Master 1

Mention Mathmatiques et Applications

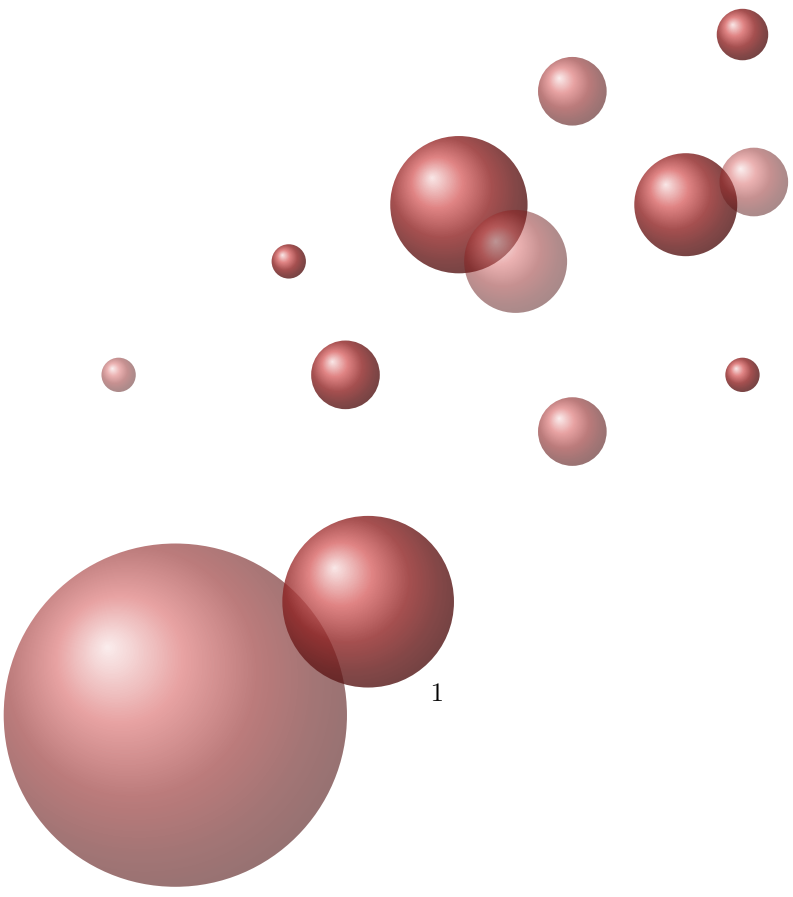
SIGAL

C++ Signal Analysis Toolbox

Randall
BALESTRIERO

randallbalestriero@gmail.com

Anne 2014 – 2015



LaTeX

Contents

Introduction

With the computational power available today, machine learning has begun a very active field finding its applications in our everyday life. One of its big challenge is classification involving data representation (the preprocessing part in a machine learning algorithm). In fact, classify linearly separable data is feasible (a simple perceptron can do it). The aim of the whole preprocesing part is to obtain well represented data by mapping raw data into a feature space where simple classifiers can be used efficiently. For example, everything around audio processing uses MFCC until now. This toolbox gives the basic tools for audio representation using the C++ programming language also providing an implementation of the Scattering Network which brings a new and powerful solution for these tasks. Furthermore, the use of this toolbox is not limited to machine learning preprocessing. It can also be used for more advanced biological analysis such as animal communication behaviours analysis or any biological study related to signal analysis.

1 WAV

1.1 File Structure

The WAV file is an instance of a Resource Interchange File Format (RIFF) defined by IBM and Microsoft. Thee header part of this file is made of complementary chunks describing the architecture of the wav allowing easy information storing. Let's see how these chunks are organized in a WAV file :

