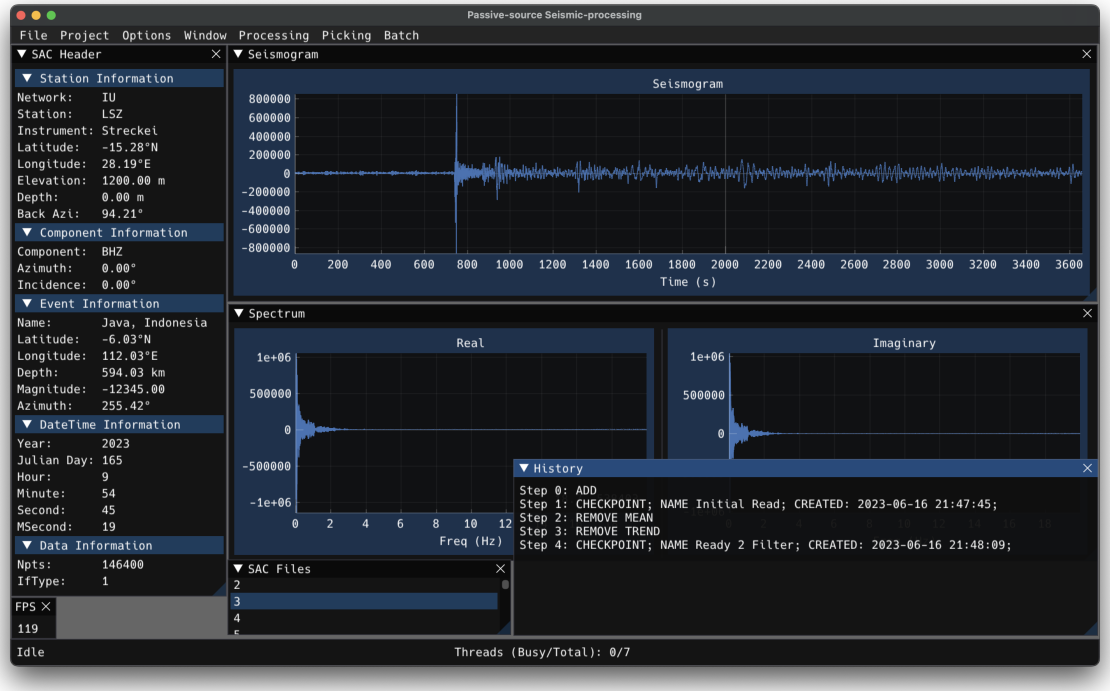
Passive-source Seismic-processing (PsSp)

PsSp aims to provide an OS-independent, graphical, seismic-processing software package targeted at passive-source seismologists.

