=*int* **Default: 500**

The number of lines. More lines means finer gradiation, but slower printing.

gradmidpoint=*num* **Default: .9**

The position of the midpoint, as a fraction of the distance from top to bottom. *num* should be between 0 and 1.

gradangle=*angle* **Default: 0**

The image is rotated by *angle*.

gradbegin and **gradend** should preferably be rgb colors, but grays and cmyk colors should also work. The definitions of the colors gradbegin and gradend are:

```
\newrgbcolor{gradbegin}{0 .1 .95}
\newrgbcolor{gradend}{0 1 1}
```

Here are two ways to change the gradient colors:

```
\newrgbcolor{gradbegin}{1 .4 0}
```

and

```
\psset{gradbegin=blue}
```

Try this example:

```
\psframe[fillstyle=gradient,gradangle=45](10,-20)
```

37 Adding color to tables

colortab

The file colortab.tex/colortab.sty contains macros that, when used with color commands such as those in PSTricks, let you color the cells and lines in tables. See colortab.doc for more information.

38 Typesetting text along a path

textpath

The file textpath.tex/textpath.sty defines the command **\pstextpath**, for typesetting text along a path. It is a remarkable trick, but there are some caveats:

- textpath.tex only works with certain DVI-to-PS drivers. Here is what is currently known:
  - It works with Rokicki's dvips, version 5.487 or later (at least up to v5.495).
  - It does not work with earlier versions of dvips.
  - It does not work with TeXview (to preview files with NeXT-TeX 3.0, convert the .dvi file to a PostScript file with dvips -o and use Preview).
  - "Does not work" means that it has no effect, for better or for worse.
  - This may work with other drivers. The requirement is that the driver only use PostScript's show operator, unbound and unloaded, to show characters.
- You must also have installed the PostScript header file textpath.ps, and **\pstheader** must be properly defined in pstricks.con for your driver.
- Like other PSTricks that involve rotating text, this works best with PostScript (outline) fonts.
- PostScript rendering with textpath.tex is slow.

Because of all this, no samples are shown here. However, there is a test file tp-test.tex and PostScript output tp-test.ps that are distributed with PSTricks.

Here is the command:

\pstextpath[*pos*](*x*,*y*)**{*graphics object*}{*text*}**

text is placed along the path, from beginning to end, defined by the PSTricks graphics object. (This object otherwise behaves normally. Set **linestyle=none** if you don't want it to appear.)

text can only contain characters. No TeX rules, no PSTricks, and no other \special's. (These things don't cause errors; they just don't work right.) Math mode is OK, but math operators that are built from several characters (e.g., large integral signs) may break. Entire boxes (e.g., \parbox) are OK too, but this is mainly for amusement.

pos is either

l	justify on beginning of path
c	center on path
r	justify on end of path.

The default is l.

(*x*,*y*) is an offset. Characters are shifted distance *x* along path, and are shifted up by *y*. "Up" means with respect to the path, at whatever point on the path corresponding to the middle of the character. (*x*,*y*) must be Cartesian coordinates. Both coordinates use **\psunit** as the default. The default coordinate is (0,\TPoffset), where **\TPoffset** a command whose default value is -.7ex. This value leads to good spacing of the characters. Remember that ex units are for the font in effect when **\pstextpath** occurs, not inside the *text* argument.

More things you might want to know:

- Like with \rput and the graphics objects, it is up to you to leave space for \pstextpath.
- Results are unpredictable if *text* is wider than length of path.
- \pstextpath leaves the typesetting to TEX. It just intercepts the show operator to remap the coordinate system.

39 Stroking and filling character paths

charpath

The file charpath.tex/charpath.sty defines the command:

\pscharpath*[*par*]**{*text*}**

It strokes and fills the *text* character paths using the PSTricks **linestyle** and **fillstyle**.

The restrictions on DVI-to-PS drivers listed on page 76 for \pstextpath apply to \pscharpath. Furthermore, only outline (PostScript) fonts are affected.

Sample input and output files chartest.tex and chartest.ps are distributed with PSTricks.

