Why and How I Use LilyPond

Daniel F. Savarese

Version 1.1

Copyright © 2018 Daniel F. Savarese[1]

Introduction

In June of 2017, I received an email from someone using my classical guitar transcriptions inquiring about how I use LilyPond[2] to typeset (or *engrave*) music. He was dissatisfied with his existing WYSIWYG[3] commercial software and was looking for alternatives. He was impressed with the appearance of my transcription of *Lágrima* and wondered if I would share the source for it and my other transcriptions.

I sent the inquirer a lengthy response explaining that I'd like to share the source for my transcriptions, but that it wouldn't be readily usable by anyone given the rather involved set of support files and programs I've built to support my music notation efforts. I never heard back from the inquirer and don't know if he even received my reply.[4] Perhaps he was discouraged by the amount of work LilyPond appeared to require.

Given the time and effort I spent crafting my explanation of my use of LilyPond, I decided in January of 2018 to edit and expand my email reply into this article to share with anyone who may be considering using LilyPond, especially for notating guitar music. In this article, I explain why I use LilyPond and how I use LilyPond, along with a high-level summary of the support tools I have built to streamline the engraving process.

Why I Use LilyPond

Many years ago, before the LilyPond project existed, a few attempts had been made to implement music engraving features on top of TeX, Donald E. Knuth's widely used (at least in academia) typesetting program. I tried using some of those programs and macro packages, but they were very limited in function. I wanted to use music engraving software, not spend my time developing it. That's how I got started using Finale, the commercial music notation software from Coda Music Technology, which was later bought by Net4Music SA (eventually becoming MakeMusic, Inc.). It cost me an arm and a leg, even with an academic discount. I never got my money's worth out of it. At the time I couldn't explain exactly why, but I was never productive using it.

Years later, when I started playing piano, I upgraded to the latest version of Finale and suddenly found it easier to produce scores using the software. It had nothing to do with new features in the product. After notating eight original piano compositions, I realized that my previous difficulties had to do with the idiosyncratic requirements of guitar music that were not well-supported by the software. Nevertheless, note entry and the overall user interface of Finale were tedious. I appreciated how accurate the MIDI playback could be with respect to dynamics, tempo changes, articulations, and so on. But I had little need for MIDI output.

Searching for software that would let me generate scores more rapidly, I tried Sibelius, the British music notation software produced by Sibelius Software Ltd. that later became a part of Avid Technology. I immediately found Sibelius much easier to use. Most important, I found I could notate guitar music with much less difficulty. That is, with a caveat: Sibelius required a lot of customization to support guitar notation. Once I made those customizations, I was more productive. But as with all WYSIWYG programs that use proprietary binary file formats, there were problems I just couldn't live with.

Version Control

Version control is difficult with binary file formats. If you use a general purpose version control system, performing a **diff**[5] won't show you what notes or markup changed. If you use an application-specific version control system, such as the one included with Sibelius, you can only examine changes graphically; and you can't integrate with the the revision control system you may be using for a larger parent project without maintaining a duplicate versioning tree.

[1] https://www.savarese.org/

[2] LilyPond [http://www.lilypond.org/] is an open source text-based music engraving program.

[3] What You See Is What You Get (i.e., software with a graphical as opposed to a text-based user interface).

[4] It has unfortunately become quite common for valid email to be silently misclassified as spam by the major email service providers.

[5] A command that shows all of the differences between two text files.

LilyPond relies on text input files, allowing you to use the same revision control system you use for all of your other projects.

Music Reuse

Repeating music without having to copy it is awkard at best in WYSIWYG programs. Some programs have a feature that allows you to make a dynamically updated copy of a set of measures, but it is cumbersome and inflexible. Including music or music fragments from a single source in multiple projects is largely impossible.

LilyPond allows you to store music fragments in macro variables that you can reference multiple times. You can even selectively turn off decorations on each macro instantiation. For example, if the first appearance of a measure includes fingering, you can turn it off for the later appearances by defining a new macro that turns off fingering before referencing the original macro and turns it back on afterward. Including music in multiple projects is also easy, but requires you to think ahead with respect to how you organize your files and name your variables. I'll discuss this issue in the "how" section.

Single Source, Multiple Outputs

Generating multiple outputs from same source using commercial software is largely impossible. You generally have to maintain two separate files with different formatting.

With LilyPond (and my custom preprocessor), I can generate guitar music that uses roman numerals for positions and barres or arabic numerals without making any changes to the source file (cf. Llobet's *Preludio (en re mayor)*).

Part Extraction

Part extraction is an inconvenience that essentially requires you to format a piece of music an additional time for each extracted part. There is no need for part extraction when using LilyPond. As long as you organize your project with different parts in different files—or at least different parts in different variables—you can generate a score for a subset of parts with little effort. Your parts are not tightly bound together. Therefore, they need not be extracted.