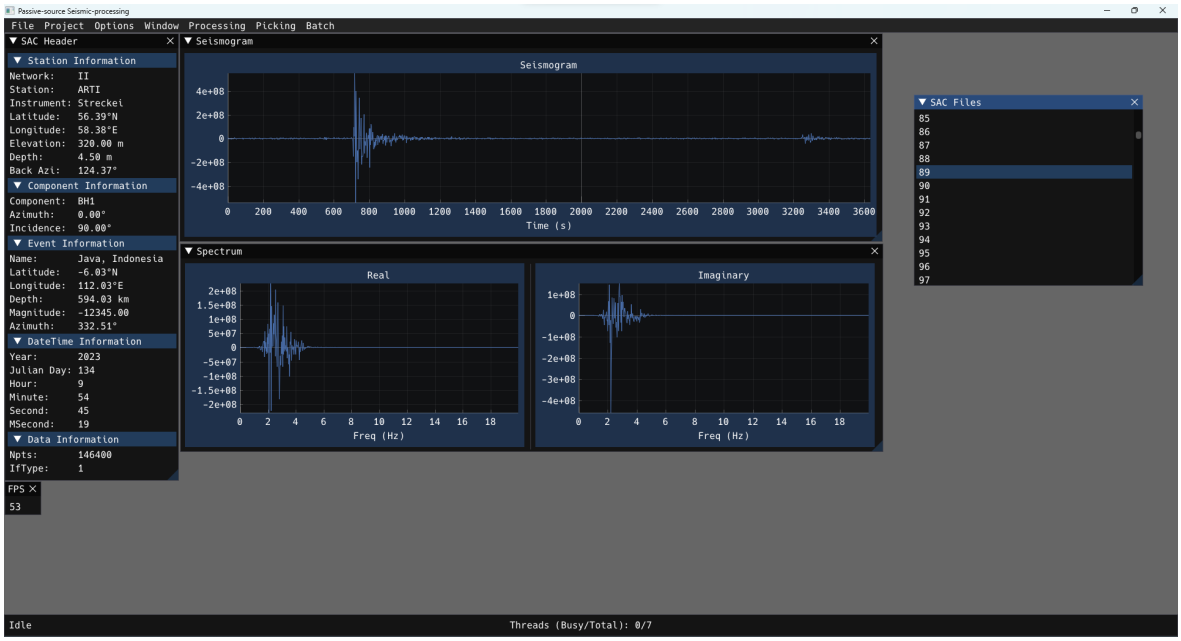
Screenshots

MacOS 13.4 - 16 June 2023



Ubuntu Linux 23.04 - 22 June 2023

Windows 11 - 22 June 2023



Why does this exist?

Summary

The purpose of this project is to **extend the productivity suite of the passive-source seismologist**. There exist great tools for writing manuscripts (e.g. MS Word, LibreOffice, LaTeX, ...). There exist great tools for creating presentations (e.g. MS Powerpoint, Impress Presentation, ...). There exist great tools for communicating with each other across the world (Outlook, Thunderbird, Zoom, MS Teams, ...). What tools exist for doing seismic analysis? Far too often, it is whatever the analyst manages to cludge together, provided it *seems* to do the job.

Introduction

Despite the various seismological tools that exist, and because of how they are designed, the seismologist will **most likely** need to code their own tool(s) (as a shell script stitching programs together, as a Python script using ObsPy, as a SAC macro, etc.). While the ability to do that if it is desired by the researcher is awesome, the need to do it is unfortunate as not everyone wants to (or knows how to) write their own codes. It gets worse when you consider the performance of these codes, or how the codes end up becoming obsolete after a short time (try using someone's old Python scripts, or Matlab codes, have them not work and be stuck trying to figure out what is wrong instead of making progress on your research).

Discussion

The primary issues that I see today are:

1. There is a lack of available tools with a modern graphical user interface (GUI).
2. Often tools only do one or a few jobs. This makes life easier for the developer (following the KISS philosophy), but it makes life harder for the end-user. Often the end-user needs to stitch/cludge together different tools, developed by different persons/groups, in order to perform a given research task.
3. The additional complication of OS-exclusive software, locking users of the wrong operating system out from certain tools is really quite unfortunate.
4. The tools are often not parallel at all.

The problem is magnified when you consider that often the end-user doesn't necessarily know how to use the tool, nor the underlying assumptions, nor the limitations. Often, these tools were never designed to be shared and therefore are designed in a non-intuitive fashion, with virtually no comments in the actual code. These tools are often not documented (or under-documented, or even *incorrectly documented*), they tend to be assumed as just plain *obvious*, despite that being entirely dependent upon a very specific (and undocumented) workflow by someone who simply does not care about UI/UX.

The disconnected nature of the typical seismic workflow leads to reproducibility issues. A researcher must keep track of every step taken in the analysis manually, without error. This is easy when a research task is a straight line. However, when there is back-tracking, iterative analysis with minor tweaks, abandoned lines of exploration, and so on, it becomes exceedingly difficult to be able to provide an accurate account of the actual processing steps necessary to consistently reproduce presented/published results. In this age of modern

computing, it is simply **absurd** that the seismologist has no other choice than to work with this *severe tool-deficit*.