With the optional *, the character path is not removed from the PostScript environment at the end. This is mainly for special hacks. For example, you can use \pscharpath* in the first argument of \pstextpath, and thus typeset text along the character path of some other text. See the sample file denis1.tex. (However, you cannot combine \pscharpath and \pstextpath in any other way. E.g., you cannot typeset character outlines along a path, and then fill and stroke the outlines with \pscharpath.)

The command

\pscharclip*[*par*]**{*text*} ... \endpscharclip**

works just like **\pscharpath**, but it also sets the clipping path to the character path. You may want to position this clipping path using **\rput** *inside* **\pscharclip**'s argument. Like **\psclip** and **\endpsclip**, **\pscharclip** and **\endpscharclip** should come on the same page and should be properly nested with respect to TEX groups (unless **\AltClipMode** is in effect). The file denis2.tex contains a sample of **\pscharclip**.

40 Importing EPS files

PSTricks does not come with any facility for including Encapsulated PostScript files, because there are other very good and well-tested macros for exactly that. If using Rokicki's dvips, then try epsf.tex/epsf.sty, by the man himself!

What PSTricks *is* good for is embellishing your EPS picture. You can include an EPS file in in the argument of **\rput**, as in

```
\rput(3,3){\epsfbox{myfile.eps}}
```

and hence you can include an EPS file in the **\pspicture** environment. Turn on **\psgrid**, and you can find the coordinates for whatever graphics or text you want to add. This works even when the picture has a weird bounding box, because with the arguments to **\pspicture** you control the bounding box from TEX's point of view.

This isn't always the best way to work with an EPS file, however. If the PostScript file's bounding box is the size you want the resulting picture to be, after your additions, then try

```
\hbox{<picture objects> \epsfbox{<file.eps>}
```

This will put all your picture objects at the lower left corner of the EPS file. \epsfbox takes care of leaving the right amount of space.

If you need to determine the bounding box of an EPS file, then you can try of the automatic bounding box calculating programs, such as bbfig (distributed with Rokicki's dvips). However, all such programs are easily fooled; the only sure way to determine the bounding box is visually. **\psgrid** is a good tool for this.

41 Exporting EPS files

pst2eps

You must load pst2eps.tex or pst2eps.sty to use the PSTricks macros described in this section.

If you want to export an EPS file that contains both graphics and text, then you need to be using a DVI-to-PS driver that suports such a feature. If you just want to export pure graphics, then you can use the **\PSTricksEPS** command. Both of these options are described in this section.

Newer versions of Rokicki's dvips support an -E option for creating EPS files from TEX .dvi files. E.g.,

$$dvipsfoo:dvi - E - ofoo:eps$$

Your document should be a single page. dvips will find a tight bounding box that just encloses the printed characters on the page. This works best with outline (PostScript) fonts, so that the EPS file is scalable and resolution independent.

There are two inconvenient aspects of this method. You may want a different bounding box than the one calculated by dvips (in particular, dvips ignores all the PostScript generated by PSTricks when calculating the bounding box), and you may have to go out of your way to turn off any headers and footers that would be added by output routines.

PSTricks contains an environment that tries to get around these two problems:

\TeXtoEPS
stuff
\endTeXtoEPS

This is all that should appear in your document, but headers and whatever that would normally be added by output routines are ignored. dvips will again try to find a tight bounding box, but it will treat *stuff* as if there was a frame around it. Thus, the bounding box will be sure to include *stuff*, but might be larger if there is output outside the boundaries of this box. If the bounding box still isn't right, then you will have to edit the

```
%%BoundingBox <llx lly urx ury>
```

specification in the EPS file by hand.

If your goal is to make an EPS file for inclusion in other documents, then dvips -E is the way to go. However, it can also be useful to generate an EPS file from PSTricks graphics objects and include it in the same document,[17] rather than just including the PSTricks graphics directly, because TEX gets involved with processing the PSTricks graphics only when the EPS file is initially created or updated. Hence, you can edit your file and preview the graphics, without having to process all the PSTricks graphics each time you correct a typo. This speed-up can be significant with complex graphics such as \pslistplot's with a lot of data.

