LaTeX Environments to Image Format

v1.8 — 2020-05-23*

CTAN: https://www.ctan.org/pkg/ltximg
https://github.com/pablgonz/ltximg

Abstract

`ltximg` is a `perl` *script* that automates the process of extracting and converting environments provided by tikz, pstricks and other packages from ⟨*input file*⟩ to image formats and standalone files using `ghostscript` and `poppler-utils`. Generates a file with only extracted environments and another with all extracted environments converted to `\includegraphics`.

Contents

1 License

2 Motivation and Acknowledgments

The original idea was to extend the functionality of the `pst2pdf`[8] script (only for `pspicture` and `postscript`) to work with `tikzpicture` and other environments.

The tikz[2] package allows to *externalize* the environments, but, the idea was to be able to extend this to any type of environment covering three central points:

1. Generate a separate image files for environments.
2. Generate a standalone files with only the extracted environments.
3. Generate a file replacing the environments by `\includegraphics`.

*This file describes a documentation for version 1.8, last revised 2020-05-23.

†E-mail: «pablgonz@yahoo.com»

From the side of TeX there are some packages that cover several of these points such as the preview[1], xcomment[11], extract[12] and cachepic[13] packages among others, but none covered all points.

In the network there are some solutions in bash that were able to extract and convert environments, but in general they presented problems when the document contained *"verbatim style"* code or were only available for Linux.

Analysed the situation the best thing was to create a new *"script"* that was able to cover the three points and was multi platform, the union of all these ideas is born ltximg.

This script would not be possible without the great work of Herbert Voß author of pst2pdf[1] and Heiko Oberdiek author of pdfcrop[2]. Several parts of the code have been taken and adapted from both scripts.

Finding the correct *"regular expressions"* and writing a *"good documentation"* would be the great mission (which does not end yet).

3 How it works

It is important to have a general idea of how the *"extraction and conversion"* process works and the requirements that must be fulfilled so that everything works correctly, for this we must be clear about some concepts related to how to work with the ⟨*input file*⟩, the ⟨*verbatim content*⟩ and the ⟨*steps process*⟩.

4 Requirements for operation

For the complete operation of ltximg you need to have a modern TeX distribution such as TeXLive or MiKTeX, have a version equal to or greater than 5.28 of perl, a version equal to or greater than 9.24 of ghostscript and have a version equal to or greater than 0.52 of poppler-utils.

The distribution of TeXLive 2020 for Windows includes ltximg and all requirements, MiKTeX users must install the appropriate software for full operation.

The script has been tested on Windows (v10) and Linux (fedora 32) in x64 architecture using ghostscript v9.52, poppler-utils v0.84, perl v5.30 and the standard classes offers by LaTeX: book, report, article and letter. If an ⟨*output file*⟩ is generated, the graphicx[9] and grfext[10] packages will be needed.

5 The input file

The ⟨*input file*⟩ must comply with certain characteristics in order to be processed, the content at the beginning and at the end of the ⟨*input file*⟩ is treated in a special way, before \documentclass and after \end{document} can go any type of content, internally the script will *"split"* the ⟨*input file*⟩ at this points.

If the ⟨*input file*⟩ contains files using \input{⟨*file*⟩} or \include{⟨*file*⟩} these will not be processed, from the side of the *script* they only represent lines within the file, if you want them to be processed it is better to use the latexpand[3] first and then process the file.

Like \input{⟨*file*⟩} or \include{⟨*file*⟩}, blank lines, vertical spaces and tab characters are treated literally, for the *script* the ⟨*input file*⟩ is just a set of characters, as if it was a simple text file. It is advisable to format the source code ⟨*input file*⟩ using utilities such as chktex[4] and latexindent[5], especially if you want to extract the source code of the environments.

Both \thispagestyle{⟨*style*⟩} and \pagestyle{⟨*style*⟩} are treated in a special way by the script, if they do not appear in the preamble then \pagestyle{⟨*empty*⟩} will be added and if they are present and {⟨*style*⟩} is different from {⟨*empty*⟩} this will be replaced by {⟨*empty*⟩}.

This is necessary for the image creation process, it does not affect the ⟨*output file*⟩, but it does affect the *standalone* files.

For the script the process of dividing the ⟨*input file*⟩ into four parts and then processing them:

```
% Part One: Everything before \documentclass
\documentclass{article}
% Part two: Everything between \documentclass and \begin{document}
\begin{document}
% Part three: : Everything between \begin{document} and \end{document}
\end{document}
% Part Four: Everything after \end{document}
```

[1] https://ctan.org/pkg/pst2pdf
[2] https://ctan.org/pkg/pdfcrop
[3] https://www.ctan.org/pkg/latexpand
[4] https://www.ctan.org/pkg/chktex
[5] https://www.ctan.org/pkg/latexindent

6 Verbatim contents

One of the greatest capabilities of this script is to *"skip"* the complications that ⟨*verbatim content*⟩ produces with the extraction of environments using tools outside the "TEX world"[6]. In order to *"skip"* the complications, the ⟨*verbatim content*⟩ is classified into three types:

- Verbatim in line.
- Verbatim standard.
- Verbatim write.

6.1 Verbatim in line

The small pieces of code written using a *"verbatim macro"* (including ⋆ argument) are considered ⟨*verbatim in line*⟩, such as `\verb|`⟨*code*⟩`|` or `\macro{`⟨*code*⟩`}`.

