The `preview` Package for LaTeX
Version 11.87

David Kastrup*

2010/02/14

1 Introduction

The main purpose of this package is the extraction of certain environments (most notably displayed formulas) from LaTeX sources as graphics. This works with DVI files postprocessed by either Dvips and Ghostscript or dvipng, but it also works when you are using PDFTeX for generating PDF files (usually also postprocessed by Ghostscript).

Current uses of the package include the `preview-latex` package for WYSIWYG functionality in the AUCTeX editing environment, generation of previews in LyX, as part of the operation of the ps4pdf package, the tbook XML system and some other tools.

Producing EPS files with Dvips and its derivatives using the `-E` option is not a good alternative: People make do by fiddling around with `\thispagestyle{empty}` and hoping for the best (namely, that the specified contents will indeed fit on single pages), and then trying to guess the baseline of the resulting code and stuff, but this is at best dissatisfactory. The preview package provides an easy way to ensure that exactly one page per request gets shipped, with a well-defined baseline and no page decorations. While you still can use the preview package with the 'classic'

```
dvips -E -i
```

invocation, there are better ways available that don't rely on Dvips not getting confused by PostScript specials.

For most applications, you'll want to make use of the `tightpage` option. This will embed the page dimensions into the PostScript or PDF code, obliterating the need to use the `-E -i` options to Dvips. You can then produce all image files with a single run of Ghostscript from a single PDF or PostScript (as opposed to EPS) file.

*dak@gnu.org

Various options exist that will pass TEX dimensions and other information about the respective shipped out material (including descender size) into the log file, where external applications might make use of it.

The possibility for generating a whole set of graphics with a single run of Ghostscript (whether from LATEX or PDFLATEX) increases both speed and robustness of applications. It is also feasible to use dvipng on a DVI file with the options

```
-picky -noghostscript
```

to omit generating any image file that requires Ghostscript, then let a script generate all missing files using Dvips/Ghostscript. This will usually speed up the process significantly.

2 Package options

The package is included with the customary

```
\usepackage[⟨options⟩]{preview}
```

You should usually load this package as the last one, since it redefines several things that other packages may also provide.

The following options are available:

`active` is the most essential option. If this option is not specified, the `preview` package will be inactive and the document will be typeset as if the `preview` package were not loaded, except that all declarations and environments defined by the package are still legal but have no effect. This allows defining previewing characteristics in your document, and only activating them by calling LATEX as

```
latex '\PassOptionsToPackage{active}{preview}
\input{⟨filename⟩}'
```

`noconfig` Usually the file `prdefault.cfg` gets loaded whenever the `preview` package gets activated. `prdefault.cfg` is supposed to contain definitions that can cater for otherwise bad results, for example, if a certain document class would otherwise lead to trouble. It also can be used to override any settings made in this package, since it is loaded at the very end of it. In addition, there may be configuration files specific for certain `preview` options like `auctex` which have more immediate needs. The `noconfig` option suppresses loading of those option files, too.

`psfixbb` Dvips determines the bounding boxes from the material in the DVI file it understands. Lots of PostScript specials are not part of that. Since the TEX boxes do not make it into the DVI file, but merely characters, rules and specials do, Dvips might include far too small areas. The option `psfixbb` will include `/dev/null` as a graphic file in the ultimate upper left and lower right corner of the previewed box. This will make Dvips generate an appropriate bounding box.

`dvips` If this option is specified as a class option or to other packages, several packages pass things like page size information to Dvips, or cause crop marks or draft messages written on pages. This seriously hampers the usability of previews. If this option is specified, the changes will be undone if possible.

`pdftex` If this option is set, PDFTEX is assumed as the output driver. This mainly affects the `tightpage` option.

`xetex` If this option is set, XeTEX is assumed as the output driver. This mainly affects the `tightpage` option.

`displaymath` will make all displayed math environments subject to preview processing. This will typically be the most desired option.

