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.

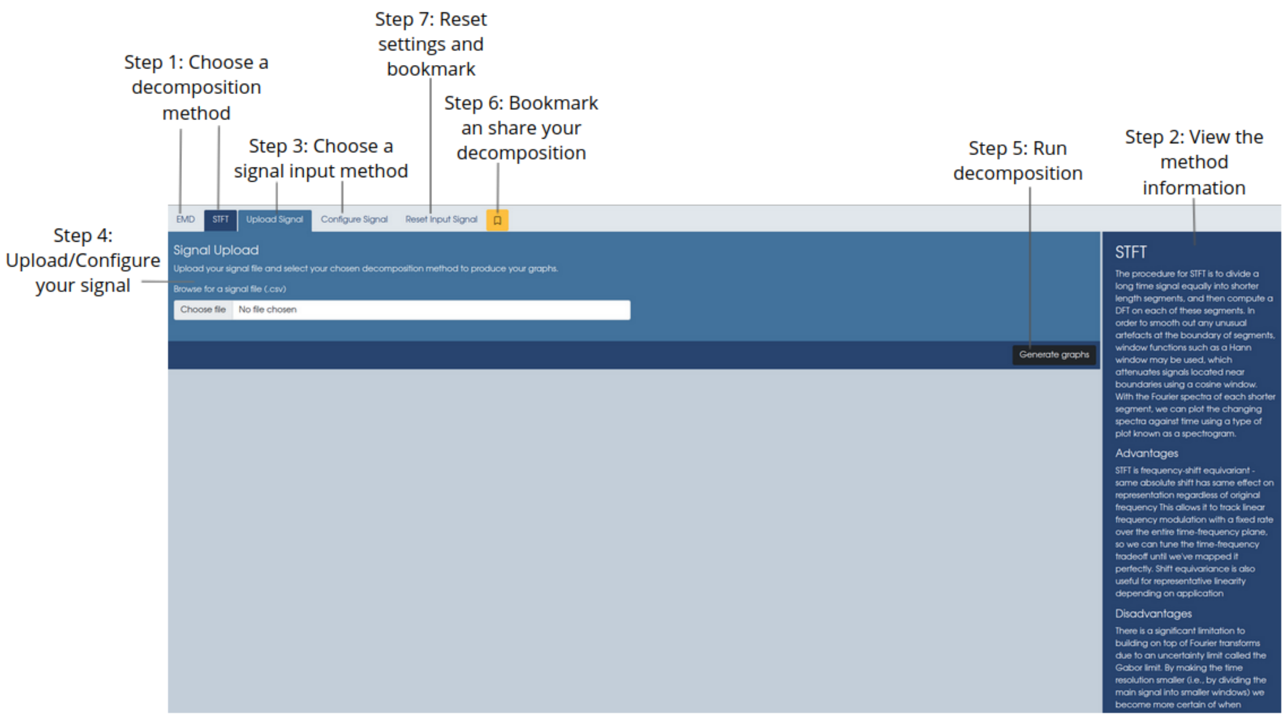
2 End-User guide

2.1 Introduction

This guide demonstrates how to use our developed application for time series decomposition. Our demo video can be found at: Application Demonstration Video

2.2 Quick Start Guide

This is a quick guide to help a user get started with our application.


Step 1: Choose a decomposition method
Step 7: Reset settings and bookmark
Step 3: Choose a signal input method
Step 6: Bookmark an share your decomposition
Step 5: Run decomposition
Step 2: View the method information
Step 4: Upload/Configure your signal
EMD
STFT
Upload Signal
Configure Signal
Reset Input Signal
Choose file
No file chosen
Generate graphs

Signal Upload

Upload your signal file and select your chosen decomposition method to produce your graphs.

Browse for a signal file (.csv)

STFT

The procedure for STFT is to divide a long time signal equally into shorter length segments, and then compute a DFT on each of these segments. In order to smooth out any unusual artefacts at the boundary of segments, window functions such as a Hann window may be used, which attenuates signals located near boundaries using a cosine window. With the Fourier spectra of each shorter segment, we can plot the changing spectra against time using a type of plot known as a spectrogram.

