The **pkgloader** and **lt3graph** packages: Toward simple and powerful package management for LaTeX

Michiel Helvensteijn

Abstract

This article introduces the `pkgloader` package. I recently wrote this package to address one of the major frustrations of LaTeX: package conflicts. It also introduces `lt3graph`, a LaTeX3 library used by `pkgloader` to do most of the heavy lifting.

1 Introduction

LaTeX package conflicts are a common source of frustration. If you are reading this article, you're probably experienced enough with LaTeX to have encountered them more than once. I cannot improve upon the words of Freek Dijkstra [3] on the subject:

"Package conflicts are a hell."

Package conflicts can exist because of the sheer power of TeX [4], the language on which LaTeX is based. Not only is it Turing complete [8], but most of the language can be redefined from within the language itself. This was famously demonstrated with the TeX script `xii.tex`, written by David Carlisle [1] (if you haven't seen it yet, download and compile it now; it's awesome). LaTeX packages can not only add new definitions, but also remove and modify existing ones. They can offer domain specific languages [6], monkey-patch the core language to hook into existing commands [7], and even change the meaning of individual symbols by altering their category code [5]. Put simply, LaTeX packages have free rein. This power can be quite useful, but makes it too easy for independent package authors to step on each others' toes.

A different type of conflict concerns package options. If a package is requested more than once with different options, LaTeX bails out with an error message. This is an understandable precaution. Because of the way package loading works, LaTeX has no way to apply the second set of options. The package will have already been loaded with the first set.

Most package authors are well aware of these problems. Document authors are told to avoid certain package combinations, or to load packages in some specific order. Some of the larger packages are designed to test for the presence of other packages in order to circumvent known conflicts. Unfortunately, this is all done in an ad hoc fashion.

Solving these problems on a case-by-case basis takes time and effort for both document and package authors. It pollutes the code, makes maintenance more difficult, and confuses new users. We need a systematic approach to resolve package conflicts. This is where the power of TeX comes in handy. A package can be written to oversee the package loading process: a LaTeX package manager. It should be easy enough to use for the casual document author, yet powerful enough to allow package authors to hook into it to simplify their development process.

Section 2 introduces `pkgloader`, which I wrote to fulfill this rôle. Reading this section should be enough to give you a general idea of how to use it. Section 3 describes `lt3graph`, a utility library using the LaTeX3 programming layer which does most of the heavy lifting for `pkgloader`. Finally, Section 4 goes through some of the more advanced and planned features of `pkgloader`.

Here is a quick glimpse of `pkgloader` in use:

```
\RequirePackage{pkgloader}
   ...
   \documentclass{article}
   ...
   \usepackage{algorithm}  ⎫
   \usepackage{hyperref}   ⎬ any order
   \usepackage{float}      ⎭
   ...
\begin{document}
   ...
\end{document}
```

Warning: This package is still under development. Some of the features described in this article may not yet be fully implemented, and the presented syntax may still change in the coming months. However, its main purpose and the fundamental ideas underlying its implementation are here to stay.

Community collaboration: I intend for the development and maintenance of this package to be as open as possible to community collaboration. This package has a very wide scope, and is rather invasive. If done right, it has the potential to become widely used and improve the LaTeX experience for document authors and package authors alike. If done wrong, it will break things and annoy many people.

I hope to bring some useful domain knowledge to the development effort, but there are many LaTeX gurus out there who have more experience and insight than I do. If you like the idea of `pkgloader` and would like to contribute in any way, I encourage you to contact me personally, or to file issues or pull requests through the `pkgloader` Github page: `github.com/mhelvens/latex-pkgloader`

2 The **pkgloader** package (Part 1)

This package was inspired by my PhD research [2], which happens to be all about conflicts between independently developed modules. I also took cues from similar libraries and standards for other languages, such as JavaScript, which is surprisingly similar to LaTeX in many ways.

2.1 How to use the package manager

LaTeX packages are generally loaded with one of the commands **\usepackage**, **\RequirePackage**, or **\RequirePackageWithOptions**. In a similar way, document classes are loaded with **\documentclass**, **\LoadClass** or **\LoadClassWithOptions**. Normally, when such a command is reached, the relevant class or package is loaded on the spot. The idea behind `pkgloader` is to make it the very first file you load: before the document class, and before any other package. Thus, the main file for a LaTeX document using `pkgloader` would be structured like this:

```
\RequirePackage{pkgloader}
   ⟨document class and packages in any order⟩
\LoadPackagesNow                 }
...                              } optional
\begin{document}
   ...
\end{document}
```

The area between **\RequirePackage**{pkgloader} and **\LoadPackagesNow** is called the *pkgloader area.* Inside this area, the loading of all classes and packages is postponed. It also ends automatically upon reaching the end of the preamble. The package manager analyzes the intercepted loading requests and executes them in the proper order and with the proper options, lifting this burden from the user.