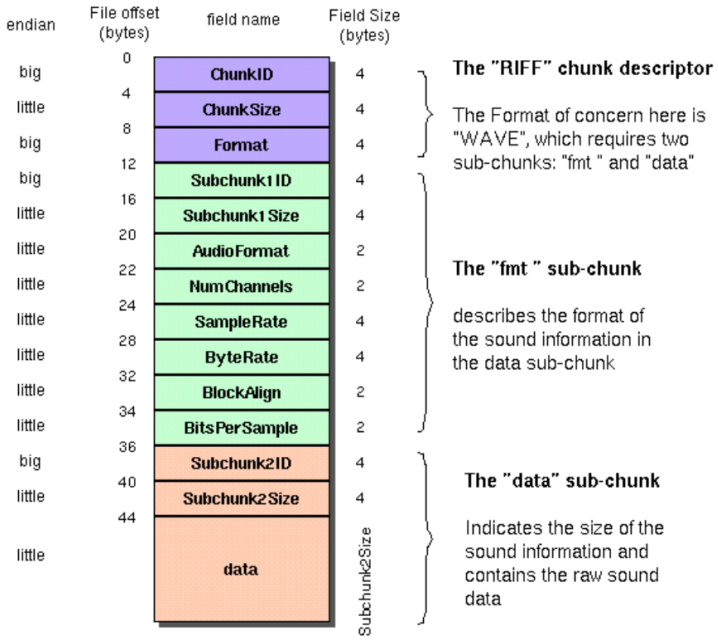


Figure 1: The Canonical WAV file format `https://ccrma.stanford.edu/courses/422/projects/WaveFormat/`

The file is made of three main chunks each having a specific role that we will describe here.

- ChunkID identifies the type of the first chunk with four characters : "RIFF".
- ChunkSize is the size of the file left from this point. It will be 36 (sum of the other chunks sizes) plus Subchunk2Size and this can be easily seen by summing the different sizes on the right of the header representation.
- Format will be four characters : "WAVE" (this allows us to check if we are really reading a wav file during the process).

- SubChunk1ID identifies the second chunk. It is a four characters name : "fmt " and starts the data description block.
- SubChunk1Size is simply the size of this block which is 16.
- AudioFormat, also called Format tag, is the option indicating the data compression used. It is almost always equals to 1 which stands for : no compression is used.
- NumChannels is the number of channels (1 for mono and 2 for stereo).
- SampleRate is simply the number of samples per second, the frequency.

- ByteRate is the average number of bytes per second, this can be found with the following formula : SampleRate $*$ NumChannels $*$ $\frac{\text{BitsPerSample}}{8}$.
- BlockAlign won't be necessary for us. It can be computed with a formula : $NumChannels * \frac{\text{BitsPerSample}}{8}$.
- BitsPerSample can either be 8 or 16 but in general the later is used.

- SubChunk2ID identifies the last chunk block, it is made of the four characters : "data".
- Subchunk2Size is the size of the file left which is just the size of the data since this is the last thing not seen yet..
- Data is finally the values of the signal in the standard pulse-code modulation representation.

1.2 Implementation

The use of the built-in class WAV is simple, the only thing to provide is the name of the wav file. This can be done during the instantiation of the class of at any other time. Let's look at an example.

```
WAV<double> Signal("mysignal.wav");
WAV<double> Signal2;
"mysignal.wav">>Signal2;
"mysignal2.wav">>Signal;
```

Note that the template parameter can be ignored leading to the default value : float. One instance of the class can be used for different wav files which can be useful. This WAV variable allows easy interactions and can provide informations about the loaded file :

```
Signal._Size;
Signal._SampleRate;
Signal_NumberOfChannel;
Signal[0];//return the first value of the loaded signal
```

Finally, to export the loaded file two options are available. Firstly, it is possible to export it into a txt file which only export the signal data disregarding all the other informations. This loss can be avoided using a special method which export the data back into a wav file, with the following syntax :

```
    Signal>>"newsignal.txt";//to .txt
Signal.PrintWav("newsignal.wav");//to .wav
```

With this implementation, WAV can be seen as a special type. Note that during the loading, no normalization is used. In fact, only the user can define the normalizing constant he is interested in (max, L^2-norm,...). Finally, for an external use of this toolbox, one should not need to use this class since it is just here as a input/output convenience for the other algorithm we will now describe.