Advantages

STFT is frequency-shift equivariant - same absolute shift has same effect on representation regardless of original frequency This allows it to track linear frequency modulation with a fixed rate over the entire time-frequency plane, so we can tune the time-frequency tradeoff until we've mapped it perfectly. Shift equivariance is also useful for representative linearity depending on application

Disadvantages

There is a significant limitation to building on top of Fourier transforms due to an uncertainty limit called the Gabor limit. By making the time resolution smaller (i.e., by dividing the main signal into smaller windows) we become more certain of when

Figure 1. Application Interface

2.3 User Instructions

2.3.1 Configuring a combined signal

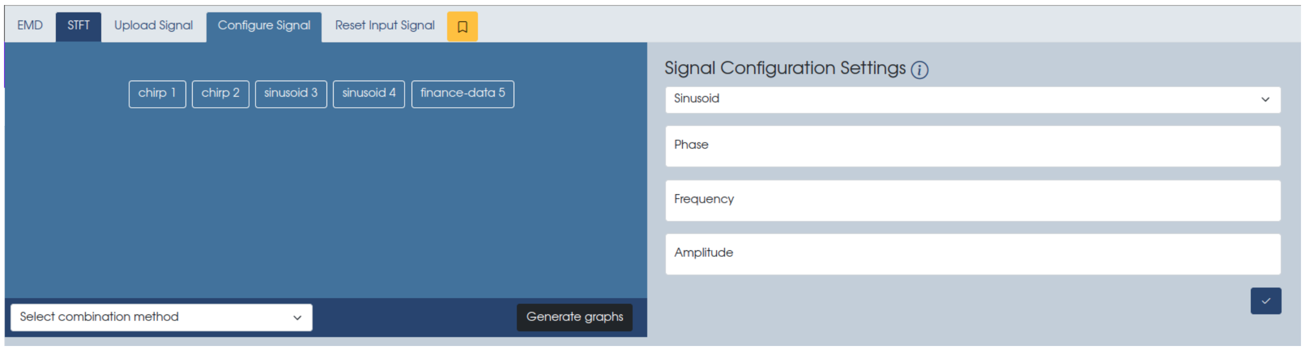


Figure 2. Signal Configuration

1. Click on the configure signal tab found on the top panel.
2. Select a signal type from the drop down in the 'Signal Configuration Settings' panel.
3. Input your values into the various fields.
4. Press the tick button to add your signal. Your signal should appear on the left hand panel.
5. To edit or delete your signal, click on the signal and make changes or delete in the 'Signal Configuration Settings' panel.
6. Set your combination method for your signals. Note: only sinusoids and trends should be combined using product method.

2.3.2 Uploading a signal

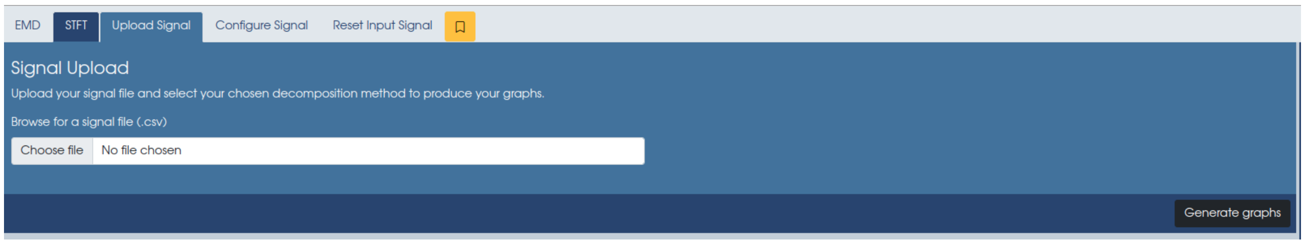

EMD
STFT
Upload Signal
Configure Signal
Reset Input Signal
Signal Upload
Upload your signal file and select your chosen decomposition method to produce your graphs.
Browse for a signal file (.csv)
Choose file
No file chosen
Generate graphs

Figure 3. Signal Upload

1. Click on the upload signal tab found on the top panel.
2. Choose a file to upload. Note: Only .csv files are currently supported.
3. You should receive an upload complete message if the upload was a success.

2.3.3 SELECTING DECOMPOSITION METHOD

1. Click on either the 'EMD' or 'STFT' tab on the top panel to select a method. An information panel will open up on the right hand side.
2. EMD (Empirical Mode Decomposition): This method will result in various IMF (Intrinsic Mode Function) graphs and a Hibert-Huang Spectrum
3. STFT: This method will result in an FFT (Fast Fourier Transform) graph and a resultant spectrogram.