2.2 A conflict resolution database

The `pkgloader` package does *not* analyze the actual code of each package in order to detect conflicts. In fact, because TeX is Turing complete, this would be mathematically impossible. The package manager is backed by a database of *rules* for recognizing and resolving known conflicts, as well as performing other neat tricks. The following shows some examples:

```
\Load {float} before {hyperref}
\Load {algorithm} after {hyperref}
\Load {fixltx2e} always early
      because {it fixes some imperfections
               in LaTeX2e}
\Load error if {algorithms && pseudocode}
      because {they provide the same
               functionality and conflict
               on many command names}
```

The first two rules encode some workarounds for the `hyperref` package, which is notorious for causing conflicts. The first one says that `float` must be loaded before `hyperref`. Similarly, the second rule ensures that `hyperref` is loaded before `algorithm`. These are the rules that would allow the code on page 39 to compile without problems. Note that neither rule actually loads any packages. They simply tell the package manager how to treat certain pairs of packages, should they ever be requested together in a single document.

The third rule states that `fixltx2e` must always be loaded, and must be loaded early. The fourth rule states that the `algorithms` and `pseudocode` packages should never be loaded together. They both also include a textual reason, which documents the rule, and is included in certain error messages.

To better understand how these rules work, let's dive into their underlying model: a directed graph.

3 The `lt3graph` Package

The `pkgloader` package is written in the experimental LaTeX3 programming layer `expl3`, which gives us something akin to a traditional imperative programming language, with data structures, `while` loops, and so on. Let's face it, TeX is no one's first choice for a programming language. But `expl3` makes it bearable. So kudos to the LaTeX3 team!

To represent graphs, I wrote a data-structure package called `lt3graph`.[1] This library was born as a means to an end, but has grown into a full-fledged general-purpose data structure for representing and analyzing directed graphs.

A directed graph contains vertices (nodes) and edges (arrows). Using `lt3graph`, an example graph may be defined as follows:

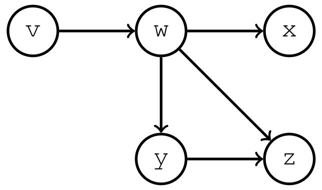
```
\ExplSyntaxOn
\graph_new:N    \l_my_graph
\graph_put_vertex:Nn \l_my_graph {v}
\graph_put_vertex:Nn \l_my_graph {w}
\graph_put_vertex:Nn \l_my_graph {x}
\graph_put_vertex:Nn \l_my_graph {y}
\graph_put_vertex:Nn \l_my_graph {z}
\graph_put_edge:Nnn \l_my_graph {v} {w}
\graph_put_edge:Nnn \l_my_graph {w} {x}
\graph_put_edge:Nnn \l_my_graph {w} {y}
\graph_put_edge:Nnn \l_my_graph {w} {z}
\graph_put_edge:Nnn \l_my_graph {y} {z}
\ExplSyntaxOff
```

Each vertex is identified by a *key*, which, to this library, is a string: a list of characters with category code 12 and spaces with category code 10. An edge is then declared between two vertices by referring to their keys. By supplying an additional argument to the functions above, you can store arbitrary data in

[1] Bearing the prefix `lt3` rather than the more common prefix `l3` indicates that the package is not officially supported by the LaTeX3 team.

a vertex or edge for later retrieval. Let's use Ti*k*Z to visualize this graph:

```
\newcommand{\vrt}[1]
  {\node(#1){\ttfamily\vphantom{Iy}#1};}
\begin{tikzpicture}[every path/.style=
                {line width=1pt,->}]
   \matrix[nodes={circle,draw},
         row sep=1cm, column sep=1cm,
         execute at begin cell=\vrt]
      { v  &  w  &  x  \\
          &  y  &  z  \\ };
   \ExplSyntaxOn
      \graph_map_edges_inline:Nn
         \l_my_graph
         { \draw (#1) to (#2); }
   \ExplSyntaxOff
\end{tikzpicture}
```



Just to be clear, this library does not, inherently, understand any Ti*k*Z. What it does is help you to analyze the structure of your graph. For example, does it contain a cycle?

```
\ExplSyntaxOn
   \graph_if_cyclic:NTF
      \l_my_graph {Yep} {Nope}
\ExplSyntaxOff
```

Nope

Indeed, there are no cycles in this graph. While we're at it, is vertex `w` reachable from vertex `y`?

```
\ExplSyntaxOn
   \graph_acyclic_if_path_exist:NnnTF
      \l_my_graph {y} {w} {Yep} {Nope}
\ExplSyntaxOff
```

Nope

Quite true. Finally, and most importantly, you can interpret the graph as a dependency graph and list its vertices in topological order:

```
\ExplSyntaxOn
  \clist_new:N \LinearClist
  \graph_map_topological_order_inline:Nn
     \l_my_graph
     { \clist_put_right:Nn
       \LinearClist {\texttt{#1}} }
\ExplSyntaxOff
$ \LinearClist $
```

`v`, `w`, `x`, `y`, `z`

A topological order is not uniquely determined. The important thing is that the constraints imposed by the graph are respected.

