

{robust-externalize}

Cache anything (*TikZ*, *python*...),
in a robust, efficient and pure way.

Léo Colisson Version 2023/03/22-unstable

github.com/leo-colisson/robust-externalize

Contents

1	Introduction	1
1.1	Why do I need to cache (a.k.a. externalize) parts of my document?	1
1.2	Why not using <i>TikZ</i> 's externalize library?	2
1.3	FAQ	2
2	Quickstart	3
2.1	Installation	3
2.2	Usage	3
2.2.1	For \LaTeX based content	3
2.3	For non- \LaTeX code	5
2.4	Example of more advanced setup	7
3	Manual	9
4	Operations on the cache	9
4.1	Cleaning the cache	9
4.2	Listing all figures in use	10
4.3	Manually compiling the figures	10
5	TODO and known bugs:	10

WARNING: This library is very young and has not been tested extensively. Even if we try to stay backward compatible, the only guaranteed way to be immune to changes is to copy/paste the library in your main project folder.

1 Introduction

1.1 Why do I need to cache (a.k.a. externalize) parts of my document?

One often wants to cache (i.e. store pre-compiled parts of the document, like figures) operations that are long to do: For instance, *TikZ* is great, but *TikZ* figures often takes time to compile (it can easily take a few seconds per picture). This can become really annoying with documents containing many pictures, as the compilation can take multiple minutes: for instance my thesis needed roughly 30mn to compile as it contains many tiny figures, and \LaTeX needs to compile the document multiple times before converging to the final result. But even on much smaller documents you can easily reach a few minutes of compilation, which is not only high to get a useful feedback in real time, but worse, when using online \LaTeX providers (e.g. *overleaf*), this can be a real pain as you are unable to process your document due to timeouts.

Similarly, you might want to cache the result of some codes, for instance a text or an image generated via *python* and *matplotlib*, without manually compiling them externally.

1.2 Why not using TikZ’s externalize library?

TikZ has an externalize library to pre-compile these images on the first run. Even if this library is quite simple to use, it has multiple issues:

- If you add a picture before existing pre-compiled pictures, the pictures that are placed after will be recompiled from scratch. This can be mitigated by manually adding a different prefix to each picture, but it is highly not practical to use.
- To compile each picture, TikZ’s externalize library reads the document’s preamble and needs to process (quickly) the whole document. In large documents (or in documents relying on many packages), this can result in a significant loading time, sometimes much bigger than the time to compile the document without the externalize library: for instance, if the document takes 10 seconds to be processed, and if you have 200 pictures that take 1s each to be compiled, the first compilation with the TikZ’s externalize library will take roughly half an hour instead of 3mn without the library. And if you add a single picture at the beginning of the document... you need to restart everything from scratch. For these reasons, I was not even able to compile my thesis with TikZ’s external library in a reasonable time.
- If two pictures share the same code, it will be compiled twice
- Little purity is enforced: if a macro changes before a pre-compiled picture that uses this macro, the figure will not be updated. This can result in different documents depending on whether the cache is cleared or not.
- As far as I know, it is made for TikZ picture mostly, and is not really made for inserting other stuff, like matplotlib images generated from python etc...
- According to some maintainers of TikZ, “the code of the externalization library is mostly unreadable gibberish¹”, and therefore most of the above issues are unlikely to be solved in a foreseeable future.

1.3 FAQ

Do you need to compile using `-shell-escape`? Since we need to compile the images via an external command, the simpler option is to add the argument `-shell-escape` to let the library run the compilation command automatically (this is also the case of TikZ’s externalize library). However, people worried by security issues of `-shell-escape` (that allows arbitrary code execution if you don’t trust the L^AT_EX code) might be interested by these facts:

- If images are all already cached, you don’t need to enable `-shell-escape`.
- You can choose not to compile non-cached content, and display a dummy content instead until you choose to compile them.
- You can compile manually the images: all the commands that are left to be run are listed in `robExt-compile-missing-figures.sh` and you can just run them, either with `bash robExt-compile-missing-figures.sh` or by typing them manually (most of the time it’s only a matter of running `pdflatex somefile.tex`).