2.3.4 RUNNING DECOMPOSITION METHOD

1. Once you have completed configuring your settings, click on 'Generate graphs'.
2. Your decomposition graphs should appear below the settings panel. This may take a couple of minutes based on your settings.
3. You may click on your charts on the left hand panel to expand them into the larger view area.

2.3.5 BOOKMARKING

1. Once you have completed configuring your settings, simply click on the yellow bookmark button.
2. Your settings will be saved into your URL and your URL is now copied to your clipboard and can be pasted anywhere.
3. You should now be able to access your settings in a different window by pasting the URL into the address bar.

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 custom Python code that can be run directly 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 in most cases 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 the use of pre-existing, tested, and verified packages for signal generation, EMD and STFT analysis, and graph generation, combining them carefully - alongside performing personalised unit and integration testing - into the 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 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 created the majority of challenges during implementation.

We made use of several Python packages during implementation to varying degrees, these packages are listed below:

- **NumPy -** NumPy is the fundamental Python package for scientific computing, implementing multidimensional arrays, matrix structures, alongside basic FFT's and signal generation techniques. We cover the specifics of which functions that we use in the Signal Generation, EMD, and STFT. sections.
- **SciPy -** SciPy is a scientific Python package built on-top of NumPy, incorporating further functions for signal processing, optimisation, and statistics. We make use of several available functions when Generating Signals, alongside performing STFT Analysis.
- **mpld3 -** A Python package that exports a graph MatPlotLib plot to HTML, CSS, JavaScript, and most importantly D3. D3 is a powerful graphics library for JavaScript, that allows for interactive graphs entirely in browser. More specifically, mpld3 makes use of a the MatPlotLib mplexporter framework, which parses an input MatPlotLib plot, and outputs a JSON representation. This representation can then be read in by mpld3.js, a standalone JS framework built on-top of D3, which exports the HTML graph. We also directly use d3 when creating a spectrogram to display STFT analysis. This is because generating a full interactive spectrogram plot in mpld3 can be quite intensive and slow, whereas generating the spectrogram data and then displaying it interactively with d3 alone is significantly faster.
- **micropip -** A Python package specifically for Pyodide, this package allows loading Python packages from PyPi or any arbitrary source at runtime.
- **emd -** A Python package for emperical mode decomposition and related spectral analyses. This package is used to deconstruct the supplied input signal, be that generated or uploaded, and return a list of IMF's to be

displayed in line graphs to the user.

The use of Pyodide also has impacts on how the rest of the system is shaped, 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, and may be referenced later in the 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 Python scripts to run 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 a 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 nor session storage can 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 this case, and as detailed later in the specific details on STFT implementation and specific details on 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. It can however, be fixed. MatPlotLib makes use of a frontend $\leftrightarrow$ 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 graphs to be saved as required.

- **mpld3 -** While MatPlotLib can generate graph images and interactive graphs when using a framework such as Qt, without these frameworks it cannot alone export interactive graphs like those that we need to display the signals. We use mpld3 to generate these interactive graphs from inside Python. However, the same issue was encountered with this library as detailed above in the MatPlotLib package - it was trying to access the 'js' package inside a Web Worker. Once again to solve this, we changed the backend that mpld3 used, which prevented the problem.

- **emd -** While STFT functionality was baked into SciPy, we had to find another package capable of performing EMD analysis. In theory, this was easy, as the 'emd' package on PyPi - a large Python package repository - had the 'py3-none-any' tag. This meant that the wheel had no particular abi restrictions, and could be used on any platform - in general this meant that inside the package there was only pure Python code, and no C source code. This was needed, as mentioned in the Pyodide Description, as any wheel with pure Python code could be run in Pyodide with no modification, where as a wheel containing C code would need to be compiled separately. However, 'emd''s dependencies did not contain only pure Python code. Following the chain, 'emd' required 'sparse', which implemented n-dimensional sparse arrays, but again was all Python code. 'sparse' then required 'numba', which was a just-in-time (JIT) compiler for numerical functions in Python. This is where the problem lay, as this JIT compiler made use of 'LLVMLite' a lightweight python binding for writing JIT compilers, which made use of 'LLVM', a collection of reusable compilers and toolchains. Attempting the compilation of 'LLVMLite' to WebAssembly ourselves, and then searching through troves of online information, we found that through our own attempts, and the attempts of numerous other users, that this process would prove incredibly difficult.