4 The `pkgloader` package (Part 2)

The `pkgloader` package uses graph vertices to represent LaTeX packages and arrows to encode package ordering rules. At the end, selected packages are loaded in topological order.

Let's take a more detailed look at some of the features of `pkgloader`.

4.1 Rules

Each **`\Load`** rule can contain a number of different clauses. We look at them one by one.

It usually contains a *package description*, consisting of a name, a set of options and a minimal version, just like the **`\usepackage`** command:

```
\Load [options] {package-name} [version]
```

It can contain a *condition clause*, indicating when the rule should be applied. This takes the form of a Boolean formula in `expl3` style, in which the atomic propositions are package names:

```
\Load {pkg1} if {pkg2 || pkg3 && !pkg4}
```

This rule would load `pkg1` if either `pkg2` will be loaded too, or if `pkg3` will be loaded but `pkg4` will not. Alternatively, the condition clause can be **`always`**, indicating that the rule should be applied under any conditions. Finally, the keywords **`if loaded`** can be used to apply the rule only if the package named in the package description is requested anyway. This is the default behavior, but the keywords can be included to make it explicit.

There is one exception to the structure described above. Instead of a package description, a rule can contain the **`error`** keyword, followed by a condition clause, to describe conditions that should never occur — usually invalid package combinations:

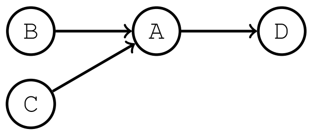
```
\Load error if {pkgX && pkgY}
```

When multiple condition clauses are present in a single rule, their *disjunction* is used. In other words, the rule is applied if *any* of its conditions is satisfied.

A non-error rule may contain an *order clause*, to ensure that the package described by the rule is loaded in a specific order with regard to other packages:

```
\Load {A} after {B,C} before {D}
```

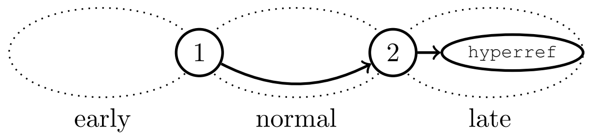
This rule ensures that if package `A` is ever loaded, it is never loaded before `B` or `C`, and never after `D`. This is where the graph representation comes in. The above rule would yield a graph like this:



This can take care of specific known package ordering conflicts. But some packages should, as a rule of thumb, be loaded before all other packages, or after all others, unless specified otherwise. A typical example is the `hyperref` package, which should almost always be loaded late in the run. For this, the **`early`** and **`late`** clauses may be used:

```
\Load {hyperref} late
```

The **`early`** and **`late`** clauses work by ordering the package relative to one of two placeholder nodes:



These two nodes are always present in the graph. Ordering a package **`early`** is intuitively the same as ordering it '**`before {1}`**'. And ordering it **`late`** is the same as ordering it '**`after {2}`**'. All packages that are, after considering all rules, not (indirectly) ordered '**`before {1}`**' or '**`after {2}`**' are automatically ordered '**`after {1} before {2}`**'. A rule can have any number of order clauses, and all are taken into account when one of the conditions of the rule is satisfied.

Finally, a rule can be annotated with a *reason*, explaining why it was created:

```
\Load {comicsans} always
    because {that font is awesome!}
```

This text does not have any effect on the behavior of the rule. It is meant for human consumption, though should not be formatted in any way. It should be semantically and grammatically correct when following the words "This rule was created because ...". It can also be used for citing relevant sources. It is used in certain `pkgloader` error messages and may eventually be used to generate documentation.

4.2 Rulesets

You may be wondering: who makes up these rules?

Short answer: Anyone. Rules can be placed directly inside the `pkgloader` area, but they can also be bundled in a `.sty` file. By default, `pkgloader` loads a recommended set of rules, allowing the average user to get started without any hassle. But this behavior can be overwritten using package options:

```
\RequirePackage[recommended=false,
               my-better-rules]
              {pkgloader}
    ...
\LoadPackagesNow
```

This means: the `pkgloader-recommended.sty` file, which is usually preloaded by default, should *not* be loaded for this document. Instead, load the `pkgloader-my-better-rules.sty` file.