`floats` will make all float objects subject to preview processing. If you want to be more selective about what floats to pass through to a preview, you should instead use the `\PreviewSnarfEnvironment` command on the floats you want to have previewed.

`textmath` will make all text math subject to previews. Since math mode is used throughly inside of LATEX even for other purposes, this works by redefining `\(`, `\)` and `$` and the `math` environment (apparently some people use that). Only occurences of these text math delimiters in later loaded packages and in the main document will thus be affected.

`graphics` will subject all `\includegraphics` commands to a preview.

`sections` will subject all section headers to a preview.

`delayed` will delay all activations and redefinitions the `preview` package makes until `\begin{document}`. The purpose of this is to cater for documents which should be subjected to the `preview` package without having been prepared for it. You can process such documents with

```
latex '\RequirePackage[active,delayed,⟨options⟩]{preview}
\input{⟨filename⟩}'
```

This relaxes the requirement to be loading the `preview` package as last package.

⟨***driver***⟩ loads a special driver file `pr`⟨*driver*⟩`.def`. The remaining options are implemented through the use of driver files.

`auctex` This driver will produce fake error messages at the start and end of every preview environment that enable the Emacs package `preview-latex` in connection with AUCTEX to pinpoint the exact source location where the previews have originated. Unfortunately, there is no other reliable means of passing the current TEX input position *in* a line to external programs. In order to make the parsing more robust, this option also switches off quite a few diagnostics that could be misinterpreted.

You should not specify this option manually, since it will only be needed by automated runs that want to parse the pseudo error messages. Those runs will then use `\PassOptionsToPackage` in order to effect the desired behaviour. In addition, `prauctex.cfg` will get loaded unless inhibited by the `noconfig` option. This caters for the most frequently encountered problematic commands.

`showlabels` During the editing process, some people like to see the label names in their equations, figures and the like. Now if you are using Emacs for editing, and in particular `preview-latex`, I'd strongly recommend that you check out the RefTEX package which pretty much obliterates the need for this kind of functionality. If you still want it, standard LATEX provides it with the `showkeys` package, and there is also the less encompassing `showlabels` package. Unfortunately, since those go to some pain not to change the page layout and spacing, they also don't change `preview`'s idea of the TEX dimensions of the involved boxes. So if you are using `preview` for determing bounding boxes, those packages are mostly useless. The option `showlabels` offers a substitute for them.

`tightpage` It is not uncommon to want to use the results of `preview` as graphic images for some other application. One possibility is to generate a flurry of EPS files with

> `dvips -E -i -Pwww -o` ⟨*outputfile*⟩`.000` ⟨*inputfile*⟩

However, in case those are to be processed further into graphic image files by Ghostscript, this process is inefficient since all of those files need to be processed one by one. In addition, it is necessary to extract the bounding box comments from the EPS files and convert them into page dimension parameters for Ghostscript in order to avoid full-page graphics. This is not even possible if you wanted to use Ghostscript in a *single* run for generating the files from a single PostScript file, since Dvips will in that case leave no bounding box information anywhere.

The solution is to use the `tightpage` option. That way a single command line like

> `gs -sDEVICE=png16m -dTextAlphaBits=4 -r300`
> `-dGraphicsAlphaBits=4 -dSAFER -q -dNOPAUSE`
> `-sOutputFile=`⟨*outputfile*⟩`%d.png` ⟨*inputfile*⟩`.ps`

will be able to produce tight graphics from a single PostScript file generated with Dvips *without* use of the options `-E -i`, in a single run.

The `tightpage` option actually also works when using the `pdftex` option and generating PDF files with PDFTEX. The resulting PDF file has separate page dimensions for every page and can directly be converted with one run of Ghostscript into image files.

If neither `dvips` or `pdftex` have been specified, the corresponding option will get autodetected and invoked.