Most *"verbatim macro"* provide by packages minted[17], fancyvrb[15] and listings[16] have been tested and are fully supported. They are automatically detected the *verbatim macro* generates by `\newmint` and `\newmintinline` and the following list:

- `\mint`
- `\spverb`
- `\qverb`
- `\fverb`
- `\verb`
- `\Verb`
- `\lstinline`
- `\pyginline`
- `\pygment`
- `\Scontents`
- `\tcboxverb`
- `\mintinline`

Some packages define abbreviated versions for *"verbatim macro"* as `\DefineShortVerb`, `\lstMakeShortInline` and `\MakeSpecialShortVerb`, will be detected automatically if are declared explicitly in ⟨*input file*⟩.

The following consideration should be kept in mind for some packages that use abbreviations for verbatim macros, such as shortvrb[14] or doc[14] for example in which there is no explicit macro in the document by means of which the abbreviated form can be detected, for automatic detection need to find `\DefineShortVerb` explicitly to process it correctly. The solution is quite simple, just add in ⟨*input file*⟩:

```
\UndefineShortVerb{\|}
\DefineShortVerb{\|}
```

depending on the package you are using. If your *"verbatim macro"* is not supported by default or can not detect, use the options described in 10.2 and 10.3.

6.2 Verbatim standard

These are the *"classic"* environments for *"writing code"* are considered ⟨*verbatim standard*⟩, such as `verbatim` and `lstlisting` environments. The following list (including ⋆ argument) is considered as ⟨*verbatim standard*⟩ environments:

- `Example`
- `CenterExample`
- `SideBySideExample`
- `PCenterExample`
- `PSideBySideExample`
- `verbatim`
- `Verbatim`
- `BVerbatim`
- `LVerbatim`
- `SaveVerbatim`
- `PSTcode`
- `LTXexample`
- `tcblisting`
- `spverbatim`
- `minted`
- `listing`
- `lstlisting`
- `alltt`
- `comment`
- `chklisting`
- `verbatimtab`
- `listingcont`
- `boxedverbatim`
- `demo`
- `sourcecode`
- `xcomment`
- `pygmented`
- `pyglist`
- `program`
- `programl`
- `programL`
- `programs`
- `programf`
- `programsc`
- `programt`

They are automatically detected ⟨*verbatim standard*⟩ environments (including ⋆ argument) generates by commands:

- `\DefineVerbatimEnvironment`
- `\NewListingEnvironment`
- `\DeclareTCBListing`
- `\ProvideTCBListing`
- `\lstnewenvironment`
- `\newtabverbatim`
- `\specialcomment`
- `\includecomment`
- `\newtcblisting`
- `\NewTCBListing`
- `\newverbatim`
- `\NewProgram`
- `\newminted`

[6] Only TEX can understand TEX, all other languages and programs are just lines in a file.

If any of the ⟨*verbatim standard*⟩ environments is not supported by default or can not detected, you can use the options described in 10.2 and 10.3.

6.3 Verbatim write

Some environments have the ability to write *"external files"* or *"store content"* in memory, these environments are considered ⟨*verbatim write*⟩, such as `scontents`, `filecontents` or `VerbatimOut` environments. The following list is considered (including `*` argument) as ⟨*verbatim write*⟩ environments:

- `scontents`
- `filecontents`
- `tcboutputlisting`
- `tcbexternal`
- `tcbwritetmp`
- `extcolorbox`
- `extikzpicture`
- `VerbatimOut`
- `verbatimwrite`
- `filecontentsdef`
- `filecontentshere`
- `filecontentsdefmacro`
- `filecontentsdefstarred`
- `filecontentsgdef`
- `filecontentsdefmacro`
- `filecontentsgdefmacro`

They are automatically detected ⟨*verbatim write*⟩ (including `*` argument) environments generates by commands:

- `\renewtcbexternalizetcolorbox`
- `\renewtcbexternalizeenvironment`
- `\newtcbexternalizeenvironment`
- `\newtcbexternalizetcolorbox`
- `\newenvsc`

If any of the ⟨*verbatim write*⟩ environments is not supported by default or can not detected, you can use the options described in 10.2 and 10.3.

7 Steps process

For creation of the image formats, extraction of source code of environments and creation of an ⟨*output file*⟩, **ltximg** need a various steps. Let's assume that the ⟨*input file*⟩ is `test.tex`, ⟨*output file*⟩ is `test-out`, the working directory are `workdir/`, the directory for images are `workdir/images` and the temporary directory is `/tmp` and we want to generate images in `pdf` format together with the source code of the environments.

We will use the following code as `test.tex`

```
% Some commented lines at begin file
\documentclass{article}
\usepackage{tikz}
\begin{document}
Some text
\begin{tikzpicture}
  Some code
\end{tikzpicture}
Always use \verb|\begin{tikzpicture}| and \verb|\end{tikzpicture}| to open
and close environment
\begin{tikzpicture}
  Some code
\end{tikzpicture}
Some text
\begin{verbatim}
\begin{tikzpicture}
  Some code
\end{tikzpicture}
\end{verbatim}
Some text
\end{document}
Some lines that will be ignored by the script
```

7.1 Validating Options

The first step is read and validated `[`⟨*options*⟩`]` from the command line and `test.tex`, verifying that `test.tex` contains *some* environment to extract, check the name and extension of `test-out`, check the directory `workdir/images` if it doesn't exist create it and create a temporary directory `/tmp/hG45uVklv9`.