File Format

I also had concerns about the longevity and interoperability of the propietary file formats and found that MusicXML lost too much information to be a viable export format. As already mentioned, I wanted a text-based format that could be versioned using the same source code revision control system(s) I use for software development. Of course, it needed to produce professional-calibre output, but flexibility and customizability were higher priorities. LilyPond met most of my requirements and seemed to be able to be molded into doing whatever I needed via its embedded Scheme language or via preprocessing. All that said, LilyPond does not do "the right thing" out of the box (except perhaps for piano music). With respect to guitar notation, just like Sibelius, LilyPond requires a lot of customization—an issue I'll address shortly.

How I Use LilyPond

LilyPond is no panacea, especially when it comes to guitar notation. Over the years, I have developed an extensive set of functions and macros to support the engraving process and I'm still not done. My customization libraries are roughly divided into a common set of instrument-independent functions, macros, and default formatting directives and a set of instrument-specific include files. I've also had to write a compiler wrapper, dependency generator, and score file generator all of which are used by a build system that ties everything together.

Libraries

At the base of the library tree sits `use.ily`,[6] which defines a custom `\use` function that includes a file only once if it has already been included. The standard LilyPond `\include` will re-include a file, which causes tremendous problems when you try to reuse music from different files—re-inclusions can blow the stack and crash the program. The \use function is automatically available to all source files by way of **lyc**, a compiler script that wraps the **lilypond** command with various options and preprocessing. Therefore, no source files ever use `\include` directly; they all use `\use` instead. The only use of `\include` is the statement dynamically inserted by **lyc** to include a common preamble that at the moment only consists of `use.ily`.

[6] This used to be called `import.ily`, but it had to be renamed after it was found to conflict with the Guile 2.0 `import` REPL command. Technically, there is no `use.ily` file. There are `use-v1.ily` and `use-v2.ily` files that are used for Guile 1.8 and Guile 2.x respectively. The `\use` function must be implemented differently depending on the version of Guile being used by LilyPond. All of this is hidden from the user, so we treat the implementation conceptually as a single `use.ily` library file.

The next layer in the library tree is a common library, defining instrument-independent customizations, including custom markup functions, convenience macros, default layout and paper definitions, as well as numerous additional customizations. In some sense, the default settings in the common library file constitute a style (e.g., default margins, fonts, etc.), but I treat each piece individually and adjust staff size, spacing, and margins, on a per-piece basis. For a collection of works, I try to keep the styling parameters as consistent as possible. From there, the library tree branches into classes of instruments (e.g., strings) and then specific instruments (e.g., violin or guitar). For classical guitar, I have custom functions for rendering barre, position, and string indications as well as macros for right-hand fingering.

Tools and Build System

I use a build system based on GNU autoconf and automake which generates a GNU **make** `Makefile`. In addition to **lyc**, the compiler script I mentioned, I have written **lydep**, a dependency generator that figures out all of the dependencies for a given source file in a manner compatible with GNU **make**. That way, the `Makefile` can rebuild exactly what needs to be rebuilt when a given source file changes. For example, if a library file is modified, then all scores that directly or indirectly import that file are rebuilt. The dependency generator takes into account LilyPond `\include` directives and my custom `\use` directives.

Finally, I have written **lygen**, a tool that generates a `.ly` source file from a high-level description of the score implemented as a Lua[7] table. For example, my score file for *Lágrima* looks like this:

```
return {
  source = "Lagrima.ily",
  document = "score",
  instrument = "guitar",
  namespace = "g_tarrega_lagrima",
  options = {
    subtitle = true,
    remove_empty_staves = true,
    paper = {
      top_margin = "0.75\\in",
      bottom_margin = "0.75\\in"
    }
  }
}
```

As you can see, the file contains no LilyPond markup. All the music goes in `Lagrima.ily`, which I will describe in my discussion of namespaces in the next subsection.

Tips

Perhaps the most important practice to apply when using LilyPond is not to pollute the global namespace. Assign each element of your score to a variable and name all of the variables with a unique score-specific prefix. If you place music in the global namespace, you can't reuse it elsewhere. You'll notice the cryptic "g_tarrega_lagrima" assigned to the `namespace` parameter in `Lagrima.ly.lua` above. That's a namespace prefix. LilyPond has no notion of namespaces, requiring you to approximate them via variable naming conventions. If you write a bunch of scores and decide you want to collect them into a book or if you want to include music from one file into another, you'll find it won't work if you used global artifacts. You must avoid polluting the global state. Everything should be encapsulated. My `Lagrima.ily` source file looks like this (without the data):

```
\use "classical_guitar.ily"

g_tarrega_lagrima_title = ...
g_tarrega_lagrima_subtitle = ...
g_tarrega_lagrima_composer = ...
g_tarrega_lagrima_copyright = ...

% Separate out first part so we can use
% reference it for midi output.

g_tarrega_lagrima_voice_one_a = { ... }

g_tarrega_lagrima_voice_one_b = { ... }

g_tarrega_lagrima_voice_one = {
  \repeat volta 2 {
    \g_tarrega_lagrima_voice_one_a
  } |
  \g_tarrega_lagrima_voice_one_b
}

g_tarrega_lagrima_voice_two_a = { ... }

g_tarrega_lagrima_voice_two_b = { ... }

g_tarrega_lagrima_voice_two = {
  \repeat volta 2 {
    \g_tarrega_lagrima_voice_two_a
  } |
  \g_tarrega_lagrima_voice_two_b
}

g_tarrega_lagrima_voice_three_a = { ... }

g_tarrega_lagrima_voice_three_b = { ... }

g_tarrega_lagrima_voice_three = {
  \repeat volta 2 {
    \g_tarrega_lagrima_voice_three_a
  } |
  \g_tarrega_lagrima_voice_three_b
```