  We decided to start back at the drawing board instead, and delve into the 'emd' package code, and find where the 'sparse' package was being used. We found that the only place that it was being used, was for further spectral analysis than was required in the scope of this project. So as such, we decided to strip out the requirement for 'sparse', and repackage the whole 'emd' library and distribute it directly via our system (allowed under the license GPLv2+).

- **Hilbert-Huang Transform (HHT) -** To perform the additional HHT and produce a resulting spectrogram, a custom script was made contained in `FILLINNAMEHERE`, which makes use of the HHT code inside 'SciPy'.

  We might need to reme
  implement it

- **JSON Bottleneck -** While making use of JSON means that we do not face any problems with object type auto-translation between Python and JavaScript, it was found during testing that this serialisation and deserialisation to and from JSON is a considerable bottle neck in the speed of the system, especially considering that whole arrays of thousands of samples from time-series are being converted. If performance was more of a focus of this application, and more time was at hand, this problem could be avoided using the provided 'JsProxy' and 'PyProxy' objects, however as performance was not a focus, this has been left for future developers to implement if desired.

- **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 session storage as a replacement, simply involving writing JavaScript handling code.

3.1.3 How Pyodide fits in with the rest of the system.

Hopefully now through the previous sections it is clear that while Pyodide has some hurdles that must be jumped, it provides an excellent staging ground for this system to launch from. It must now integrate with the rest of the client side web stack, being executed for example through JavaScript events from button presses, or returning data through post messages to be added to the page DOM.

To keep things simple, we have a single Web Worker running Pyodide inside, setup importing the required packages, and implementing any fixes to the caveats in the system as detailed above. This single Web Worker is setup to accept and return post messages, a method of communication between threads in JavaScript. An initial JSON post message is sent to the Worker with signal information (stored in session storage) and analysis type. This JSON message is then forwarded into the Pyodide code, where Python decides which method of analysis should be performed, generates the signals from input parameters, combines said signals, and finally performs the analysis. The result of all this Python code is a string of HTML code, that is returned once more through a post message from the Web Worker to the main thread, where it is injected into the DOM.

This remaining web stack makes use of Bootstrap material design to provide a consistent and modern design to the front end, detailed further in the Front-end interface section. Pyodide graphs are then generated in HTML and injected directly into the DOM, making use of the mpld3 package. Alongside this, the whole system only uses plain JavaScript for simplicities sake.

3.2 Front-end Interface

The front-end interface is built so that any user can experiment and perform EMD and STFT analysis both on their own uploaded data and on data that they have generated through this system.

The bulk of the front-end is contained with in the `src/index.html` file. Inside the `<head>` of the document, the styled version of bootstrap is loaded, alongside our bundled `main.bundle.js` JavaScript code. The system makes use of webpack to bundle all JavaScript code into a single file, alongside optimising and removing any duplication. In the remainder of the `<body>` of the document, bootstrap's buttons and navigation constructs are used to create the navigation bar, along with option buttons. Further in the document, the modal containing information on the types of signal that the user can generate is created. This is created using an accordion class from bootstrap, allowing for a drop-down menu for each signal type. Inside these signal types, details on the available input parameters are also listed, providing clear information to the user on what their choice of input signals will achieve.

Moving on to the `main.js` file, the first step is loading in all JavaScript modules and their relevant functions into the `main.js` file. Using this module and loading system means modularity can be implemented during development, making code easier to understand, test, and refactor, while using `webpack` as mentioned previously to bundle our code together when it is served to the end user. Next, document `querySelector`'s are used to find, and then attach click event listeners to the required user-interface buttons. Using this method means as much arbitrary code as required can be executed upon button press, such as saving bookmarks or setting the signal analysis method. Alongside this, the `DOMContentLoaded` event is attached to. This event is only called when the page has completed loading, and means that it is possible to manipulate the DOM in any way required, creating a `promise` that the page is ready. Once this promise is fulfilled, previously mentioned module classes are instantiated, such as `InfoPanel`. These classes expose functions that generate or modify content in the DOM. In the example of `InfoPanel`, the `displayInfoPanel` function is exposed, taking a boolean argument on whether it should be displayed or not. This class and function also grab data from a separate JSON file named `methodInfo.json`. When required to be displayed, the relevant information from the `methodInfo.json` file is displayed to the user, providing advantages, disadvantages, and a description of each signal analysis type.