2 Fourier Transform

2.1 Definitions

A sinusoidal wave is characterised by three parameters: amplitude, frequency and phase.

- The amplitude is the amount the function varies, positively or negatively, from zero in the y direction.
- The frequency is how many complete cycles there are of the wave in unit distance on the x axis (which often measures time)
- The phase is relevant when comparing two waves of the same frequency. It is how much (measured in degrees or radians) one is shifted relative to the other on the x axis.

This terminology comes from sound engineering where higher frequency sounds have higher pitch and waves of greater amplitude are louder. As an alternative to specifying the frequency, the number of cycles in unit distance, we can instead specify the wavelength, the length of one cycle. The higher the frequency, the shorter the wavelength. The lower the frequency the longer the wavelength.
The Nyquist frequency is the maximum frequency that can be detected for a given sampling rate. This is because in order to measure a wave one needs at least two sample points to identify it (trough and peak). We will abbreviate the continuous Fourier transform with CFT and the discrete Fourier transform with DFT.

Interpretation of the CFT Using the Euler's formula, we can see the Fourier Transform as a decomposition of a signal into complex sinus using convolutions.

$$e^{ix} = \cos(x) + i\sin(x)$$

$$\begin{aligned}
\hat{f}(\xi) &= \int_{-\infty}^{\infty} f(x)e^{-2\pi ix\xi}dx \\
&= \int_{-\infty}^{\infty} f(x)(\cos(-2\pi x\xi) + i\sin(-2\pi x\xi))dx \\
&= \int_{-\infty}^{\infty} f(x)\cos(-2\pi x\xi) + f(x)i\sin(-2\pi x\xi)dx \\
&= \int_{-\infty}^{\infty} f(x)\cos(-2\pi x\xi)dx + \int_{-\infty}^{\infty} f(x)i\sin(-2\pi x\xi)dx
\end{aligned}$$

2.2 Fast Fourier Transform

We will now denote x_k as the k^{th} value on the signal in the time space and X_k the k^{th} value of the signal in the frequency domain, N will denote the length of the signal. A length of N means the indices range from 0 to $N-1$ using the standard counting start at 0.
The fast Fourier transform (FFT) is a instance of DFT which is able to perform the DFT in $O(n\log(n))$ complexity. The DFT formula using the Twiddle Factor notation :

$$\forall k \in \mathbb{Z}, X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi ikn}{N}}$$

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}$$

As we can see, we need to perform N operations for each $X_k, k \in \{0, 1, ..., N-1\}$ thus we are in $O(N^2)$. It is possible to use the scaling factor :$1/\sqrt{N}$ in order to have an unitary operator (Parseval's theorem) which implies that the sum (or integral) of the square of the function is equal to the sum (or integral) of the square of its transform. In order to go from N^2 operations to $N\log(N)$ operations, three main concepts have to be defined :

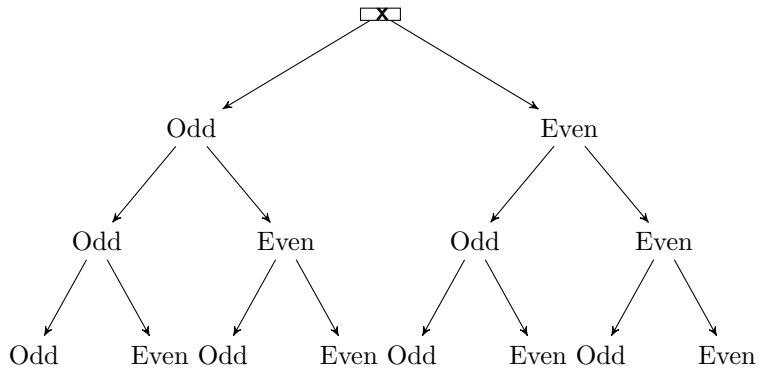
- The Danielson-Lanczos Lemma
- The Twiddle Factor properties
- The Butterfly Scheme