Is it working on overleaf? Yes: overleaf automatically compiles documents with `-shell-escape`, so nothing special needs to be done there (of course, if you use this library to run some code, the programming language might not be available, but I heard that python is installed on overleaf servers for instance, even if this needs to be doubled checked). If the first compilation of the document to cache images times out, you can just repeat this operation multiple times until all images are cached.

¹<https://github.com/pgf-tikz/pgf/issues/758>

Do you have some benchmarks? On an early draft of a small paper containing 76 small tikz-cd based pictures (from my other zx-calculus library), we measured:

- 35 seconds for a normal compilation without externalization
- 75 seconds for the first compilation with this library
- 2.4 seconds for the next runs

So during the first compilation, we lost a x2 factor (roughly an additional time of .5 seconds per picture coming from the time to start L^AT_EX, it seems like on average a picture takes .5 seconds to be built in my benchmark), but then we have a speedup of x15 (2.43s instead of 34.63s) for all subsequent runs. And I expect this to be even higher with more pictures and more complex documents.

Can you deal with baseline position? Yes, the depth of the box is automatically computed and used to include the figure by default.

How is purity enforced? Purity is the property that if you remove the cached files and re-compile your document, you should end-up with the same output. To enforce purity, we compute the hash of the final program, including the compilation command and the dependency files used for instance in `\input{include.tex}` (unless you prefer not to, for instance to keep parts of the process impure for efficiency reasons), and put the code in a file named based on this hash. Then we compile it if it has not been used before, and include the output. Changing a single character in the file, the tracked dependencies, or the compilation command will lead to a new hash, and therefore to a new generated picture.

Can I extend it easily? We tried to take a quite modular approach in order to allow easy extensions. Internally, to support a new cache scheme, we only expect a string containing the program (possibly produced using a template), a list of dependencies, a command to compile this program (e.g. producing a pdf and possibly a tex file with the properties (depth...) of the pdf), and a command to load the result of the compilation into the final document (called after loading the previously mentioned optional tex file). Thanks to pgfkeys, it is then possible to create simple pre-made settings to automatically apply when needed.

2 Quickstart

2.1 Installation

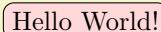
To install the library, just copy the `robust-externalize.sty` file into the root of the project. Then, load the library using:

```
\usepackage{robust-externalize}
```

2.2 Usage

2.2.1 For L^AT_EX based content

In theory, if you only care about TikZ's picture, you can just do:



Hello World!

```
%% We override the default tikzpicture environment
%% to externalize all pictures
\robExtExternalizeAllTikzpictures

\begin{tikzpicture}[baseline,anchor=base]
  \node[draw,rounded corners,fill=pink!60]{Hello World!};
\end{tikzpicture}
```

and all tikzpictures created using `\begin{tikzpicture}`... will be “externalized”. However, by default the \LaTeX template used to compile these pictures is empty, so you will quickly want to populate the template (for instance to load packages, define custom macros possibly shared with the main document etc.). So let’s step back and see how we can define an arbitrary template.

If you want to compile a \LaTeX code, say a TikZ picture, you first need to define the \LaTeX template that will wrap all your code². Because you might want to mix different templates in a document (e.g. for tikz pictures, matplotlib python code, tikz-cd or zx diagrams...), we like to define them into a preset that is just a set of configuration options³. For instance to create a basic preset called `presetTikz`, use:

```
\robExtConfigure{%
  presetTikz/.style={
    % We define the code that wraps all our figures
    defineTemplate={
      \documentclass{standalone} % standalone ensures that the pdf output size matches the content
      \usepackage{tikz} % Loads any package you need to compile your pictures
      % You can define macros, just make sure to double the number of sharps
      % as otherwise they will be understood as options of the preset.
      \def\sayHello##1{Hello ##1}
      \input{input_externalize.tex} % you can put in this file regular LaTeX code to share with the main document
      \begin{document}%
      \robExtMainContent % This macro will be replaced with (notably) the code for the figure
      \end{document}
    },
    % The dependency files needed to compile the file
    % (included when computing the hash). Separate them with commas.
    dependencies={input_externalize.tex},
  },
}
```

(see the comments for details)

Then, if needed, create the dependency files, and use this preset as follows in your code:

See that the baseline is respected Hello World!

```
See that the baseline is respected %
\begin{robExtern}{presetTikz}%
  \begin{tikzpicture}[baseline,anchor=base]%
    \node[draw,rounded corners,fill=pink!60]{\sayHello{World}!};
  \end{tikzpicture}
\end{robExtern}
```