To create an EPS file from PSTricks graphics objects, use

\PSTtoEPS[*par*]**{*file*}{*graphics objects*}**

The file is created immediately, and hence you can include it in the same document (after the **\PSTtoEPS** command) and as many times as you want. Unlike with dvips -E, only pure graphics objects are processed (e.g., \rput commands have no effect).

\PSTtoEPS cannot calculate the bounding box of the EPS file. You have to specify it yourself, by setting the following parameters:

bbllx=*dim*	**Default: -1pt**
bblly=*dim*	**Default: -1pt**
bburx=*dim*	**Default: 1pt**
bbury=*dim*	**Default: 1pt**

Note that if the EPS file is only to be included in a PSTricks picture with **\rput** you might as well leave the default bounding box.

\PSTricksEPS also uses the following parameters:

[17] See the preceding section on importing EPS files.

headerfile=*file* **Default: s**

()This parameter is for specifying PostScript header files that are to be included in the EPS file. The argument should contain one or more file names, separated by commas. If you have more than one file, however, the entire list must be enclosed in braces {}.

headers=*none/all/user* **Default: none**

When none, no header files are included. When all, the header files used by PSTricks plus the header files specified by the **headerfile** parameter are included. When user, only the header files specified by the **headerfile** parameter are included. If the EPS file is to be included in a TEX document that uses the same PSTricks macros and hence loads the relevant PSTricks header files anyway (in particular, if the EPS file is to be included in the same document), then **headers** should be none or user.

Help

A Boxes

Many of the PSTricks macros have an argument for text that is processed in restricted horizontal mode (in LaTeX parlance, LR-mode) and then transformed in some way. This is always the macro's last argument, and it is written {*stuff*} in this *User's Guide*. Examples are the framing, rotating, scaling, positioning and node macros. I will call these "LR-box" macros, and use framing as the leading example in the discussion below.

In restricted horizontal mode, the input, consisting of regular characters and boxes, is made into one (long or short) line. There is no line-breaking, nor can there be vertical mode material such as an entire displayed equation. However, the fact that you can include another box means that this isn't really a restriction.

For one thing, alignment environments such as \halign or LaTeX's tabular are just boxes, and thus present no problem. Picture environments and the box macros themselves are also just boxes. Actually, there isn't a single PSTricks command that cannot be put directly in the argument of an LR-box macro. However, entire paragraphs or other vertical mode material such as displayed equations need to be nested in a \vbox or LaTeX \parbox or minipage. LaTeX users should see fancybox.sty and its documentation, fancybox.doc, for extensive tips and trick for using LR-box commands.

The PSTricks LR-box macros have some features that are not found in most other LR-box macros, such as the standard LaTeX LR-box commands.

With LaTeX LR-box commands, the contents is always processed in text mode, even when the box occurs in math mode. PSTricks, on the other hand, preserves math mode, and attempts to preserve the math style as well. TeX has four math styles: text, display, script and scriptscript. Generally, if the box macro occurs in displayed math (but not in sub- or superscript math), the contents are processed in display style, and otherwise the contents are processed in text style (even here the PSTricks macros can make mistakes, but through no fault of their own). If you don't get the right style, explicitly include a \textstyle, \displaystyle, \scriptstyle or \scriptscriptstyle command at the beginning of the box macro's argument.

In case you want your PSTricks LR-box commands to treat math in the same as your other LR-box commands, you can switch this feature on and off with the commands

\psmathboxtrue
\psmathboxfalse

You can have commands (such as, but not restricted to, the math style commands) automatically inserted at the beginning of each LR-box using the

\everypsbox{*commands*}

command.[18]

If you would like to define an LR-box environment *name* from an LR-box command *cmd*, use

\pslongbox{*name*}{*cmd*}

For example, after

```
\pslongbox{MyFrame}{\psframebox}
```

you can write

```
\MyFrame  <stuff>\endMyFrame
```

instead of

```
\psframebox{<stuff>}
```

Also, LaTeX users can write

```
\begin{MyFrame}  <stuff>\end{MyFrame}
```

It is up to you to be sure that *cmd* is a PSTricks LR-box command; if it isn't, nasty errors can arise.