3.2.1 Input Selection

The input selection is shown as two different tabs on the front end namely 'Upload Signal' and 'Create Signal'. These two different tabs are implemented through modules `UploadSignal.js` and `ConfigureSignal.js`. Uploading signals is mainly covered through the fileIO module for file processing and passing through the signal data as the front-end simply includes the file input element and the button to generate the graphs as there are no other configuration settings involved with this method other than choosing the analysis method. Configuring signals within the application is implemented through dynamic forms created for each of the signal types as they all required different parameters to create. These are loaded in through HTML templates using the ES6 syntax and create a button or 'chip' for every signal produced so they can be edited and deleted as necessary. These actions all update the URL as required to ensure it is accurate when the user bookmarks the application state. All the form text inputs have been limited to numerical values though the required attribute has not been able to be implemented as there were issues with Bootstrap custom forms and using browser defaults. There is also a help link on the signal form to give more information on the signal types units. The combination method selector includes a tool-tip to explain that combining via product should be limited to certain data types as it was decided that we should not restrict the user experimentation as much as possible.

3.2.2 Bookmarking

As part of our original specification we were tasked to create a method of saving and sharing the users application settings and signal configurations. The implemented solution is based off creating a bookmark feature that copies the current URL to the user's clipboard. Bookmarking is handled within the `bookmark.js` module and is used to update the URL parameters with the users settings and choices through the UI. In order to consistently update the URL when an analysis method is chosen or signal parameter is added or edited the URL is updated on each button click which triggers a UI change for that action as well as saving the URL string within session storage. This string can then be processed to create an object that can be used as the input data for the graphs. There are different methods for dealing with the signal parameters as the keys follow a format of `[signal number]-[signal parameter name]=[signal value]` in order to differentiate and identify the different signal's values within the URL.

3.3 File IO

3.3.1 Main Logic

The application supports users supplying their own input data for both EMD and STFT analysis.

This functionality is achieved in a separate class in the source code located at **src/fileIO.js**.

The constants required by the class, and those that reference it, are declared at the beginning. These include an array of supported file types (this is currently just **.csv**, but utilizing an array will make it easier in the future to add more supported file-types) and a **csvDataKey**, which is used as the key to store uploaded csv data in the browser's sessionStorage.

In typical use-cases, the class is invoked via the **parseFile()** method, which will attempt to read a .csv file selected by the user. There is a safety check to verify that a file has indeed been selected by the user, and the *'Generate Graphs'* button hasn't just been pressed without something to process: if that is the case, the method will safely return early, without an error.

After verifying that a file has indeed been selected by the user, there is another safety check to validate that the type of file selected is supported; this is in addition to specifying this as a parameter of the HTML input option in **src/modules/uploadSignal.js**. If more file types are to be supported in the future, this parameter should reference the static *supportedFileTypes* array mentioned earlier to reduce tediousness of the work, as only one file will need to be altered to allow for more file extensions to be accepted. If, for some reason, a file-type has slipped through the cracks and this check fails, an alert will be displayed informing the user that the given file-type is not supported, and to try again. The method will then return early.

Finally, just before the file is read in, there is a check to determine whether *csvDataKey* in browser storage is currently populated; if it is, then it means it's data from a previous upload and should be cleared before progressing.

Afterwards, a new **FileReader()** object is created to read in the supplied file. Error/Progress/Abort/Load handlers are all attached to the object to ensure robustness. Once the *'loadend'* event is triggered, a call is made to the **loaded(target) function**. This is the function responsible for actually parsing the data read in by the file reader.

Upon being called, the *loadend* method will change the status displayed to the user as "Finished Loading!" and will then call the **csvToArray** method to convert the text which has just been read in to a JavaScript array. Once finished being converted, the resultant array is then converted into a JSON string and stored in sessionStorage under the *csvDataKey* key. The object is converted to a JSON String to preserve the structure of the data (without this, the '[' and ']' are lost, making the data much harder to parse) and for consistency with other classes utilizing sessionStorage.

Originally, the **csvToArray** method invoked an external library called *convertCSVToArray* which handled all the transformation logic. This worked quite well for relatively small CSV files, <1MB in size, however the performance was unsatisfactory for larger file inputs (a browser's given session storage limit is 5MB per application) and therefore a new solution was required to accomplish this requirement. It was decided instead to create a custom, efficient csv parsing method which requires a specific format (discussed in 3.3.3). This csv parsing method reads from the csv file in memory, splits the headers and remaining rows into separate variables, and then loops through each row, returning an array of 2 values each time, split via the ',' delimiter in the csv file. Ultimately this returns a 2D array containing the x,y values of the entire time series. Any rows containing values which are not strictly numerical are removed from the final array by filtering based on the return value of JavaScript's **isNaN(value)** (isNotANumber(value)) method; thankfully this method accounts for values containing exponents (numbers such as 0.0314E+2) so they will be included in the final array.