Note that one might be tempted to move `\begin{tikzpicture}` inside the template to save a bit of typing. However, our library is actually replacing `\robExtMainContent` with the content wrapped around some code⁴ that need to wrap the whole content. While we provide other macros that don’t add this wrapping code⁵, it is actually simpler, more configurable, and more typing-friendly, to define an environment that automatically picks the right preset and adds `\begin{tikzpicture}` automatically for us:

²While you could use the same preamble as the main project, for instance using a shared `\input{input.tex}` file, this is not recommended as it will not only be longer to load (some packages are useless to build your average tikzpictures), but it also harms purity or efficiency: if you choose to track this common input (i.e. dependency), then it will recompile the pictures every time you change `input.tex`, and if you don’t track this dependency, then you might skip a needed recompilation leading to a different outcome after invalidating the cache.

³Internally this is just a pgfkeys style, if you don’t want to define presets, you can just write the configuration outside of the preset... but this is not recommended, except for some specific configuration options like `disable externalization` that you may want to apply globally.

⁴Basically creating a box in order to compute the depth of the content, and write it to a tex file to use for later.

⁵But then we need to manually add this box if we want to compute the appropriate depth, and we will need to redefine the command to disable externalization.

```

\DeclareDocumentEnvironment{mytikzpicture}{0}{0}{b}{% = 2 optional arguments + the body (b), cf xparse
\begin{robExtern}{presetTikz,#2}%
\begin{tikzpicture}[#1]%
#3
\end{tikzpicture}%
\end{robExtern}%
}{}

```

This way, in your code you can just use:

See the respected baseline: Hello World!

```

See the respected baseline: %
\begin{mytikzpicture}[baseline,anchor=base]
\node[draw,rounded corners,fill=pink!60]{\sayHello{World}!};
\end{mytikzpicture}

```

By choosing the name `tikzpicture` instead of `mytikzpicture`, you would actually override TikZ's macro, which should be perfectly fine if you want to externalize TikZ's pictures by default. Because this usecase will likely be important, we actually provide a command that defines a similar environment, except that it uses a preset called `presetTikzDefault` (this style is populated with a very simple template, but you surely want to quickly override it with your own template by just creating a new preset called `presetTikzDefault`):

Hello World!

```

%% We override the default tikzpicture environment:
\robExtExternalizeAllTikzpictures

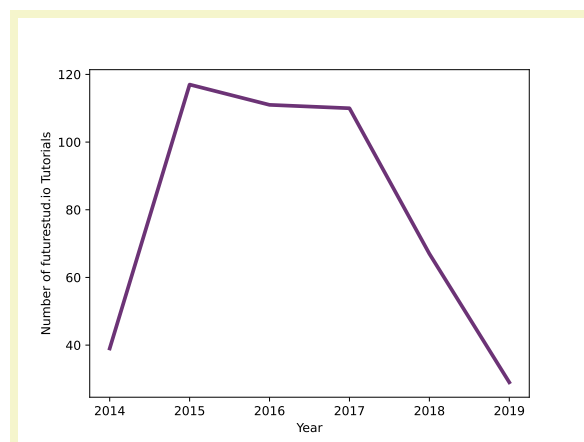
\begin{tikzpicture}[baseline,anchor=base]
\node[draw,rounded corners,fill=pink!60]{Hello World!};
\end{tikzpicture}

```

2.3 For non- \LaTeX code

Due to the way \LaTeX works, non- \LaTeX code can't be reliably read inside macros and some environments (e.g. `align`) as some characters are removed (e.g. percent symbol). For this reason, we sometimes need to separate the time where we define the code and where we insert it, and we provide therefore different commands to deal with non- \LaTeX code.

