Signal Processing - MATH 458

Sergei Kliavinek & Maximo Cravero

January 22, 2021

1 Compiling the Program

In order to compile the program, to perform the following steps:

- Clone the repository.
- Add the git submodules for the 'AudioFile' and 'googletest' repositories.
- Create a build directory (to keep all the files that are generated during the build process in one place).
- Run the 'cmake' and 'make' commands to generate the executables.
- Run the executables via './test_signal_processor' and './run_signal_processor' from within the 'build' directory.

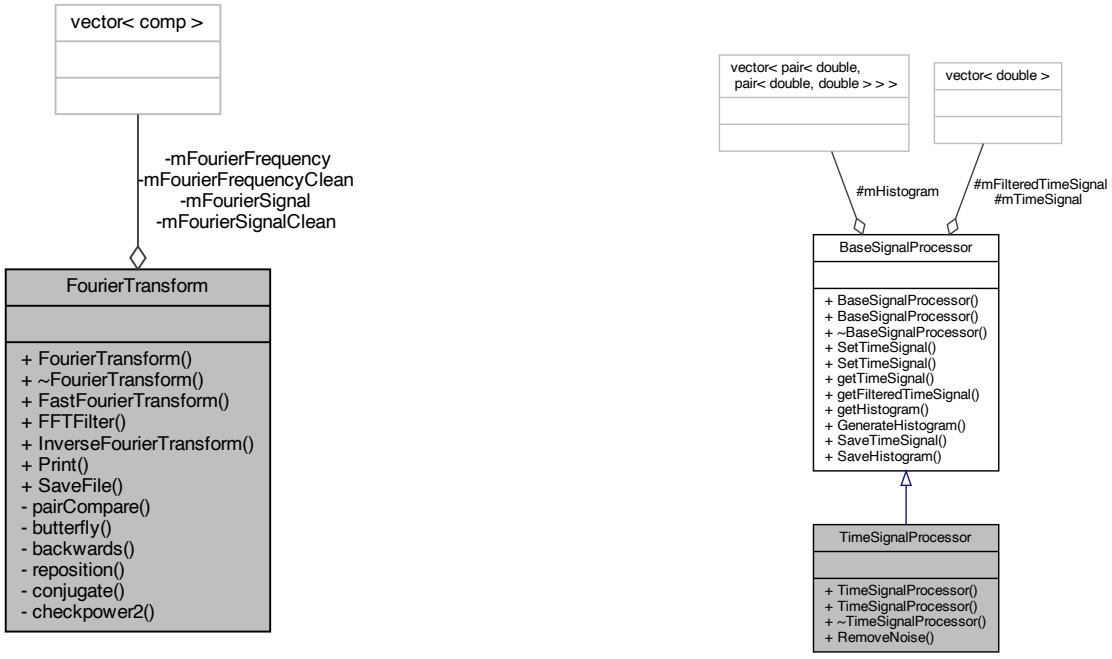
The first executable will run all the tests, which are described in more detail in section 4. The second one will prompt you to input parameters pertaining to both the time and frequency domain signal processing methods that were implemented. The output files will be saved in the 'output' directory, from which plots are generated to visualize the results. This is done in the Jupyter notebook 'plots.ipynb'.

2 Typical Program Execution (Flow)

This repository makes use of an additional submodule called 'AudioFile' [1], which is used to read in the mono WAV files. These are subsequently processed both in the time and frequency domains. The general flow consists of loading in a given audiofile, in this case 'CantinaBand3.wav', filtering the signal, and outputting the results to the output folder for plotting. To verify that the functions perform as they should, the 'googletest' submodule [2] is used to implement various tests.

3 List of Features (Implementations)

The program contains two primary classes, namely the 'FourierTransform' and 'TimeSignalProcessor' classes (where the latter is a derived class of the 'BaseSignalProcessor'). The decision was made to split these two in order to separate the treatment of the signal in the time and frequency domains to avoid any confusion or misuse of their methods. Their respective class diagrams are indicated in Figure 1. More detailed explanations regarding their functionality are provided in the following subsections.



(a) 'FourierTransform' class diagram

(b) 'SignalProcessor' class diagram

Figure 1: Overview of main classes

Fourier Transform Class

The 'FourierTransform' class contains methods to perform the Discrete Fourier Transform (DFT) via means of the Fast Fourier Transform (FFT), which is an $\mathcal{O}(n \log n)$ algorithm as compared to the DFT's $\mathcal{O}(n^2)$ running time. Likewise it contains a method to perform the inverse FT. These two methods can be used in conjunction with an additional class method to filter out certain frequencies, either to remove high frequency noise or otherwise. Public methods of this class are methods of basic functions - the forward Fourier Transform (**FastFourierTransform**), inverse Fourier Transform (**InverseFourierTransform**), Fourier Filter (**FFTFilter**), and save to file and print to screen (**Print**, **SaveFile**). The private fields of class **FourierTransform mFourierSignal**, **mFourierFrequency**, **mFourierSignalClean** and **mFourierFrequencyClean** contain the input signal, its Fourier waveform and the filtered signal and Fourier waveform respectively. These are accessed using the output functions. Private functions are auxiliary functions used exclusively within basic Fourier transform methods.