After this process is completed, the data contained in the supplied file is available to be read by any class that requires it via the *csvDataKey* key in the browser's sessionStorage.

3.3.2 Helper Class

A small class was created to assist with reading and writing to the Browser's sessionStorage. This class is located in **src/helpers/sessionStorage.js**.

A disclaimer is at the top of the class notifying which browser versions support session storage (and thus the functionality to upload files). The remainder of the class is then populated with getter/setter methods as well as a method to check if a value is associated with a given key.

3.3.3 CSV File Format

This section provides guidance for how to structure your .csv file(s) so that they can be read by the application.

Although the title of your headers doesn't particularly matter, their number and ordering does. The application expects your headers to follow the convention **Time, Value**, such that the first value represents the time of the event (your X axis value), and the second represents the value at the given time (your Y axis value).

An example is provided below:

Time	**Value**
0.0	0.0
0.00628947	0.00012583
0.01257895	0.00050233

3.4 Generating Graphs

The process of generating graphs, generating signals, combining signals, and performing analysis, is all performed in one large swoop, involving heavy use of Pyodide and Python, and is subsequently outlined in the following section.

Prior to the journey beginning, when the page is initially constructed an instance of the Pyodide Web Worker must first be created. This Web Worker creates a Pyodide Python interpreter inside it, and installs the Required Packages, using both micropip and the `pyodide.loadPackage` methods. Once these packages have been loaded, and the Worker is satisfied that it is ready to receive data, it assigns a positive value to the `pyodideReady` boolean, signalling to the rest of the code that the Worker is ready.

The Web Worker then sits idle, until the pressing of the `Generate Graphs` button present in the User Interface. This button press calls an event, which in turn makes a chain of function calls. First, it calls the `displayLoadingGraphs` function contained inside the `graphs.js` file. This function creates the placeholders for the graphs in the DOM, and adds a loading animation to the placeholders to tell the user that there is some background processing occurring.

Next, the `handleCallPyodide` function from `pyodideHelpers.js` is called, which creates in turns calls the `evaluatePython` function, while creating a promise on said function to return some data for future processing. This `evaluatePython` function creates a promise, which when resolved correctly, sends a post message to the previously mentioned Pyodide Web Worker containing the method that the data was created (uploaded or generated using the system), the analysis method to be performed, and if the signal was generated using the system, the information about the signal. The Web Worker then takes these input parameters, and decides which functions need to be called. Error handling is baked into this process too, ensuring that the Web Worker is ready to receive data, with data verification taking place ensuring that the session storage is collected and passed through correctly.

If the signal provided was generated using the tool, then the Signal Generation code shall be called, otherwise, the program will skip straight to the STFT or EMD code.

3.4.1 Signal Generation

Signal generation is handled entirely in Python, inside the Pyodide Web Worker's `script.py` file, with the signal data from the user passed in via the `analysis_runner` function call.

This function then parses the input data, and generates signals according to the input parameters from the user, calling the relevant functions inside the `signals.py` file. Inside this file, a separate function, e.g. `simple_sin`, `chirp`, `white_noise`, is created, that accepts the required input parameters, and returns a time series of points over a predefined period. These functions also make use of the Required Packages, specifically NumPy and SciPy, to generate the time-series. Using these separate functions again introduces modularity, and allows modifying and testing of specific components.

3.4.2 Signal Combining

Signal combining is performed in a separate function to signal generation, with each of the initial generated signals from the user selected signals being passed in through a 2D array. These signals are then combined depending on the selected combine type, either sum or product. Both of these functions are almost identical, however a $+$ or $*$ is used depending if the selected method is sum or product respectively.

3.4.3 STFT Analysis

STFT analysis is performed on either user uploaded data or tool generated signals depending on the choice made in the user interface. Either way, the process is the same, where the exposed `stft_analysis` function is called, passing in the required input parameters (be that a combined signal from the tool, or user uploaded data).