If you need this in a batch environment where you don't want to use `preview`'s automatic extraction facilities, no problem: just don't use any of the extraction options, and wrap everything to be previewed into `preview` environments. This is how LyX does its math previews.

If the pages under the `tightpage` option are just too tight, you can adjust by setting the length `\PreviewBorder` to a different value by using `\setlength`. The default value is `0.50001bp`, which is half of a usual PostScript point, rounded up. If you go below this value, the resulting page size may drop below `1bp`, and Ghostscript does not seem to like that. If you need finer control, you can adjust the bounding box dimensions individually by changing the macro `\PreviewBbAdjust` with the help of `\renewcommand`. Its default value is

```
\newcommand \PreviewBbAdjust {-\PreviewBorder
-\PreviewBorder \PreviewBorder \PreviewBorder}
```

This adjusts the left, lower, right and upper borders by the given amount. The macro must contain 4 TEX dimensions after another, and you may not omit the units if you specify them explicitly instead of by register. PostScript points have the unit `bp`.

`lyx` This option is for the sake of LyX developers. It will output a few diagnostics relevant for the sake of LyX' preview functionality (at the time of writing, mostly implemented for math insets, in versions of LyX starting with 1.3.0).

`counters` This writes out diagnostics at the start and the end of previews. Only the counters changed since the last output get written, and if no counters changed, nothing gets written at all. The list consists of counter name and value, both enclosed in `{}` braces, followed by a space. The last such pair is followed by a colon (`:`) if it is at the start of the preview snippet, and by a period (`.`) if it is at the end. The order of different diagnostics like this being issued depends on the order of the specification of the options when calling the package.

Systems like `preview-latex` use this for keeping counters accurate when single previews are regenerated.

`footnotes` This makes footnotes render as previews, and only as their footnote symbol. A convenient editing feature inside of Emacs.

The following options are just for debugging purposes of the package and similar to the corresponding TEX commands they allude to:

`tracingall` causes lots of diagnostic output to appear in the log file during the preview collecting phases of TEX's operation. In contrast to the similarly named TEX command, it will not switch to `\errorstopmode`, nor will it change the setting of `\tracingonline`.

`showbox` This option will show the contents of the boxes shipped out to the DVI files. It also sets `\showboxbreadth` and `\showboxdepth` to their maximum values at the end of loading this package, but you may reset them if you don't like that.

3 Provided Commands

`preview` The `preview` environment causes its contents to be set as a single preview image. Insertions like figures and footnotes (except those included in minipages) will typically lead to error messages or be lost. In case the `preview` package has not been activated, the contents of this environment will be typeset normally.

`nopreview` The `nopreview` environment will cause its contents not to undergo any special treatment by the `preview` package. When `preview` is active, the contents will be discarded like all main text that does not trigger the `preview` hooks. When `preview` is not active, the contents will be typeset just like the main text.

Note that both of these environments typeset things as usual when preview is not active. If you need something typeset conditionally, use the `\ifPreview` conditional for it.

`\PreviewMacro` If you want to make a macro like `\includegraphics` (actually, this is what is done by the `graphics` option to `preview`) produce a preview image, you put a declaration like

```
\PreviewMacro[*[[!]{\includegraphics}
```

or, more readable,

```
\PreviewMacro[{*[][]{}}]{\includegraphics}
```

into your preamble. The optional argument to `\PreviewMacro` specifies the arguments `\includegraphics` accepts, since this is necessary information for properly ending the preview box. Note that if you are using the more readable form, you have to enclose the argument in a `[{` and `}]` pair. The inner braces are necessary to stop any included `[]` pairs from prematurely ending the optional argument, and to make a single `{}` denoting an optional argument not get stripped away by TEX's argument parsing.

The letters simply mean

`*` indicates an optional `*` modifier, as in `\includegraphics*`.

`[` indicates an optional argument in brackets. This syntax is somewhat baroque, but brief.