2.2.1 Danielson-Lanczos Lemma

This theorem is the foundation of the FFT by allowing a divide and conquer strategy. In fact, we have the following relations :

$$\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{\frac{-i2\pi 2kn}{N}} + x_{2n+1} e^{\frac{-i2\pi(2k+1)n}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{\frac{-i2\pi kn}{N/2}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{\frac{-i2\pi 2kn - i2\pi n}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{\frac{-i2\pi kn}{N/2}} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{\frac{-i2\pi kn}{N/2}}
\end{aligned}$$

For every X_i we can now divide the N sums into two different summation group (Even and Odd). Note that for the special case $N = 2$ the sums are removed and n is replaced by 0 which means that we are left with a simple linear combination of the input signal and the Twiddle Factor. If we apply this recursively we obtain the following architecture :



And now for any given input size we are able to break it done into a linear combination of the input signal with twiddle factors. For example, if $N = 4$ we

have after full decomposition :

$$X_k = x_0 + W_2^k x_2 + W_4^k x_2 + W_4^k W_2^k x_3$$

And for $N = 8$:

$$X_k = x_0 + W_2^k x_4 + W_4^k x_2 + W_4^k W_2^k x_6 + W_8^k x_1 + W_8^k W_2^k x_5 + W_8^k W_4^k x_3 + W_8^k W_4^k W_2^k x_7$$

This puts a constraint though, the signal length has to be a power of 2. The number of decomposition is thus $\log_2(N)$. If the signal size is not a power of 2 it is necessary to use zero padding (add as may 0 as necessary at the end of the input). Note that padding with 0 in time domain leads to an interpolation of the FFT. Middle zero padding the FFT (in the frequency domain) interpolates the IFFT (time domain). Periodizing in the frequency domain implies sub-sampling in the time domain (this will be useful for the Scattering Network). One last thing to notice here is the order of the input values in the decomposition. Because of the nature of this decomposition (even/odd) we end up with the x terms being rearranged in a specific order : the bit-reversal order. This can be found by taking the symmetric of the binary position of the input value as seen in this little example for $N = 8$:

$$\begin{aligned}
0 : 000 &\rightarrow 000 : 0 \\
1 : 001 &\rightarrow 100 : 4 \\
2 : 010 &\rightarrow 010 : 2 \\
3 : 011 &\rightarrow 110 : 6 \\
4 : 100 &\rightarrow 001 : 1 \\
5 : 101 &\rightarrow 101 : 5 \\
6 : 110 &\rightarrow 011 : 3 \\
7 : 111 &\rightarrow 111 : 7
\end{aligned}$$

2.2.2 Twiddle Factor Properties

Complexity has already been broken down but we can still optimize the implementation by exploiting the Twiddle Factor properties using roots of unity properties. In fact we have :

$$W_N^k = e^{\frac{-i2\pi k}{N}} = \cos(2\pi k/N) - i\sin(2\pi k/N)$$

Thus for $N = 2$:

$$\begin{aligned}
W_2^0 = W_2^2 = W_2^4 = ... \\
W_2^1 = W_2^3 = W_2^5 = ...
\end{aligned}$$

And for $N = 4$

$$\begin{aligned}
W_4^0 = W_4^4 = W_4^8 = ... \\
W_4^1 = W_4^5 = W_4^9 = ... \\
W_4^2 = W_4^6 = W_4^{10} = ... \\
W_4^3 = W_4^7 = W_4^{11} = ...
\end{aligned}$$