This function then returns a number of variables back to the caller. Previously this function made use of the same mpld3 method used in EMD analysis, however the spectrograph implementation was taking too long to generate, and was outputting in too low a resolution. Instead of this, the raw JSON representing a spectrogram is output, and returned through the chain of calls, from `stft_analysis` $\rightarrow$ Web Worker $\rightarrow$ post message $\rightarrow$ `evaluatePython` promise $\rightarrow$ `handleCallPyodide` promise callback. Inside this promise callback, the spectrogram data is then parsed, and fed into the d3 library directly. A custom graph generation JavaScript file contained within `spectrogram.js` is then used, which quickly generates a spectrogram, and replaces the placeholder graphs with this generated spectrogram.

Alongside this spectrogram data, along the same chain the combined signal graph and component signal graphs are also returned from the mpld3 library. These graphs we still generate using mpld3 as line graphs are fast, and still of high quality, when generated using MatPlotLib.

3.4.4 EMD Analysis

EMD analysis is again performed on either user uploaded data or tool generated signals depending on the choice made in the user interface. The process either way is the same, where the exposed `analysis_runner` function is called, passing in the required input parameters (be that a combined signal from the tool, or user uploaded data).

This function then returns a number of variables back to the caller. As EMD analysis returns a list of IMF's for viewing purposes, these graphs can be generated as line graphs over a common time base in mpld3, as line graphs are quick to generate, and their HTML code can be returned through a series of calls. This chain flows as so: from `emd_analysis` → Web Worker → post message → `evaluatePython` promise → `handleCallPyodide` promise callback. Inside this callback, the HTML graphs are parsed, and placed in the appropriate positions according to the placeholder graphs.

Alongside these IMF graphs, along the same chain the combined signal graph and component signal graphs are also returned from the mpld3 library. These graphs we still generate using mpld3 as line graphs are fast, and still of high quality, when generated using MatPlotLib.

4 Installation

Installation of the whole system only requires a preexisting version of Node (version 16 or later) and npm (Node Package Manager).

Note: We only use Node to aid the deployment speed and to use Webpack, and Node does *not* act as a backend for our system. Files are served to the browser with no further communication taking place after receiving.

The following commands are then executed from the root project directory once the project code is cloned into the correct directory:

To install required node packages:

```
npm install
```

To build the project statically in `dist`:

```
npm run build
```

To run a lightweight server that hosts the content on port 8080:

```
npm start
```

5 Maintenance

To keep the over-time functionality of the application without any unexpected crashes, the application should be maintained throughout its lifetime. To do the maintenance we will need to check up on the various packages and tools used in the implementation of the application to make sure that the features used at the time of the development of the application are still supported by the new and more updated packages and libraries. The main libraries that needs to be maintained for this web application are mentioned below

5.1 Pyodide

Pyodide is the backbone of this web application, all of the code is interlinked using pyodide and the libraries used in this application are converted to Web Assembly using Pyodide. The maintenance of Pyodide is the most important to do. To ensure the packages chosen during the development are still compatible with Pyodide will be checked by testing all the key functionalities of the packages that are used by pyodide to ensure the application runs smoothly and to ensure that dependencies for packages that we use all work and if they contain C code, are compiled to web assembly appropriately. The Pyodide website will be checked to see if the modules are still supported or not, as pyodide updates their users on their website of the latest changes to the supported modules. Pyodide will be updated accordingly to ensure everything is working and to improve security risks.

5.2 Graphing

Another important feature of this web application is to have graphing features for the users to interact with. The libraries used for graphing such as d3, mpld3, and MatPlotLib will be also checked for any newer versions, specially d3 as it keeps getting updated versions, some of the code written in an older version might not be supported on the newer version, so its important to keep the version of d3 to be the one that supports the code written during development.

5.3 General checks and updates

Some of the other libraries used in the application such as node will be updates regularly to ensure the latest and updated security and performance. Ensuring long term compatibility with web assembly in case browsers add a permission - similar to the microphone permission - will also be looked out for to avoid any unexpected bugs.

6 Troubleshooting

This section will explain any known issues and how to fix them.

1. Browser not supported: Our application has been tested using Chromium Browsers, specifically Chrome v87 & v99. Incompatibilities were experienced with Firefox (specifically Firefox 78.15.0esr) and Safari (all versions), hence those browsers are not recommended for use.

2. File not supported: Only .csv/.CSV files are supported. We recommend that you follow the format specified in the FileIO Section.

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`.