The entire `test.tex` file is loaded into memory and split to start the extraction process.

7.2 Comment and ignore

In the second step, once the file `test.tex` is loaded and divided in memory, proceeds (in general terms) as follows:

> Search the words `\begin{` and `\end{` in verbatim standard, verbatim write, verbatim in line and commented lines, if it finds them, converts to `\BEGIN{` and `\END{`, then places all code to extract inside the `\begin{preview}` ...`\end{preview}`.

At this point all the code you want to extract is inside `\begin{preview}` ...`\end{preview}`.

7.3 Creating files and extracting

In the third step, the script extract a sub files `test-fig-1.tex`, `test-fig-2.tex`, ... and saved in `./images`. A temporary file with a random number (1981 for example) with all environments is created and proceed in two ways according to the [⟨*options*⟩] passed to the script:

1. If script is call *whitout* `-n`,`--noprew` options, adds the following lines to the beginning of the `test.tex` (in memory):

   ```
   \AtBeginDocument{%
   \RequirePackage[active,tightpage]{preview}
   \renewcommand\PreviewBbAdjust{-60pt -60pt 60pt 60pt}}%
   % rest of input file
   ```

   And save in a temporary file `test-fig-1981.tex` in `workdir/`.

2. If script is call *whit* `-n`,`--noprew` options, all code to extract its put inside the `preview` environment. The `\begin{preview}`...`\end{preview}` lines are only used as delimiters for extracting the content *without* using the package preview.

   Creating a temporary file `test-fig-1981.tex` in `workdir/` whit the same preamble of `test.tex` but the body only contains code that you want to extract.

If `--norun` is passed, the temporary file `test-fig-1981.tex` is renamed to `test-fig-all.tex` and moved to `workdir/images`.

7.4 Generate image formats

In the fourth step, the script run:

```
[user@machine ~:]$ ⟨compiler⟩ -recorder -shell-escape test-fig-1981.tex
```

generating the file `test-fig-1981.pdf` whit all code extracted, move `test-fig-1981.pdf` to `/tmp/hG45uVklv9` and rename to `test-fig-all.pdf`, separate in individual files `test-fig-1.pdf` and `test-fig-2.pdf` and copy to `workdir/images`. The file `test-fig-1981.tex` is moved to the `workdir/images` and rename to `test-fig-all.tex`.

Note the options passed to ⟨*compiler*⟩ does not include `-output-directory` (it is not supported) and always use `-recorder -shell-escape`.

7.5 Create output file

In the fifth step, the script creates the output file `test-out.tex` converting all extracted code to `\includegraphics` and adding the following lines at end of preamble:

```
\usepackage{graphicx}
\graphicspath{{images/}}
\usepackage{grfext}
\PrependGraphicsExtensions*{.pdf}
```

The script will try to detect whether the graphicx package and the `\graphicspath` command are in the preamble of the ⟨*output file*⟩. If it is not possible to find it, it will read the log file generated by the temporary file. Once the detection is complete, the package grfext will be added then proceed to run:

```
[user@machine ~:]$ ⟨compiler⟩ -recorder -shell-escape test-out.tex
```

generating the file `test-out.pdf`.

7.6 Clean temporary files and dirs

In the sixth step, the script read the files `test-fig-1981.fls` and `test-out.fls`, extract the information from the temporary files and dirs generated in the process and then delete them together with the directory `/tmp/hG45uVklv9`.

Finally the output file `test-out.tex` looks like this:

```
% some commented lines at begin document
\documentclass{article}
\usepackage{tikz}
\graphicspath{{images/}}
\usepackage{grfext}
\PrependGraphicsExtensions*{.pdf}
\begin{document}
Some text
\includegraphics[scale=1]{test-fig-1}
Always use \verb|\begin{tikzpicture}| and \verb|\end{tikzpicture}| to open
and close environment
\includegraphics[scale=1]{test-fig-2}
Some text
\begin{verbatim}
\begin{tikzpicture}
  Some code
\end{tikzpicture}
\end{verbatim}
Some text
\end{document}
```

8 Extract content

The script provides two ways to ⟨*extract*⟩ content from ⟨*input file*⟩, using ⟨*environments*⟩ and ⟨*docstrip tags*⟩. Some environment (including `*` argument) are supported by default. If environments are nested, the outermost one will be extracted.

8.1 Default environments

`\begin{preview}` ⟨*env content*⟩ `\end{preview}`

Environment provide by preview[1] package. If `preview` environments found in the ⟨*input file*⟩ will be extracted and converted these. Internally the script converts all environments to extract in `preview` environments. Is better comment this package in preamble unless the option `-n,--noprew` is used.

`\begin{pspicture}` ⟨*env content*⟩ `\end{pspicture}`

Environment provide by pstricks[3] package. The plain TeX syntax `\pspicture ... \endpspicture` its converted to LaTeX syntax `\begin{pspicture} ... \end{pspicture}` if not within the `PSTexample` environment.

`\begin{psgraph}` ⟨*env content*⟩ `\end{psgraph}`

Environment provide by pst-plot[4] package. The plain TeX `\psgraph ... \endpsgraph` its converted to LaTeX syntax `\begin{psgraph} ... \end{psgraph}` if not within the `PSTexample` environment.