The researcher shouldn't need to expend immense amounts of time/energy/mental-bandwidth on making their tools, nor on making them work together. They should be focused on doing science. While advances in machine learning are allowing the modern seismologist to parse massive amounts of data with relatively little effort, we must still look at our data and question the validity of our analysis/interpretation. And we should be able to do this with relative ease and minimal pain.

Purpose

PsSp is being developed to solve these problems; to empower the seismologist with tools that are easy to use and foster exploration. By enabling the scientist to do exploratory analysis quickly, easily, iteratively, and visually I hope to allow the end-user to improve their intuitive understanding of what they are doing with their data so that they can make an informed descision of how best to proceed with their analysis. I hope this will also make entry into seismology easier (undergrads, summer interns, new graduate students, etc.) and will make it easier for more-seasoned seismologists to use newer and more advanced tools, thus improving everyone's workflow and the quality of research that is accomplished while minimizing the amount of time (and frustration) devoted to simply trying to get a functional workflow.

Current status

This is extremely early in development.

Current Focus: Unit and Integration Testing for Improved Stability

This project has gone too far without proper testing. Bugs are hard to find, they disrupt the mental flow when working on a given problem via distraction with a new, different, and annoying problem. Testing will help mitigate these issues. As the code-base grows, this will become progressively more important and more difficult to freshly introduce to the workflow. To that end I am extending the freeze on new analysis functionality. If this is going to be used it cannot cause the analyst headaches due to being unstable.

To that end, the focus will be on implementing unit testing and integration testing. I will be using Catch2 to setup and execute the tests. Once that is all said and done, another round of bug squashing will need to occur. After that, there will finally be a sufficiently stable base to justify building upon. I'm also looking

into using Valgrind to incorporate some more advanced dynamic analysis tools into the development workflow.

Last Focus: Memory Management

All data used to be maintained in memory all at once. Assuming that will be the case for all possible projects would be beyond naive. To that end, I implemented a data-pool object that handles distributing smart-pointers to data objects in memory. If a requested object is not in memory, it gets loaded in. Only a finite number are allowed to be in the memory and the user can change that amount (currently limited to a minimum of `n_threads` to prevent deadlock and a maximum of INT_MAX, which is simply absurd and should be updated in the future). If the pool is full, an unused data object in memory is migrated to a temporary data table in the sqlite3 database for the project. The data-pool must allow at least as many objects as the number of threads in the thread-pool, otherwise the ensueing competition for data from each thread will result in deadlock. Smaller data-pools result in slower operations, having as much data in memory as possible is fastest. There is a lot more work to do on memory management, but I'd like to build a more stable base through unit/integration testing.

ToDo

See the Todo list at the top of the ToDo.md file for more info on what is currently going-on/planned for the future as well as the above discussion on the project focus.

Dependencies

Dependencies that are marked as 'Git submodule' are handled automatically. Other packages must be installed via your package manager of choice or manually. For those other packages I provide installation guidance for MacOS, Linux, and Windows systems here.

Necessary

- Boost
  - Provides some convenient string manipulation operations.
- Dear ImGui v1.89.5
  - This provides the OS-independent GUI.
  - Git submodule.
- FFTW3
  - This is necessary for spectral functionality (FFT, IFFT).

    - By using a plan-pool, that has an appropriate semaphore lock, I have implemented fft and ifft in a thread-safe fashion (super-fast!).
- GLFW3
    - This is a graphical backend for the GUI.
- ImGuiFileDialog, Lib_Only branch
    - This adds OS-independent File Dialogs to Dear ImGui.
    - Git submodule.
- ImPlot
    - This adds OS-independent plotting functionality to Dear ImGui.
    - Git submodule.
- MessagePack
    - Provides data-serialization, used to serialize/deserialize program settings from a binary file.
- sac-format
    - This provides binary SAC-file (seismic) I/O, both low-level functions and the high-level SacStream class.
    - Git submodule.
- SQLite3
    - Projects are implemented as internal sqlite3 databases.
    - We are able to maintain data provenance information, processing checkpoints, and so on via a serverless relational database.

