Design Manual - F21DG

Callum Stewart, Saad Badshah, Abigail Rivera, Daniel Scott, Bruce Wilson, Sebastian Zebrowski

1 Preface

1.1 How to use this manual

If you are reading this manual, congratulations, you own your very own copy of the F21DG group project implementing EMD and STFT analysis entirely in browser! This manual contains information detailing both how the end user will make use of the system, and how future software development students may make use of the system.

Alongside this, the manual attempts not only to share details on how the system works, but guide future developers through why certain decisions were made, so that if in the future they wish to modify it, we may already have an explanation on why parts are designed the way that they are.

We do hope that this system can be expanded, and that by the time students do wish to do so, further advances in software that we began to look at (e.g. WebGL) can be fully utilised to further improve the teaching abilities of this software.

1.2 Description of the user

This created tool intents to be an educational resource allowing end users to interactively explore the differences between STFT and EMD time-series analysis techniques. However, while the tool is a prototype, we do wish for it to have a high degree of usability. As such, the tool is implemented in such a way that users have the ability to bookmark examples, use pre-generated signals as inputs, view the demonstration of their combined input signal being decomposed, and much more - detailed in the System Description section.

2 End-User guide

3 System Description

This section contains documentation detailing the top-level design of the system, alongside further documentation on how the future developer reading this can modify and reuse this code in the future. It will first guide through an overview of how the entire system works, building up background information that should be understood first, then continuing on to dive into specifics.

3.1 Design overview

3.1.1 PYODIDE.

That word may not mean much, however it is critical to how this whole project fits together. Pyodide is a full CPython reference implementation compiled and embedded entirely in browser using a virtual instruction set language named WebAssembly. Using WebAssembly, it is possible to get near (system, configuration, and more dependant) native run-times of languages while encapsulating software systems in browser tabs. Even while being a relatively new technology - initially appearing in 2017 (Haas et al., 2017) - it is supported in the most commonly used browsers, including mobile OS browsers.

Not only does this allow us to write our own Python code that can be run in browser, but we can also make use of Pythons humongous pre-existing library of pre-built packages (wheels) that are written in pure python.

Note: Some packages do however contain C code alongside their Python code, but using the same system that enabled the CPython implementation to be compiled to WebAssembly, we can do the same with the Python packages containing C, allowing almost any package to run entirely in browser.

In summary, Pyodide enables us to use pre-existing, tested, and verified packages for signal generation, EMD and STFT analysis, and graph generation, combining them carefully - alongside performing our own unit and integration testing - into our software system. However Pyodide is not all we shall need to produce the full working system, and there are many caveats with Pyodide itself.

3.1.2 HOW WE MAKE USE OF PYODIDE.

Pyodide appears to be an all-powerful tool, however how much use can we really get out of it? While it is able to solve many of our problems revolving around complex algorithm implementation, signal generation and combining, and graph generation, we still need methods using standard web technologies, i.e. HTML, CSS, JavaScript, to make the system. It is these cross-overs between Pyodide technologies and standard web technologies that presented us with the majority of challenges during implementation.

As such, our use of Pyodide has impacts on how we shape the rest of the system, and choices were made during development on how we would work around caveats that came to light during cross technology implementation. These caveats are detailed below in a broad manner, and are again explored in specific sections later in this manual:

- **Web Workers -** Using Pyodide with its default implementation runs all code in the main tab render thread. This can present a serious problem in any system that makes use of graphical interface, as during any long length computation, the tab can become completely unresponsive, harming the user experience. To fix this, we made use of Pyodide only inside Web Workers. These Web Workers are a simple means to run scripts in background threads, and therefor allow us to run our Python scripts in a completely separate thread, leaving the tab render thread to continue unimpaired. There are however consequences of this fix. When Pyodide is constructed in each Web Worker, it creates its entire own interpreter. This means we cannot make use of the Python global namespace for information handling or sharing, and must share messages from Web Workers back to the Javascript storage handler via post messages. Further complicating the matter, there are restrictions imposed by both Pyodide and Javascript on what objects can be passed and auto-translated between environments. We are able to alleviate this problem though, by passing around Javascript objects that execute the Python code and only returning the final result - using the bullet point below - a raw HTML graph, with all included controls and data, at no point does there require any Python $\rightarrow$ Javascript object translation. A final problem is that the page DOM cannot be accessed from inside a Web Worker Pyodide interpreter, and any attempt to access parts of the 'js' package causes Pyodide to crash. However as we return values from the Web Workers, there is no need to access the DOM directly, and this does not present a problem.

- **MatPlotLib -** MatPlotLib is a comprehensive Python package used to create various forms of visualisation. In our case, and as detailed later in the specific details on our STFT implementation and specific details on our EMD implementation, we make use of the spectrogram and line graph visualisations contained within the library. However, when using the Web Worker technology detailed above, MatPlotLib makes an attempt to access part of the 'js' package as part of its loading process. This immediately crashes the Web Worker thread, even when we don't want MatPlotLib to access the DOM directly, just create a graph to be stored as a Base64 string. It can however, be fixed. MatPlotLib makes use of a frontend - backend system, with the frontend handling the user facing code, and the backend handling the heavy lifting behind-the-scenes to make the figure. MatPlotLib also automatically selects which backend to use depending on the available system, and we found that is part of the default backend code that is attempting to access the 'js' package. To stop this, we can force MatPlotLib to use a different backend, such as AGG (exports high quality images using the Anti-Grain Geometry engine), enabling us to save graphs as we require.

- **mpld3 -**

- **Bookmarking -** As mentioned previously, due to the use of Web Workers, the Python global name space for storing Python generated signal types is not available. However, we can work around this by using the browsers local storage as a replacement, simply involving writing JavaScript handling code.

References

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363.