The following code will name a template `pythonMatplotlib` (see how we use `ROBEXTMAINCONTENT` as a placeholder for the content), define a preset based on this template, and :



```

% We define our python template:
\begin{robExtNamedTemplate}[pythonMatplotlib]
import matplotlib.pyplot as plt
import sys
ROBEXTMAINCONTENT
plt.savefig(sys.argv[1]+".pdf")
\end{robExtNamedTemplate}

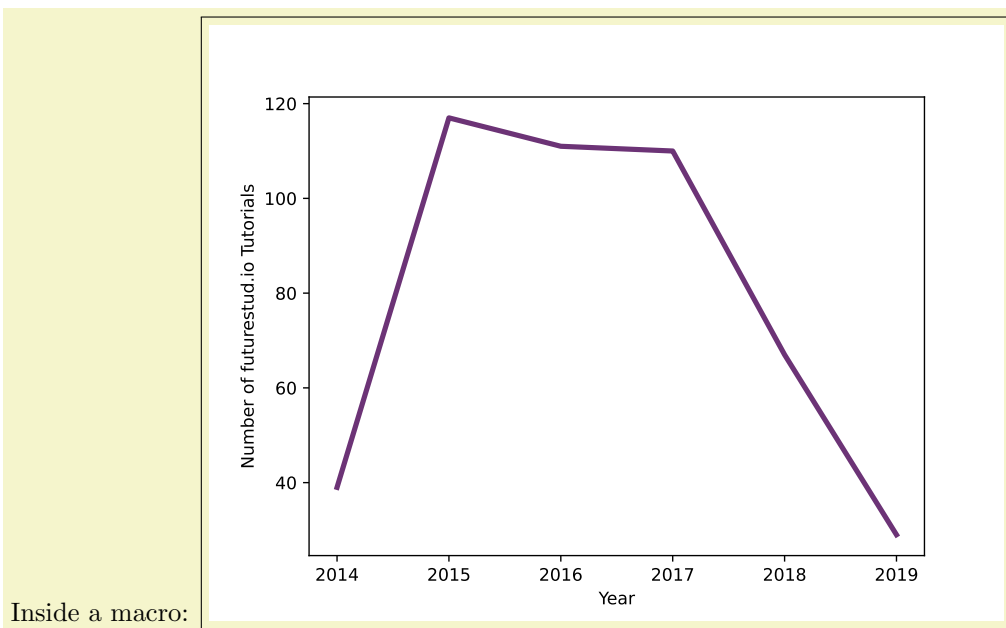
% We define a reusable preset, note that we specify the compilation command:
\robExtConfigure{
  presetMatplotlib/.style={
    defineTemplateFromName=pythonMatplotlib,
    set compilation command={python3 "\robExtFinalFile" "\robExtFinalPrefixedName"},
  },
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Above code must be written once, below is used for any drawing

% We draw our code:
\begin{robExtCode}{presetMatplotlib,include graphics args={width=.5\linewidth}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{robExtCode}

```

Note that for the fundamental reasons mentioned above, the above code can't work inside a macro. If we still want to include the result in a macro, we can separate the definition of the code and its usage:



```

% We define our python template:
\begin{robExtNamedTemplate}[pythonMatplotlib]
import matplotlib.pyplot as plt
import sys
ROBEXTMAINCONTENT
plt.savefig(sys.argv[1]+".pdf")
\end{robExtNamedTemplate}

% We define a reusable preset, note that we specify the compilation command:
\robExtConfigure{
  presetMatplot/.style={
    defineTemplateFromName=pythonMatplotlib,
    set compilation command={python3 "\robExtFinalFile" "\robExtFinalPrefixedName"},
  },
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Above code must be written once, below is used for any drawing

% We define our code (implicitly giving it a default name, that can be changed
% if multiple codes are inserted in the same macro)
\begin{robExtNamedContent}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]

plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{robExtNamedContent}

Inside a macro: %
\fbbox{\robExternPrev{presetMatplot,include graphics args={width=.7\linewidth}}}}

```

2.4 Example of more advanced setup

We can actually do many more things. For instance, here we define another preset that compiles the document with python and matplotlib, and displays the important lines of the code above the result, inside a figure with a customizable caption:

```
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]

plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
```