`\begin{postscript}` ⟨*env content*⟩ `\end{postscript}`

Environment provide by pst-pdf[5] and auto-pst-pdf[6] packages. Since the pst-pdf and auto-pst-pdf packages internally use the preview package, is better comment this in preamble.

`\begin{tikzpicture}` ⟨*env content*⟩ `\end{tikzpicture}`

Environment provide by tikz[2] package. The plain TeX `\tikzpicture ... \tikzpicture` its converted to LaTeX syntax `\begin{tikzpicture} ... \end{tikzpicture}` but no a short syntax `\tikz ... ;`.

`\begin{pgfpicture}` ⟨*env content*⟩ `\end{pgfpicture}`

Environment provide by pgf[2] package. Since the script uses a *"recursive regular expression"* to extract environments, no presents problems if present `pgfinterruptpicture`.

`\begin{PSTexample}` ⟨*env content*⟩ `\end{PSTexample}`

Environment provide by pst-exa[7] packages. The script automatically detects the `\begin{PSTexample} ...\end{PSTexample}` environments and processes them as separately compiled files. The user should have loaded the package with the [swpl] or [tcb] option and run the script using `--latex` or `--xetex`.

If you need to extract other environments you can use one of the options described in 10.2 or 10.3.

8.2 Extract whit docstrip tags

%<*ltximg> ⟨*content*⟩ %</ltximg>

All content included between %<*ltximg> ... %</ltximg> is extracted. The tags can *not* be nested and should be at the beginning of the line and in separate lines. Internally the script converts all this tags to preview environments.

```
% no space before open tag %<*
%<*ltximg>
code to extract
%</ltximg>
% no space before close tag %</
```

8.3 Prevent extraction and remove

Sometimes you do not want to *"extract all"* the environments from ⟨*input file*⟩ or you want to remove environments or arbitrary content. The script provides a convenient way to solve this situation.

\begin{nopreview} ⟨*env content*⟩ \end{nopreview}

Environment provide by preview package. Internally the script converts all *"skip"* environments to \begin{nopreview} ... \end{nopreview}. Is better comment this package in preamble unless the option -n,--noprew is used.

%<*noltximg> ⟨*content*⟩ %</noltximg>

All content betwen %<*noltximg> ... %</noltximg> are ignored and no extract. The tags can *not* be nested and should be at the beginning of the line and in separate lines. Internally the script converts all this tags to nopreview environments.

```
% no space before open tag %<*
%<*noltximg>
no extract this
%</noltximg>
% no space before close tag %</
```

%<*remove> ⟨*content*⟩ %</remove>

All content betwen %<*remove> ... %</remove> are deleted in the ⟨*output file*⟩. The tags can *not* be nested and should be at the beginning of the line and in separate lines.

```
% no space before open tag %<*
%<*remove>
lines removed in output file
%</remove>
% no space before close tag %</
```

The content will be deleted if it is *"not"* within a ⟨*verbatim*⟩ or ⟨*verbatim write*⟩ environment. If you want to remove specific environments automatically you can use one of the options described in 10.2 or 10.3.

9 Image Formats

The ⟨*image formats*⟩ generated by the **ltximg** using ghostscript and poppler-utils are the following command lines:

pdf The image format generated using ghostscript. The line executed by the system is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=pdfwrite -dPDFSETTINGS=/prepress
```

eps The image format generated using pdftoeps. The line executed by the system is:

```
[user@machine ~:]$ pdftops -q -eps
```

png The image format generated using ghostscript. The line executed by the system is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=pngalpha -r 150
```

jpg The image format generated using ghostscript. The line executed by the system is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=jpeg -r 150 -dJPEGQ=100 \
                      -dGraphicsAlphaBits=4 -dTextAlphaBits=4
```

ppm The image format generated using pdftoppm. The line executed by the system is:

```
[user@machine ~:]$ pdftoppm -q -r 150
```

`tif` The image format generated using `ghostscript`. The line executed by the system is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=tiff32nc -r 150
```

`svg` The image format generated using `pdftocairo`. The line executed by the system is:

```
[user@machine ~:]$ pdftocairo -q -r 150
```

`bmp` The image format generated using `ghostscript`. The line executed by the system is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=bmp32b -r 150
```

The script auto detects the `ghostscript`, but not `poppler-utils`. You should keep this in mind if you are using the script directly and not the version provided in your TEX distribution

10 How to use

10.1 Syntax

The syntax for `ltximg` is simple, if your use the version provided in your TEX distribution:

```
[user@machine ~:]$ ltximg ⟨compiler⟩ [⟨options⟩] [--] ⟨input file⟩
```

If the development version is used:

```
[user@machine ~:]$ perl ltximg ⟨compiler⟩ [⟨options⟩] [--] ⟨input file⟩
```

The extension valid for ⟨*input file*⟩ are `.tex` or `.ltx`, relative or absolute paths for files and directories is not supported. If used without ⟨*compiler*⟩ and [⟨*options*⟩] the extracted environments are converted to `pdf` image format and saved in the `./images` directory using `pdflatex` and `preview` package.

10.2 Command line interface

The script provides a *command line interface* with short `-` and long `--` option, they may be given before the name of the ⟨*input file*⟩, the order of specifying the options is not significant. Options that accept a ⟨*value*⟩ require either a blank space ␣ or `=` between the option and the ⟨*value*⟩. Multiple short options can be bundling and if the last option takes a ⟨*comma separated list*⟩ you need `--` at the end.

`-h, --help` ⟨*bolean*⟩ (default: off)

Display a command line help and exit.

`-l, --log` ⟨*bolean*⟩ (default: off)

Write a `.log` file with debug information.

`-v, --version` ⟨*bolean*⟩ (default: off)

Display the current version (1.8) and exit.

`-V, --verbose` ⟨*bolean*⟩ (default: off)

Show verbose information in terminal.

`-d, --dpi` ⟨*integer*⟩ (default: 150)

Dots per inch for images files.

`-t, --tif` ⟨*bolean*⟩ (default: off)

Create a `.tif` images files using `ghostscript`.

`-b, --bmp` ⟨*bolean*⟩ (default: off)

Create a `.bmp` images files using `ghostscript`.

`-j, --jpg` ⟨*bolean*⟩ (default: off)

Create a `.jpg` images files using `ghostscript`.

`-p, --png` ⟨*bolean*⟩ (default: off)

Create a `.png` transparent image files using `ghostscript`.

`-e, --eps` ⟨*bolean*⟩ (default: off)

Create a `.eps` image files using `pdftops`.

`-s, --svg` ⟨*bolean*⟩ (default: off)

Create a `.svg` image files using `pdftocairo`.

`-P, --ppm` ⟨*bolean*⟩ (default: off)

Create a `.ppm` image files using `pdftoppm`.

`-g, --gray` ⟨*bolean*⟩ (default: off)

Create a gray scale for all images using `ghostscript`. The line behind this options is:

```
[user@machine ~:]$ gs -q -dNOSAFER -sDEVICE=pdfwrite -dPDFSETTINGS=/prepress       \
                      -sColorConversionStrategy=Gray -dProcessColorModel=/DeviceGray
```

`-f, --force` ⟨*bolean*⟩ (default: off)

Try to capture `\psset{`⟨*code*⟩`}` and `\tikzset{`⟨*code*⟩`}` to extract. When using the `--force` option the script will try to capture `\psset{`⟨*code*⟩`}` or `\tikzset{`⟨*code*⟩`}` and leave it inside the `preview` environment, any line that is between `\psset{`⟨*code*⟩`}` and `\begin{pspicture}` or between `\tikzset{`⟨*code*⟩`}` and `\begin{tikzpicture}` will be captured.

`-n, --noprew` ⟨*bolean*⟩ (default: off)

Create images files without preview package. The `\begin{preview}...\end{preview}` lines are only used as delimiters for extracting the content *without* using the package preview. Sometimes it is better to use it together with `--force`.

`-m, --margin` ⟨*integer*⟩ (default: 0)

Set margins in bp for `pdfcrop`.

`-o, --output` ⟨*file name*⟩ (default: empty)

Create ⟨*file name*⟩ whit all extracted environments converted to `\includegraphics`. Only ⟨*file name*⟩ must be passed *without* relative or absolute paths.

`--prefix` ⟨*string*⟩ (default: fig)

Set ⟨*prefix*⟩ append to each generated files.

`--verbcmd` ⟨*macro name*⟩ (default: myverb)

Set custom verbatim command `\myverb`. Just pass the ⟨*name*⟩ of the macro *without* "\".

`--imgdir` ⟨*string*⟩ (default: images)

Set the name of directory for save generated files. Only the name of directory must be passed without *without* or absolute paths.

`--zip` ⟨*bolean*⟩ (default: off)

Compress the files generated by the script in `./images` in `.zip` format. Does not include ⟨*output file*⟩.

`--tar` ⟨*bolean*⟩ (default: off)

Compress the files generated by the script in `./images` in `.tar.gz` format. Does not include ⟨*output file*⟩.

`--srcenv` ⟨*bolean*⟩ (default: off)

Create separate files whit *"only code"* for all extracted environments. This option is designed to generate ⟨*standalone files*⟩ with *"only code"* of the environments, is mutually exclusive whit `--subenv`.

`--subenv` ⟨*bolean*⟩ (default: off)

Create sub files whit *"preamble and code"* for all extracted environments. This option is designed to generate ⟨*standalone compilable files*⟩ for each extracted environment, is mutually exclusive whit `--srcenv`.

`--norun` ⟨*bolean*⟩ (default: off)