`[]` also indicates an optional argument in brackets. Be sure to have encluded the entire optional argument specification in an additional pair of braces as described above.

`!` indicates a mandatory argument.

`{}` indicates the same. Again, be sure to have that additional level of braces around the whole argument specification.

`?`⟨***delimiter***⟩`{`⟨***true case***⟩`}{`⟨***false case***⟩`}` is a conditional. The next character is checked against being equal to ⟨*delimiter*⟩. If it is, the specification ⟨*true case*⟩ is used for the further parsing, otherwise ⟨*false case*⟩ will be employed. In neither case is something consumed from the input, so `{`⟨*true case*⟩`}` will still have to deal with the upcoming delimiter.

`@{`⟨***literal sequence***⟩`}` will insert the given sequence literally into the executed call of the command.

`-` will just drop the next token. It will probably be most often used in the true branch of a `?` specification.

`#{`⟨***argument***⟩`}{`⟨***replacement***⟩`}` is a transformation rule that calls a macro with the given argument and replacement text on the rest of the argument list. The replacement is used in the executed call of the command. This can be used for parsing arbitrary constructs. For example, the `[]` option could manually be implemented with the option string `?[{#{[#1]}{[{#1}]}}{}`. PStricks users might enjoy this sort of flexibility.

`:{`⟨***argument***⟩`}{`⟨***replacement***⟩`}` is again a transformation rule. As opposed to `#`, however, the result of the transformation is parsed again. You'll rarely need this.

There is a second optional argument in brackets that can be used to declare any default action to be taken instead. This is mostly for the sake of macros that influence numbering: you would want to keep their effects in that respect. The default action should use `#1` for referring to the original (not the patched) command with the parsed options appended. Not specifying a second optional argument here is equivalent to specifying `[#1]`.

`\PreviewMacro*` A similar invocation `\PreviewMacro*` simply throws the macro and all of its arguments declared in the manner above away. This is mostly useful for having things like `\footnote` not do their magic on their arguments. More often than not, you don't want to declare any arguments to scan to `\PreviewMacro*` since you would want the remaining arguments to be treated as usual text and typeset in that manner instead of being thrown away. An exception might be, say, sort keys for `\cite`.

A second optional argument in brackets can be used to declare any default action to be taken instead. This is for the sake of macros that influence numbering: you would want to keep their effects in that respect. The default action might use `#1` for referring to the original (not the patched) command with the parsed options appended. Not specifying a second optional argument here is equivalent to specifying `[]` since the command usually gets thrown away.

As an example for using this argument, you might want to specify

```
\PreviewMacro*\footnote[{[]}][#1{}]
```

This will replace a footnote by an empty footnote, but taking any optional parameter into account, since an optional paramter changes the numbering scheme. That way the real argument for the footnote remains for processing by `preview-latex`.

`\PreviewEnvironment` `\PreviewEnvironment*`

The macro `\PreviewEnvironment` works just as `\PreviewMacro` does, only for environments. And the same goes for `\PreviewEnvironment*` as compared to `\PreviewMacro*`.

`\PreviewSnarfEnvironment`

This macro does not typeset the original environment inside of a preview box, but instead typesets just the contents of the original environment inside of the preview box, leaving nothing for the original environment. This has to be used for figures, for example, since they would

1. produce insertion material that cannot be extracted to the preview properly,
2. complain with an error message about not being in outer par mode.

`\PreviewOpen` `\PreviewClose`

Those Macros form a matched preview pair. This is for macros that behave similar as `\begin` and `\end` of an environment. It is essential for the operation of `\PreviewOpen` that the macro treated with it will open an additional group even when the preview falls inside of another preview or inside of a `nopreview` environment. Similarly, the macro treated with `\reviewClose` will close an environment even when inactive.

`\ifPreview`

In case you need to know whether `preview` is active, you can use the conditional `\ifPreview` together with `\else` and `\fi`.