A classical algorithm has been chosen as the algorithm for the Fast Fourier Transform. It consists of the following steps:

- Preliminary permutation of array elements using a binary representation of their number (recursive division into even and odd pairs of elements).
- Sequentially rearrange pairs of these elements using the butterfly transformation (the transformation is performed crosswise for pairs of elements).

For the inverse Fourier transform, a direct Fourier transform of a compound conjugate signal was performed, which was then also complexly conjugate.

There are a large number of Fourier-Filtering algorithms. In this paper, the following algorithm was chosen:

- Set a passband denoting which part of the signal as a percentage of the total intensity (sum of the squares of the amplitudes across all frequencies) will be considered significant and which part of the signal will be discarded as noise.
- A fast Fourier transform is performed
- In the frequency space, the amplitudes recognized as noise are rejected.
- Inverse Fourier transform is then performed, returning the signal that has already been discarded.

To solve the above problems, **vector** < **complex** < **double** >> was chosen as the basic signal unit. The analyzed signal is in general complex and it is convenient to store it in an array. The **vector** class was chosen because it allocates and deallocates memory automatically and makes the programmer's life much easier.

Time Signal Processor Class

The 'TimeSignalProcessor' class on the other hand deals with processing the signals in the time domain. It contains methods to generate an intensity histogram and denoise the raw time signal.

It was decided to generate a base class from which to inherit commonly used methods or member variables. In this way the repository can be expanded in the future by implementing other derived time signal processing classes. The 'BaseSignalProcessor' class contains a custom constructor which allows it to be initialized with an AudioFile signal which is directly set to the 'mTimeSignal' member variable. It contains a destructor method which clears all the protected member variables, namely 'mTimeSignal', 'mFilteredTimeSignal', and 'mHistogram'. To facilitate the use of the class, it contains an overloaded 'SetTimeSignal' method such that signals may be set both from AudioFile objects or regular vector doubles.

Additionally it contains a method to generate histograms from the 'mTimeSignal' variable. The implementation consists of sorting the original time signal, and through the use of a counter to keep track of how many samples we have gone through and some conditional statements, it computes the bin frequencies and outputs them as a vector triplet, particularly 'vector<double, pair<double, double> > >'. This stores the left and right bins, as well as the frequency. An overview of the base signal class is provided in Figure 2.

The 'TimeSignalProcessor' class is then a derived class of the above base class. It makes use of the custom constructor from the base class described above. In addition it implements a moving average and exponential moving average filter, which can be selected via means of an input flag indicating which one you prefer to use. The moving average function is implemented such that its run time complexity is independent of the size of the window (except for the initialization), as only two additional computations are performed at each step to determine the moving average at the following step.

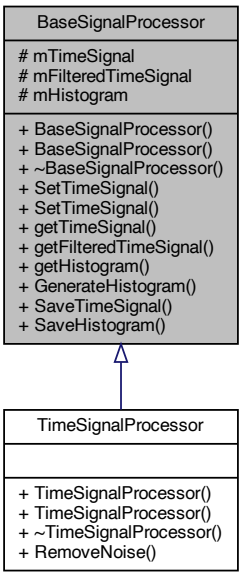


Figure 2: Base Signal Processor Class Diagram

4 Validating Tests

The tests were done for small known signals of length in 8 characters. This is enough to test the performance of the code, but the tests are time-efficient and easy to interpret. The **SciStatCalc** was used to select the tests. The written tests check the direct Fourier transform (**fft_1**, **fft_2**), inverse Fourier transform (**inverse_fft_2**, **inverse_fft_2**), also the Fourier filter for a simple signal with small amplitude for one of the frequencies (**fft_filter**) are checked. In addition, **run_file** has been written, which demonstrates how the basic functions work on a large audio file.

The tests performed for the 'TimeSignalProcessor' consist of dummy tests where a simple input vector is used to check that the output of the moving average method has the right size and right inputs. Similarly the histogram method is tested by checking that the bin frequencies are coherent with a simple example, and that they sum up to the total size of the input vector.

5 TODOs (Limitations of current project) & Final Remarks

Future updates to this repository could include the following: Support for stereo WAV files, additional filters for denoising (time signal), FFT capabilities for signals whose size is not a power of 2.

References

[1] *AudioFile git repository.* URL: `https://github.com/adamstark/AudioFile`.

[2] *Google Test Suite.* URL: `https://github.com/google/googletest`.