Execute the script, but do not create image files. This option is designed to be used in conjunction with `--srcenv` or `--subenv` and to debug the ⟨*output file*⟩.

`--nopdf` ⟨*bolean*⟩ (default: off)

Don't create a `.pdf` image files.

`--nocrop` ⟨*bolean*⟩ (default: off)

Don't run `pdfcrop` in image files.

`--arara` ⟨*bolean*⟩ (default: off)

Use `arara` for compiler files.

`--xetex` ⟨*bolean*⟩ (default: off)

Using `xelatex` compiler ⟨*input file*⟩ and ⟨*output file*⟩.

`--latex` ⟨*bolean*⟩ (default: off)

Using `latex»dvips»ps2pdf` compiler in ⟨*input file*⟩ and `pdflatex` for ⟨*output file*⟩.

`--dvips` ⟨*bolean*⟩ (default: off)

Using `latex»dvips»ps2pdf` for compiler ⟨*input file*⟩ and ⟨*output file*⟩.

`--dvipdf` ⟨*bolean*⟩ (default: off)

Using `latex»dvipdfmx` for compiler ⟨*input file*⟩ and ⟨*output file*⟩.

`--luatex` ⟨*bolean*⟩ (default: off)

Using `lualatex` for compiler ⟨*input file*⟩ and ⟨*output file*⟩.

`--clean` ⟨`doc|pst|tkz|all|off`⟩ (default: doc)

Removes specific content in ⟨*output file*⟩. Valid values for this option are:

`doc` All content after `\end{document}` is removed.
`pst` All `\psset{`⟨*code*⟩`}` and pstricks package is removed in ⟨*preamble*⟩ and ⟨*body*⟩.
`tkz` All `\tikzset{`⟨*code*⟩`}` is removed in ⟨*body*⟩.
`all` Activates `doc`, `pst` and `tkz`.
`off` Deactivate all.

`--extrenv` ⟨*comma separated list*⟩ (default: empty)

Add environments to extract, if it's the last option passed need `--` at the end.

`--skipenv` ⟨*comma separated list*⟩ (default: empty)

Add environments that should *"not be extracted"* and that the script supports by default, if it's the last option passed need `--` at the end.

`--verbenv` ⟨*comma separated list*⟩ (default: empty)

Add ⟨*verbatim standard*⟩ environment support, if it's the last option passed need `--` at the end.

`--writenv` ⟨*comma separated list*⟩ (default: empty)

Add ⟨*verbatim write*⟩ environment support, if it's the last option passed need `--` at the end.

`--deltenv` ⟨*comma separated list*⟩ (default: empty)

Add environments to deleted in ⟨*output file*⟩. The environments are delete only in ⟨*body*⟩, if it's the last option passed need `--` at the end.

10.3 Options from input file

Many of the ideas in this section are inspired by the arara[7] tool. A very useful way to pass options to the script is to place them in commented lines at the beginning of the file, very much in the *"style of* arara*"*.

`% ltximg:` ⟨*argument*⟩`: {`⟨*option one, option two, option three, ...*⟩`}`

`%!ltximg:` ⟨*argument*⟩`: {`⟨*option one, option two, option three, ...*⟩`}`

The vast majority of the ⟨*options*⟩ can be passed into the ⟨*input file*⟩. These should be put at the beginning of the file in commented lines and everything must be on the same line, the exclamation mark `!` deactivates the ⟨*options*⟩. When passing options from the ⟨*input file*⟩ you should be aware that they must *"not"* contain `-` or `--`, the `=` sign between an option and its value is mandatory, short names are disabled and options found in the ⟨*input file*⟩ overwrite those passed on the command line.

Valid values for ⟨*argument*⟩ are the following:

`% ltximg: options: {`⟨*option one = value, option two = value, option three = value, ...*⟩`}`

This line is to indicate to the script which options need to process.

`% ltximg: extrenv: {`⟨*environment one, environment two, environment three, ...*⟩`}`

This line is to indicate to the script which environments, not supported by default, are extracted.

`% ltximg: skipenv: {`⟨*environment one, environment two, environment three, ...*⟩`}`

This line is to indicate to the script which environments, of the ones supported by default, should not be extracted.

[7] https://ctan.org/pkg/arara

`% ltximg: verbenv: {`⟨*environment one, environment two, environment three, ...*⟩`}`

This line is to indicate to the script which environments, its considerate a ⟨*verbatim standard*⟩.

`% ltximg: writenv: {`⟨*environment one, environment two, environment three, ...*⟩`}`

This line is to indicate to the script which environments its consider ⟨*verbatim write*⟩.

`% ltximg: deltenv: {`⟨*environment one, environment two, environment three, ...*⟩`}`

This line is to indicate to the script which environments are deleted.

If you are going to create an ⟨*output file*⟩ and you do not want these lines to remain, it is better to place them inside the `%<*remove> ... %</remove>`. Like this:

```
%<*remove>
% ltximg: options: {png,srcenv,xetex}
% ltximg: extrenv: {description}
%</remove>
```

11 The way of arara and dvisvgm

By design, the script only runs *"one compilation"* on top of the ⟨*input file*⟩, but, sometimes you need to process the ⟨*input file*⟩ more than once or needs to be processed with something other than LaTeX, XeLaTeX, pdfLaTeX or LuaLaTeX engine.

This is where arara[18] comes in, this *"great little tool"*, is able to have complete control over the compilation of the ⟨*input file*⟩, we just have to keep a few considerations in mind:

1. Read the documentation (this always comes first).
2. Add `{ options: [-recorder] }` to "rule" for clean temporary files.
3. Avoiding the use of `clean: { extensions: [...] }`.
4. Don't set `-jobname` and `-output-directory`.
5. Remember that the ⟨*input file*⟩ and ⟨*output file*⟩ will be compiled using the same "rule".

With all these considerations in mind it is possible to extract and convert from any file, for example, by adding these lines:

```
% arara: lualatex: { options: [-recorder] }
% arara: lualatex: { options: [-recorder] }
%<*remove>
% ltximg: options: {arara, output = file-out, noprew, prefix = tkz}
%</remove>
```

and run:

```
[user@machine~:]$ ltximg test.tex
```

Create a `./images` directory (if it does not exist) whit all `tikzpicture` environments converted to image format (`pdf`) in individual files, an ⟨*output file*⟩ `file-out.tex` whit all `tikzpicture` environments converted to `\includegraphics`, a single file `test-tkz-all.tex` with only the extracted environments using `lualatex` two times for ⟨*input file*⟩ and ⟨*output file*⟩.

In this way we can ⟨*compile*⟩ and ⟨*convert*⟩ any document as long as the conditions of the ⟨*input file*⟩ are met and the correct "rule" are used,

Also, by design, the image format `svg` is created using `pdftocairo` over the generated `pdf` file, but, if you want to have a good `svg` file it is best to use `dvisvgm`[8] which is included in every modern TeX distribution. The best results of `dvisvgm`[19] are obtained when processing the file in `.dvi` or `.xdv` format, there are three possible ways to do this:

1. Pass the necessary options to `arara` and let him do the job.
2. Execute the script using `--subenv` and `--norun`, move to `images/` and generate `.dvi`(`.xdv`) files, then:

   ```
   [user@machine~:]$ for i in *.dvi; do dvisvgm [⟨options⟩] $i;done
   ```

3. Execute the script using `--norun`, move to `images/` and generate `test-fig-all.dvi`(`.xdv`), then:

   ```
   [user@machine~:]$ dvisvgm [⟨options⟩] test-fig-all.dvi
   ```

With this we can generate `svg` files that preserve our typographic fonts.

[8] https://ctan.org/pkg/dvisvgm

12 Examples

12.1 Passing command line options

```
[user@machine ~:]$ ltximg --latex -s -o test-out  test-in.ltx
```

Create a `./images` directory (if it does not exist) whit all extracted environments converted to image formats (`pdf`, `svg`) in individual files, an ⟨*output file*⟩ `test-out.ltx` whit all extracted environments converted to `\includegraphics` and a single file `test-in-fig-all.ltx` with only the extracted environments using `latex»dvips»ps2pdf` and `preview` package for ⟨*input file*⟩ and `pdflatex` for ⟨*output file*⟩.

12.2 Passing options from input file

Adding the following lines to the beginning of the file `file-in.tex`:

```
% ltximg: options: {luatex, output = file-out, imgdir = pics, prefix = env}
% ltximg: skipenv: {tikzpicture}
% ltximg: deltenv: {filecontents}
```

and run:

```
[user@machine~:]$ ltximg file-in.tex
```

Create a `./pics` directory (if it does not exist) whit all extracted environments, except `tikzpicture`, converted to image formats (`pdf`) in individual files, an ⟨*output file*⟩ `file-out.tex` whit all extracted environments converted to `\includegraphics` and environment `filecontents` removed, a single file `test-in-env-all.ltx` with only the extracted environments using `lualatex` and `preview` package for ⟨*input file*⟩ and ⟨*output file*⟩.

12.3 Using `arara` and `dvisvgm`

The process and operations required to generate the various types of ⟨*image formats*⟩ or ⟨*standalone files*⟩ have been described throughout the documentation, but, as discussed in section 11, sometimes the requirements are a *little different*. Let's take as an example[9] the next code which requires two compilations and use `arara` to generate an image in `pdf` format. We'll save our example as `test.tex`.

```
% arara: lualatex: { options: [-recorder] }
% arara: lualatex: { options: [-recorder] }
\documentclass{article}
\usepackage[osf]{libertinus}
\usepackage{tikz}
\usetikzlibrary{calc,tikzmark}
\begin{document}

By taking logarithms of both sides:

%<*ltximg>
\[
  t = \frac{30\cdot\ln(3/22)}{\ln(15/22)}
  \tikzmark{calculator}\approx\tikzmark{otherside}
    156
\]
\begin{tikzpicture}[overlay,remember picture]
  \coordinate (target) at ($(pic cs:calculator)!1/2!(pic cs:otherside) - (0,.5ex)$);
  \draw[arrows=->] (target) ++(0,-2ex) node [anchor=north] {use calculator} -- (target);
\end{tikzpicture}
%</ltximg>
\end{document}
```

Now we just run:

```
[user@machine~:]$ ltximg --arara test.tex
```

And we already have our image `test-fig-1.pdf` on `./images`.

[9] Adapted from How to get tikzmark to work

Let's modify the example[10] to generate an image in svg format using dvisvgm and arara.

```
% arara: lualatex: { options: [--output-format=dvi] }
% arara: dvisvgm: { options: [--exact-bbox, -o test-fig-1.svg] }
% ltximg: extrenv: {picture}
% ltximg: options: {arara,norun,noprew}
\documentclass{article}
\begin{document}

The best airplane ever drawn by David Carlise. No packages used, just the classic and perhaps
forgotten \verb|\begin{picture} ... \end{picture}|.

\begin{picture}(200,100)
  \put(30,40){\line(1,0){150}}
  \put(30,40){\line(0,1){60}}
  \put(30,100){\line(1,0){20}}
  \put(50,100){\line(1,-4){10}}
  \put(60,60){\line(1,0){100}}
  \put(160,60){\line(1,-1){20}}
  \put(100,50){\line(0,-1){80}}
  \put(130,50){\line(0,-1){80}}
  \put(100,-30){\line(1,0){30}}
  \put(100,61){\line(0,1){49}}
  \put(130,61){\line(0,1){49}}
  \put(100,110){\line(1,0){30}}
\end{picture}
\end{document}
```

We now run:

```
[user@machine~:]$ ltximg test.tex
[user@machine~:]$ cd images/
[user@machine~:]$ arara test-fig-all.tex
```

And we already have our image test-fig-1.svg.

Here are some of the ways you can use **ltximg** in conjunction with arara. If you are a user of latexmk[11], another great utility that automates the compilation process, the procedure is analogous, you just have to search (or write) the necessary *"rule"*.

[10] Adapted from Draw an aircraft with Tikz

[11] https://www.ctan.org/pkg/latexmk

13 Change history

The most recent publicly released version of **ltximg** is available at CTAN: https://www.ctan.org/pkg/ltximg. Historical and developmental versions are available at https://github.com/pablgonz/ltximg.

While general feedback via email is welcomed, specific bugs or feature requests should be reported through the issue tracker: https://github.com/pablgonz/ltximg/issues.

This is a short list of some of the notable changes in the history of the **ltximg** along with the versions, both development (devp) and public (ctan).

v1.8 (ctan), 2020-05-23
- It is now possible to extract any environment.
- Fix `verbose` option.
- Add `log` option.
- All calls to the system are captured.
- Re-write source code acording to `perl` v5.3x.
- Review of documentation.

v1.7 (ctan), 2019-08-24
- Add `scontents` environment support.
- Add `filecontentsdefmacro` environment support.
- Fix regex in source code.
- Update documentation.

v1.6 (ctan), 2019-07-13
- Add `zip` and `tar` options.
- Add new `Verb` from `fvextra`.
- Fix and update source code and documentation.

v1.5 (ctan), 2018-04-12
- Use `GitHub` to control version.
- Rewrite and optimize most part of source code and options.
- Change `pdf2svg` for `pdftocairo`.
- Complete support for `pst-exa` package.
- Escape characters in regex according to `perl` v5.2x.

v1.4 (devp), 2016-11-29
- Remove and rewrite code for regex and system call.
- Add `arara` compiler, clean and comment code.
- Add `dvips` and `dvipdfm(x)` for creation images.
- Add `bmp`, `tiff` image format.

v1.3 (devp), 2016-08-14
- Rewrite some part of code (norun, nocrop, clean).
- Suport `minted` and `tcolorbox` package.
- Escape some characters in regex according to `perl` v5.2x.
- All options read from command line and input file.
- Use `/tmp` dir for work process.

v1.2 (ctan), 2015-04-22
- Remove unused modules.
- Add more image format.
- Fix regex.

v1.1 (ctan), 2015-04-21
- Change `mogrify` to `gs` for image formats.
- Create output file.
- Rewrite source code and fix regex.
- Change format date to iso format.

v1.0 (ctan), 2013-12-01
- First public release.

14 References

[1] Kastrup, David. "The preview package for LaTeX". Available from CTAN, https://www.ctan.org/pkg/preview, 2017.

[2] Tantau, Till. "The TikZ and PGF Packages". Available from CTAN, https://www.ctan.org/pkg/pgf, 2020.

[3] Van Zandt, Timothy. "PSTricks - PostScript macros for generic TeX". Available from CTAN, https://www.ctan.org/pkg/pstricks-base, 2007.

[4] Van Zandt, Timothy. "pst-plot – Plot data using PSTricks". Available from CTAN, https://www.ctan.org/pkg/pst-plot, 2019.

[5] Niepraschk, Rolf. "The pst-pdf Packages". Available from CTAN, https://www.ctan.org/pkg/pst-pdf, 2019.

[6] Robertson, Will. "The auto-pst-pdf Packages". Available from CTAN, https://www.ctan.org/pkg/auto-pst-pdf, 2009.

[7] Voß, Herbert. "pst-exa - Typeset PSTricks examples, with pdfTeX". Available from CTAN, https://www.ctan.org/pkg/pst-exa, 2017.

[8] Voß, Herbert. "pst2pdf - A script to compile PSTricks documents via pdfTeX". Available from CTAN, https://www.ctan.org/pkg/pst2pdf, 2017.

[9] The LaTeX3 Project. "graphics – Enhanced support for graphics". Available from CTAN, https://www.ctan.org/pkg/graphicx, 2017.

[10] Oberdiek, Heiko. "The grfext package". Available from CTAN, https://www.ctan.org/pkg/grfext, 2019.

[11] Van Zandt, Timothy. "The xcomment package". Available from CTAN, https://www.ctan.org/pkg/xcomment, 2010.

[12] Adriaens, Hendri. "The extract package". Available from CTAN, https://www.ctan.org/pkg/extract, 2019.

[13] Trzeciak, Tomasz M. "The cachepic package". Available from CTAN, https://www.ctan.org/pkg/cachepic, 2009.

[14] Mittelbach, Frank. "The doc and shortvrb Packages". Available from CTAN, https://www.ctan.org/pkg/doc, 2020.

[15] Van Zandt, Timothy. "The fancyvrb package - Fancy Verbatims in LaTeX". Available from CTAN, https://www.ctan.org/pkg/fancyvrb, 2020.

[16] Hoffmann, Jobst. "The listings package". Available from CTAN, https://www.ctan.org/pkg/listings, 2020.

[17] Poore, Geoffrey M. "The minted package - Highlighted source code in LaTeX". Available from CTAN, https://www.ctan.org/pkg/minted, 2017.

[18] The Island of TeX. "arara - The cool TeX automation tool". Available from CTAN, https://www.ctan.org/pkg/arara, 2020.

[19] Gieseking, Martin. "dvisvgm - A fast DVI to SVG converter". Available from CTAN, https://ctan.org/pkg/dvisvgm, 2020.

15 Index of Documentation

The italic numbers denote the pages where the corresponding entry is described.