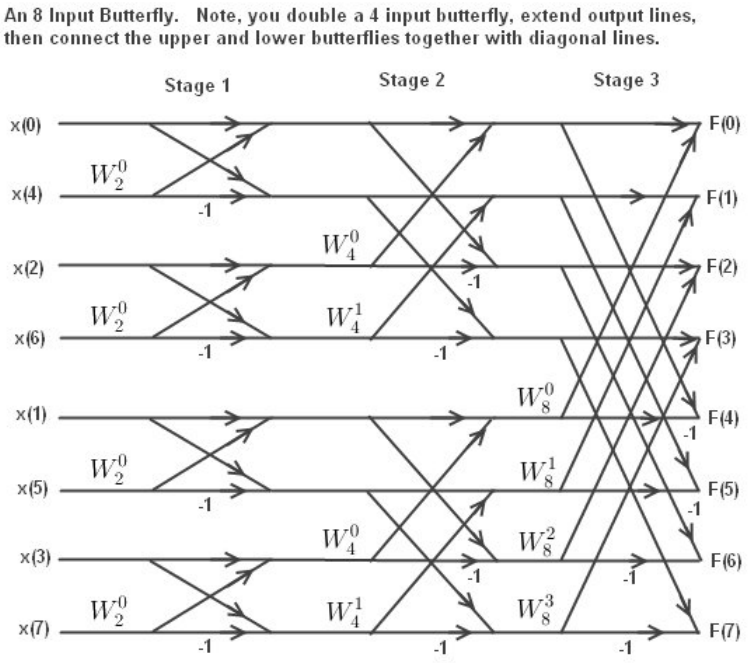
And so on using trigonometric properties, with functions here being $N\pi$-periodic. So using this will allow us to perform less Twiddle Factor computation at each level.

2.2.3 Butterfly Scheme

Finally, the last brick is the following butterfly scheme allowing an in-place FFT which is memory friendly.

2.3 Implementation

Firstly, our implementation is made of three nested loops, the main one which will go through the $\log(N)$ levels of decomposition. The second one will go through the blocks inside a specific level (the last level as 1 block whereas the first level as $N/2$ blocks). Finally the last loop will go inside a block (a block on the first decomposition level will have size 2 while the block in the last decomposition level will be of size N). For each level (i), only 2^i Twiddle factors are computed in the main loop where i is the decomposition level from 0 to $\log(N) - 1$. A simple temporary variable is used in order to perform the swapping operations. Here is an instance of this implementation for $N = 8$ and a human friendly output explaining the performed steps.




```
Level : 0
 W[0]=W(0,2)
  Block : 0
    signal[1]*=W[0]
    tmp=signal[0]
    signal[0]+=signal[1]
    signal[1]=tmp-signal[1]

Block : 1
   signal[3]*=W[0]
   tmp=signal[2]
   signal[2]+=signal[3]
   signal[3]=tmp-signal[3]

Block : 2
   signal[5]*=W[0]
   tmp=signal[4]
   signal[4]+=signal[5]
   signal[5]=tmp-signal[5]
```

```
Block : 3
  signal[7]*=W[0]
  tmp=signal[6]
  signal[6]+=signal[7]
  signal[7]=tmp-signal[7]
```

```
Level : 1
 W[0]=W(0,4),W[1]=W(1,4)
 Block : 0
    signal[2]*=W[0]
    tmp=signal[0]
    signal[0]+=signal[2]
    signal[2]=tmp-signal[2]
    signal[3]*=W[1]
    tmp=signal[1]
    signal[1]+=signal[3]
    signal[3]=tmp-signal[3]

 Block : 1
    signal[6]*=W[0]
    tmp=signal[4]
    signal[4]+=signal[6]
    signal[6]=tmp-signal[6]
    signal[7]*=W[1]
    tmp=signal[5]
    signal[5]+=signal[7]
    signal[7]=tmp-signal[7]
```

```
Level : 2
 W[0]=W(0,8),W[1]=W(1,8)
 W[2]=W(2,8),W[3]=W(3,8)
 Block : 0
    signal[4]*=W[0]
    tmp=signal[ 0 ]
    signal[0]+=signal[4]
    signal[4]=tmp-signal[4]
    signal[5]*=W[1]
    tmp=signal[1]
    signal[1]+=signal[5]
    signal[5]=tmp-signal[5]
    signal[6]*=W[2]
    tmp=signal[2]
    signal[2]+=signal[6]
    signal[6]=tmp-signal[6]
    signal[7]*=W[3]
    tmp=signal[3]
    signal[3]+=signal[7]
    signal[7]=tmp-signal[7]
```