Environments like have nice properties:

[18]This is a token register.

- The syntax is clearer when *stuff* is long.
- It is easier to build composite LR-box commands. For example, here is a framed minipage environment for LaTeX:

```
\pslongbox{MyFrame}{\psframebox}
\newenvironment{fminipage}%
  {\MyFrame\begin{minipage}}%
  {\end{minipage}\endMyFrame}
```

- You include verbatim text and other \catcode tricks in *stuff*.

The rest of this section elaborates on the inclusion of verbatim text in LR-box environments and commands, for those who are interested. fancybox.sty also contains some nice verbatim macros and tricks, some of which are useful for LR-box commands.

The reason that you cannot normally include verbatim text in an LR-box commands argument is that TeX reads the whole argument before processing the \catcode changes, at which point it is too late to change the category codes. If this is all Greek to you,[19] then just try this LaTeX example to see the problem:

```
\psframebox{\verb+\foo{bar}+}
```

The LR-box environments defined with **\pslongbox** do not have this problem because *stuff* is not processed as an argument. Thus, this works:

```
\pslongbox{MyFrame}{\psframebox}
\MyFrame  \verb+\foo{bar}+\endMyFrame
```

\foo{bar}

The commands

\psverbboxtrue
\psverbboxfalse

switch into and out of, respectively, a special PSTricks mode that lets you include verbatim text in any LR-box command. For example:

[19] Incidentally, many foreign language macros, such as greek.tex, use \catcode tricks which can cause problems in LR-box macros.

```
\psverbboxtrue
\psframebox{\verb+\foo{bar}+}
```

$\boxed{\texttt{\textbackslash foo\{bar\}}}$

However, this is not as robust. You must explicitly group color commands in *stuff*, and LR-box commands that usually ignore spaces that follow {*stuff*} might not do so when **\psverbboxtrue** is in effect.

B Tips and More Tricks

1 How do I rotate/frame this or that with LaTeX?

See fancybox.sty and its documentation.

2 How can I suppress the PostScript so that I can use my document with a non-PostScript dvi driver?

Put the command

\PSTricksOff

at the beginning of your document. You should then be able to print or preview drafts of your document (minus the PostScript, and perhaps pretty strange looking) with any dvi driver.

3 How can I improve the rendering of halftones?

This can be an important consideration when you have a halftone in the background and text on top. You can try putting

```
\pstverb{106 45 {dup mul exch dup mul add 1.0 exch sub} setscreen}
```

before the halftone, or in a header (as in headers and footers, not as in PostScript header files), if you want it to have an effect on every page.

setscreen is a device-dependent operator.

C Including PostScript code

To learn about the PostScript language, consult Adobe's *PostScript Language Tutorial and Cookbook* (the "Blue Book"), or Henry McGilton and Mary Campione's *PostScript by Example* (1992). Both are published by Addison-Wesley. You may find that the Appendix of the Blue Book, plus an understanding of how the stack works, is all you need to write simple code for computing numbers (e.g., to specify coordinates or plots using PostScript).

You may want to define TEX macros for including PostScript fragments in various places. All TEX macros are expanded before being passed on to PostScript. It is not always clear what this means. For example, suppose you write

```
\SpecialCoor
\def\mydata{23 43}
\psline(!47 \mydata add)
\psline(!47 \mydata\ add)
\psline(!47 \mydata~add)
\psline(!47 \mydata{} add)
```

You will get a PostScript error in each of the **\psline** commands. To see what the argument is expanding to, try use TEX's \edef and \show. E.g.,

```
\def\mydata{23 43}
\edef\temp{47 \mydata add}
\show\temp
\edef\temp{47 \mydata\ add}
\show\temp
\edef\temp{47 \mydata~add}
\show\temp
\edef\temp{47 \mydata{} add}
\show\temp
```

TEX expands the code, assigns its value to \temp, and then displays the value of \temp on your console. Hit *return* to procede. You fill find that the four samples expand, respectively, to:

```
47 23 43add
47 23 43\ add
47 23 43\penalty \@M \ add
47 23 43{} add
```

All you really wanted was a space between the 43 and add. The command \space will do the trick:

```
\psline(!47 \mydata\space add)
```

You can include balance braces { }; these will be passed on verbatim to PostScript. However, to include an unbalanced left or right brace, you have to use, respectively,

\pslbrace
\psrbrace

Don't bother trying \} or \{.