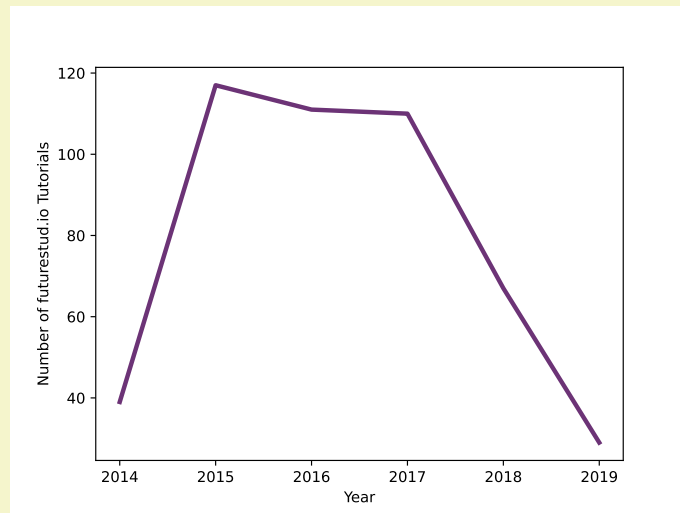


Figure 1: Here is my caption for the figure


```

% Define the template. Lines with TEMPLATECODE
% will be removed later.
\begin{robExtNamedTemplate}[pythonMatplotlib]
import matplotlib.pyplot as plt # TEMPLATECODE
import sys # TEMPLATECODE
ROBEXTMAINCONTENT
plt.savefig(sys.argv[1]+".pdf") # TEMPLATECODE
\end{robExtNamedTemplate}

\robExtConfigure{
  % More complex version that displays both the code and the result:
  presetMatplotAdvanced/.style={
    /robExt/caption/.code={\gdef\mycaption{##1}}, % Provide a key to change the caption
    defineTemplateFromName=pythonMatplotlib,
    % This command compiles the image, and creates a file removing all the lines from the code
    % containing TEMPLATECODE (useful not to display action of saving the file etc)
    % This might not be portable to windows without installing cygwin, but one can replace sed
    % with some python code doing the same for a more portable code.
    set compilation command={python3 "\robExtFinalFile" "\robExtFinalPrefixedName" %
      && sed '/TEMPLATECODE/d' "\robExtFinalFile" > "\robExtFinalPrefixedName.codeonly.py"},
    custom include command={%
      \begin{figure}[H] % Use H mostly to avoid compilation error in documentation
        \centering
        % Note that this will display the template around the code. We could avoid this by putting in our
        % template a part that creates a new file whose name is the basename of the current (script) file
        % and that contains all its code except for the template (to differentiate between template and
        % non-template, we could add a special comment on lines to remove)
        \verbatiminput{\robExtAddPrefixPathAndName{\robExtFinalName.codeonly.py}}
        \includegraphics[width=.6\textwidth]{\robExtAddPrefixPathAndName{\robExtFinalName.pdf}}%
        \caption{\mycaption}
      \end{figure}
    },
  },
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Above code must be written once, below is used for any drawing

\begin{robExtCode}{presetMatplotAdvanced, caption={Here is my caption for the figure}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]

plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{robExtCode}

```

3 Manual

TODO: the example above already provide a nice view of the main functions, the `set subfolder={robustExternalize}` option might be an important additionnal configuration option that allows you to put the cached pictures in a subfolder.

4 Operations on the cache

4.1 Cleaning the cache

You might want to clean the cache. Of course you can remove all generated files, but if you want to keep the picture in use in the latest version of the document, we provide a python script (automatically generated in the root folder) to do this. Just install python3 and run:

```
python3 robExt-remove-old-figures.py
```

You will then be prompted for a confirmation after providing the list of files that will be removed.

4.2 Listing all figures in use

After the compilation of the document, a file `robExt-all-figures.txt` is created with the list of the `.tex` file of all figures used in the current document.

4.3 Manually compiling the figures

When enabling the manual mode (useful if we don't want to enable `-shell-escape`):

```
\robExtConfigure{
  enable manual mode
}
```

the library creates a file `robExt-compile-missing-figures.sh` that contains the instructions to build the figures that are not yet in the cache. On Linux (or on Windows with bash/cygwin/... installed) you can easily execute them using:

```
bash robExt-compile-missing-figures.sh
```

5 TODO and known bugs:

- Solve problem with disable externalization not working with tikz pictures
- We should create more pre-made settings, e.g. for tikz-cd, zx-calculus etc.
- Some commands like mkdir might not be super compatible with Windows, I need to see how to improve compatibility between OS
- The documentation is still sparse.
- For now we put `compile-missing-figures.sh` in subfolders