Note that Twiddle factors are computed at the start of each main loop computing the needed values which are then reused throughout the blocks, meaning that for the first level only one value is computed and then reused all along the blocks. Here is an example of the use :

```
WAV<> wav("signal.wav");//load a wav into float type array
fft<> signalfft(signal.ptr(),signal._Size);//default padding option=1
signalfft.ComputeFFT();
signalfft.ComputeIFFT();//get back to the original signal
signalfft[2];//access the second coefficient
signalfft>>"signalfft.txt";//export it
wav<<"processed.wav";//load a new wav
signalfft.ComputeFFT(wav.ptr(),wav._Size);//perform a new FFT
```

Note that the parameters of the fft class are by default float and float, the first one stands for the type of the input signal and the latter for the coefficients type (complex¡float¿). Finally the padding option which by default is 1 can be set to 0 if the user is sure that the given signal is already a power of 2, this force to skip the padding part resulting in faster computation. Also the coefficients are stored as complex type even after having performed an IFFT meaning that one needs to use a typecast to retrieve the original float signal for example.

2.3.1 Inverse Fourier Transform

In order to simplify the algorithm we sill use the following formula :

$$IFFT(x) = \frac{1}{N} conj(FFT(conj(x)))$$

2.4 FFT Graph

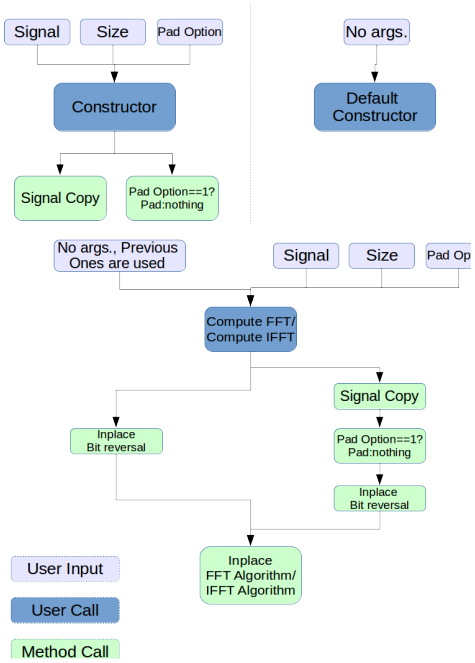


Figure 2: FFT Summary Diagram

3 Spectrogram

Each X_k is a complex number that encodes how strongly the oscillation at this frequency is represented in the data but by doing an FFT we loose the time component. A useful tool is the spectrogram allowing to retrieve part of the time information. The main idea is to perform multiples FFT on a signal each one being located enough in time so the frequency information gained by the FFT can also be linked to a more or less specific time position in the signal. Note however that precision in both time and frequency is impossible to get but depending on the needs one can choose which one to enhance by modifying the size of the considered window. Larger window gives better frequency resolution but less time precision and vice-versa. It is easy to picture the fact that smaller windows are better for the high frequency part allowing good time precision while for low frequency a larger window has to be used for being able to capture it. This problem is lessen in wavelet decompositions and thus the scattering network since this window size is not constant any more.

3.1 Algorithm

Conceptually a spectrogram is computed with the following scheme :

- splitting the signal into overlapping (or not) parts of equal length defined by the user.
- Applying to each of these chunck a windowing function (typically hanning or hamming) in order to remove artefacts by periodizing the function so the limit points (start and end of the chunk) are equal. This part is called apodization
- computing the FFT on each of these chunks
- for each of computed FFT, taking the absolute value of the coefficients will give the columns of the spectrogram.

The windowing is needed since the FFT computation presumes that the input data repeats over and over. This is important when the initial and final values of the data are not the same because the discontinuity causes artefacts in the spectrum computed by the FFT.
In addition, in this toolbox, only the first half of the FFT coefficient are put into the spectrogram thus avoiding symmetrical redundancy. This is due to the fact that our input signal is real and so the second half of the FFT coefficients are simply the complex conjugate of the first half, since in the spectrogram we display the absolute value of the coefficients, we get symmetric FFT coefficients about the middle point.
Most window functions afford more influence to the data at the center of the window than to data at the edges, which represents a loss of information. To mitigate that loss, it is common to use overlapping in time (usually 50%).