Whenever you insert PostScript code in a PSTricks argument, the dictionary on the top of the dictionary stack is tx@Dict, which is PSTrick's main dictionary. If you want to define you own variables, you have two options:

Simplest Always include a @ in the variable names, because PSTricks never uses @ in its variables names. You are at a risk of overflowing the tx@Dict dictionary, depending on your PostScript interpreter. You are also more likely to collide with someone else's definitions, if there are multiple authors contributing to the document.

Safest Create a dictionary named TDict for your scratch computations. Be sure to remove it from the dictionary stack at the end of any code you insert in an argument. E.g.,

```
TDict 10 dict def TDict begin <your code> end
```

D Troubleshooting

1 Why does the document bomb in the printer when the first item in a LaTeX file is a float?

When the first item in a LaTeX file is a float, \special's in the preamble are discarded. In particular, the \special for including PSTricks's header file is lost. The workaround is to but something before the float, or to include the header file by a command-line option with your dvi-to-ps driver.

2 I converted a .dvi file to PostScript, and then mailed it to a colleague. It prints fine for me but bombs on her printer.

Here is the most likely (but not the only) cause of this problem. The PostScript files you get when using PSTricks can contain long lines. This should be acceptable to any proper PostScript interpreter, but the lines can get chopped when mailing the file. There is no way to fix this in PSTricks, but you can make a point of wrapping the lines of your PostScript files when mailing them. E.g., on UNIX you can use uuencode and uudecode, or you can use the following AWK script to wrap the lines:

```
#! /bin/sh
# This script wraps all lines
# Usage (if script is named wrap):
#      wrap < infile > outfile
awk '
BEGIN {
   N = 78      # Max line length
}
{   if (length($0)<=N)
       print
   else {
       currlength = 0
       for (i = 1; i <=NF; i++) {
          if ((currlength = currlength + length($i) + 1) > N) {
            printf               printf               currlength = length($i)
            }
          else
            printf \ %s             }
       printf       }
} '
```

3 The color commands cause extraneous vertical space to be inserted.

For example, this can happen if you start a LaTeX \parbox or a p{} column with a color command. The solution usually is to precede the color command with \leavevmode.

4 The color commands interfere with other color macros I use.

Try putting the command **\altcolormode** at the beginning of your document. This may or may not help. Be extra careful that the scope of color commands does not extend across pages. This is generally a less robust color scheme.

5 How do I stop floats from being the same color as surrounding material?

That's easy: Just put an explicit color command at the beginning of the float, e.g., **\black**.

6 When I use some color command in box macros or with \setbox, the colors get all screwed up.

If \mybox is a box register, and you write

```
\green Ho Hum.
\setbox\mybox=\hbox{Foo bar \blue fee fum}
Hi Ho. \red Diddley-dee
\box\mybox hum dee do
```

then when \mybox is inserted, the current color is red and so Foo bar comes out red (rather than green, which was the color in effect when the box was set). The command that returns from **\blue** to the current color green, when the box is set, is executed after the \hbox is closed, which means that Hi Ho is green, but hum dee do is still blue.

This odd behavior is due to the fact that TeX does not support color internally, the way it supports font commands. The first thing to do is to explicitly bracket any color commands inside the box. Second, be sure that the current color is black when setting the box. Third, make other explicit color changes where necessary if you still have problems. The color scheme invoked by **\altcolormode** is slightly better behaved if you follow the first two rules.

Note that various box macros use \setbox and so these anomalies can arise unexpectedly.

Index