Optional

- Catch2 v3.2.2
    - This provides the unit/integration testing framework
    - Git submodule.
- Xoshiro-cpp
    - This provides good and fast pseudo-random number generation
    - Currently only used to generate random data for unit tests
    - Will be implemented in PsSp for use eventually (to generate random noise, generate random perturbations in an inversion, etc.)
    - Git submodule.

Compilation instructions

I test this on M1 MacOS (Ventura 13.4.1), x86_64 Linux (Ubuntu 23.04), and x86_64 Windows (Windows 11, using MSYS2/Mingw).

MacOS

Using Homebrew

```
brew install fftw glfw msgpack-cxx sqlite boost
```

NOTE For MacOS users, if you want a stand-alone `Application` (`PsSp.app` , no need to execute from the terminal) there are additional requirements. Please see the additional instructions for more information.

Linux (Ubuntu 23.04/Debian based)

Using apt

```
sudo apt install libfftw3-dev libglfw3-dev libboost-all-dev libmsgpack
```

Windows (Windows 11)

Using MSYS2

```
pacman -S mingw-w64-x86_64-toolchain mingw-w64-x86_64-glfw mingw-w64-x
```

Clone and Initialze

Next you need to clone this project and initialize the submodules

```
git clone https://github.com/arbCoding/PsSp.git
cd PsSp
git submodule update --init
```

That will download the appropriate submodule dependencies, with the correct commit version, automatically, from their respective GitHub repositories. You can confirm that by examining them inside the submodules directory (they will be empty before you initialize them and populate afterward).

NOTE if a submodule is not the correct version (detached head, but submodule was updated) From the base git dir (PsSp) run

```
git submodule update --init --remote submodules/
```

Then it is a simple as running

```
make
```

The above command will make an executable inside the ./bin/ directory. **Note** that on Linux/MacOS this will be a file simply named `PsSp` and on Windows it will instead be `PsSp.exe`.

To run it on Linux/MacOS use

```
./bin/PsSp
```

The above command is used to start PsSp from the command line. On Linux you can also double click the executable to start it. On MacOS you must first make an `Application` (instructions to make `PsSp.app` are below).

MacOS `Application`

To make `PsSp.app` run

```
make macos
```

`PsSp.app` can be run by double-clicking on the application file.

On Windows you can start it from the command line (MSYS2 knows to switch the `/` around to a `\` automatically).

```
./bin/PsSp.exe
```

Or by double-clicking on the executable.

Cleanup

To cleanup (including removing the compiled programs), run:

```
make clean
```

Testing

To compile and run the tests:

```
make tests
```

Note that this will involve compiling Catch2. This has been automatically integrated into the above command. The tests include verification of features as well as simple benchmarks of various operations performed by PsSp. Testing is

currently part of the main development focus, as discussed in the current status section.

Special MacOS `Application`

If you want a stand-alone MacOS application file, then there are additional steps.

I do not take credit for figuring this out, I found this blog post on the topic.

First, I use dylibbundler to handle rebinding the links for the non-standard dynamically linked libraries. The application bundle requires that they be included in the application (such that the end-user doesn't need to install them).

This can be installed via Homebrew

```
brew install dylibbundler
```

You can see which dylib's will need to be modified via the `otool` command after the program is compiled:

```
otool -L ./bin/PsSp
```

Anything not listed in `/System/Libary/` or `/usr/lib` will need to be included with the application. Fortunately, **dylibbundler** can handle that for us.

```
dylibbundler -s /opt/homebrew/lib/ -od -b -x ./PsSp.app/Contents/MacOS
```

Of course, this is implemented automatically in the Makefile, assuming you also used Homebrew to install the other packages (non-Git submodules).

For more details, checkout the Makefile. It is heavily commented to make it more accessible.