Take note of the following: every ruleset should be bundled in a `pkgloader-`⟨*something*⟩`.sty` file, and can then be loaded by specifying ⟨*something*⟩ as a package option.

So basically, any user can create rules for their own documents, or distribute custom rulesets, e.g., through CTAN. But primarily, I expect two groups of people to author `pkgloader` rules:

The LaTeX community: The `recommended` ruleset would, ideally, be populated further through the efforts of anyone who diagnoses and solves package conflicts. Perhaps through websites like `tex.stackexchange.com`, or by filing issues or pull-requests to the `pkgloader` Github page.

Package authors: `pkgloader` will eventually be directly usable for package authors just as for document authors, to include their own rules from right inside their packages. Rather than manually scanning for and fixing potential conflicts, they could leverage `pkgloader`, as in:

```
\RequirePackage{pkgloader}
   \Load me before {some-pkg}
   \Load me after {some-other-pkg}
\ProcessRulesNow
```

It may be possible to apply such rules in the same LaTeX run in which they are encountered. But if not, the package manager will know what to do in the next run through the use of auxiliary files. This functionality has not yet been implemented.

4.3 Error messages

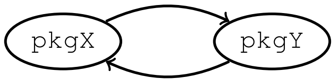
There are two types of error messages that may be generated by `pkgloader`.

The first type of error message happens when an **`error`** rule is triggered. It looks like this:

```
A combination of packages fitting
the following condition was requested:
    ⟨condition⟩
This is an error because ⟨reason⟩.
```

The second type of error message is a bit more interesting. Since rules can effectively come from any source, it is possible to apply rules that contradict each other. To give an (unrealistic) example:

```
\Load {pkgX} always before {pkgY}
         because {pkgX is better}
\Load {pkgY} always before {pkgX}
         because {pkgY is better}
```



A potential circular ordering is not necessarily a problem, so long as both rules are never applied in the same run. But in this exact example, the following error message will be generated:

```
There is a cycle in the requested
package loading order:
          pkgX
   --1--> pkgY
   --2--> pkgX
The circular reasoning is as follows:
(1) 'pkgX' is to be loaded before
    'pkgY' because pkgX is better.
(2) 'pkgY' is to be loaded before
    'pkgX' because pkgY is better.
```

Whenever this happens, the user may want to reconsider one of their included rulesets, or file a bug-report to the responsible party — especially if the circularity comes from the `recommended` ruleset.

4.4 Options and versions

The package manager need not be confined to playing with the package loading order. While intercepting package loading requests, it will be able to accumulate package options and versions as well, and then combine them in a multitude of flexible ways. Possible ways of combining option-lists include:

- Concatenate all option-lists for the same package into a single list, in any arbitrary order.
- Interpret `key=value` options, and generate an error message when two different instances call for two different values for the same key.
- Provide a tailor-made function for combining option lists for a specific package.

As for package versions, it would make sense to take the 'maximum' version-string that is encountered, and use that to load the actual package.

As of this writing, neither of these features has been implemented.

5 Conclusion

I hope this article and the packages described therein have been useful and/or inspiring. And I hope to have convinced you that the idea of a LaTeX package manager is worth pursuing.

Both packages are available on CTAN:

`www.ctan.org/pkg/pkgloader`
`www.ctan.org/pkg/lt3graph`

I love feedback, and I love questions. I can be reached through the e-mail address and website specified in the signature block below the references. Any kind of feedback or patches regarding one of the packages should go through their Github pages:

`github.com/mhelvens/latex-pkgloader`
`github.com/mhelvens/latex-lt3graph`

Happy TeXing!

References

[1] David Carlisle. xii.tex, December 1998. http://www.ctan.org/pkg/xii.

[2] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. *SIGPLAN Notices*, 46(2):13–22, February 2011. Proceedings of GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands. http://doi.acm.org/10.1145/1942788.1868298.

[3] Freek Dijkstra. LaTeX package conflicts, June 2012. http://www.macfreek.nl/memory/LaTeX_package_conflicts.

[4] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, MA, USA, 1986.

[5] Seamus. What are category codes?, April 2011. http://tex.stackexchange.com/questions/16410/what-are-category-codes.

[6] Wikipedia. Domain-specific language, 2014. http://en.wikipedia.org/wiki/Domain-specific_language.

[7] Wikipedia. Monkey patch, January 2014. http://en.wikipedia.org/w/index.php?title=Monkey_patch&oldid=586056263.

[8] Wikipedia. Turing completeness, 2014. http://en.wikipedia.org/wiki/Turing_completeness.

◇ Michiel Helvensteijn
Leiden University,
Niels Bohrweg 1,
2333 CA, Leiden,
the Netherlands
mhelvens+latex (at) gmail dot com
http://mhelvens.net