[7] Lua [https://www.lua.org/] is an interpreted programming language.

```
}

g_tarrega_lagrima_ossia = { ... }

g_tarrega_lagrima_ossia_staff = { ... }

g_tarrega_lagrima_music_body = { ... }

g_tarrega_lagrima_score_body = {
  \new Staff = "main" {
    \guitar_clef

    \key e \major
    \time 3/4
    \tempo "Moderato" 4 = 90
    \set Timing.beamExceptions = #'()

    \tag_midi {
      \unfoldRepeats
      \g_tarrega_lagrima_music_body
    }
    \tag_nomidi \g_tarrega_lagrima_music_body
  }
}
```

Other than the `\use`, there are no directives at the global scope. Everything is assigned to a variable.

Another essential practice is to notate voices separately. Never use `\parallelMusic`. Because WYSIWYG programs facilitate entering music measure by measure—and when you transcribe a piece from lute tablature, for example, you tend to go measure by measure instead of voice by voice—I started using LilyPond with `\parallelMusic`. That was a big mistake. It took a very long time to convert all of that music to separate voice blocks. `\parallelMusic` doesn't work well with many LilyPond features, making a lot of things not work or simply render incorrectly. I didn't have the presence of mind to realize that if I wanted to enter music measure by measure easily, I could keep all of the voices in separate blocks and simply split my text editor into different editing regions, one per voice. Once I did that, entering music measure by measure or voice by voice was simple. I could see the notes for all the voices in a single measure at the same time.

Along with notating voices separately, you absolutely must number your measures with a comment preceding each measure. Otherwise, locating your place in a piece of music becomes difficult and version control *diffs* become indecipherable.

One of the worst aspects about Finale and Sibelius is having to adjust every little thing by hand. Right-hand fingering was rarely automatically placed pleasingly (LilyPond does much better) and good luck repeating a section of music elsewhere without the fingering without having to copy it. With LilyPond, you still have to make adjustments with `\tweak`, but it's clear exactly what you've adjusted manually and by how much. Also, when repeating a section of music somewhere, you can simply reference a macro variable. If you want to repeat the section without fingering, there's no problem. Just disable the fingering before the section and enable it afterward.

Is LilyPond for you?

If at the outset I had realized how much code I would have to write to do everything I needed, I probably wouldn't have used LilyPond. At the same time, I don't think MuseScore or other open source alternatives would have met my needs. I would have eventually turned to LilyPond accepting how much work I'd have to do.

As I've mentioned, LilyPond requires a lot of customization for guitar notation. Chord diagrams can probably be made easy after a lot of up-front work creating a library. I've thought of auto-generating a chord diagram library from a high-level description in Lua or JSON. Tablature is so easy with the WYSIWYG programs that I always provided it (although part extraction to generate a non-tablature edition was a bit of a pain). With LilyPond, tablature can get quite messy if the defaults aren't what you want. You have to enter a lot of string number indications and turn off string number rendering for the ones you don't want rendered. At the time I started using LilyPond, I didn't want to go to any extra effort to generate tablature (because I don't use it). But now that I think about it, one could write a music event function that conveys the string number without rendering it, and use that for tablature. Another alternative is to use tags. Still, there are some pieces that would require you to specify a string number for just about every note; not something I would want to do.

If you are comfortable with the WYSIWYG experience, I'd recommend looking at other WYSIWYG alternatives before considering LilyPond. You may have noticed one reason I haven't given for why I use LilyPond. I've never said “LilyPond produces more beautiful output than *fill in the blank*.” That is because virtually all of the major music notation software packages produce readable music. Whether one is much better than another is very subjective. Unless you love LilyPond output, I would not recommend using it solely for aesthetic reasons. Producing a beautiful score with LilyPond requires effort and doesn't just happen automatically.

If you don't have experience with programming and the fundamentals of modularization, program organization, and project organization, you should probably delay using LilyPond until you've acquired some of those skills. Even

if you're prepared to learn gradually as you use the software, I can easily see someone with no programming experience hitting a brick wall when the only way to solve a problem is to write a Scheme function. Most of LilyPond's shortcomings can be overcome via its extensibility features. But if you can't understand how to use them, they may as well not exist.

Revision History
Revision 1.1 2018-05-09
Changed *import* to *use*.
Revision 1.0.1 2018-02-07
Added measure number comments tip.
Revision 1.0 2018-01-30
First draft.