3.2 Implementation

It is important to note that the spectrogram (2D-matrix) is stored by column and not by line for faster computation. In fact, during the spectrogram calculation we need to access this matrix column-wise. The operator [] returns the column while the operator () takes two arguments and return the corresponding value in a normal way. Let's look at an example :

```
spectrogram<> b("signal1.wav",256,0.25);//default window function : hamming
WAV<> wav("signal2.wav");//load another wav
b.Perform(wav.ptr(),wav._Size);//compute spectrogram given these new entries
b>>"lifespectro.txt";//write the matrix into a txt file
    b[1][0];//second column,first element
    b(0,1);//first line second element same result as above
```

The template parameter defines the coefficients type. The default value is float. Also note that no transformation is performed after the absolute value is computed, which means that if one want to apply a logarithmic function (most common) this has to be done manually after computation.
The apodization can be done using one of the available windowing function :

- hamming
- hanning
- triangular
- hann poisson

but can also be used with a specific function and passing its pointer as the last argument of the spectrogram class.

3.3 Graph

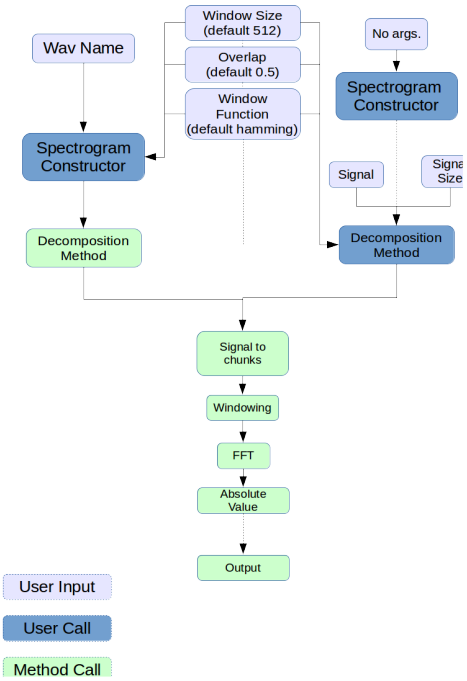


Figure 3: Spectrogram Summary Diagram

3.4 Examples

Let's look at some spectrogram examples. Note that a logarithmic function has been applied to the computed values (improving coefficient representation for us). The signals are from a bird and a dolphin. Note that the first one was taken from a classification challenge [1].

[1] `http://www.imageclef.org/lifeclef/2015`

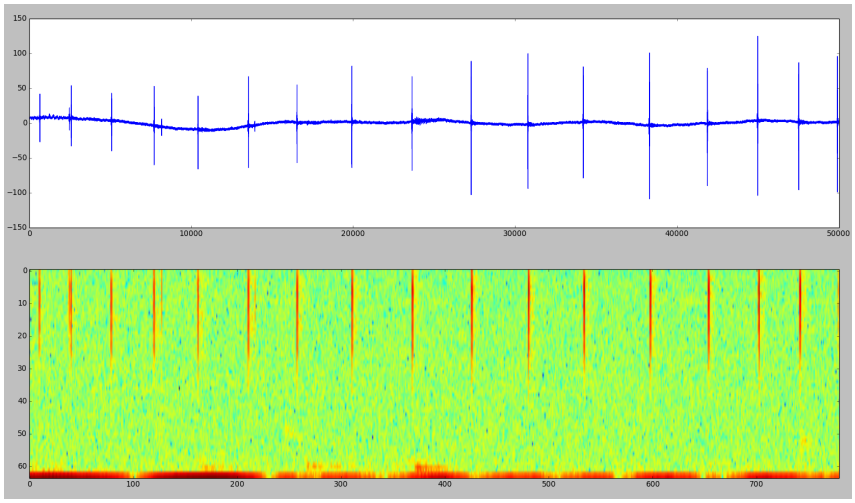


Figure 4: Spectrogram 128 50%

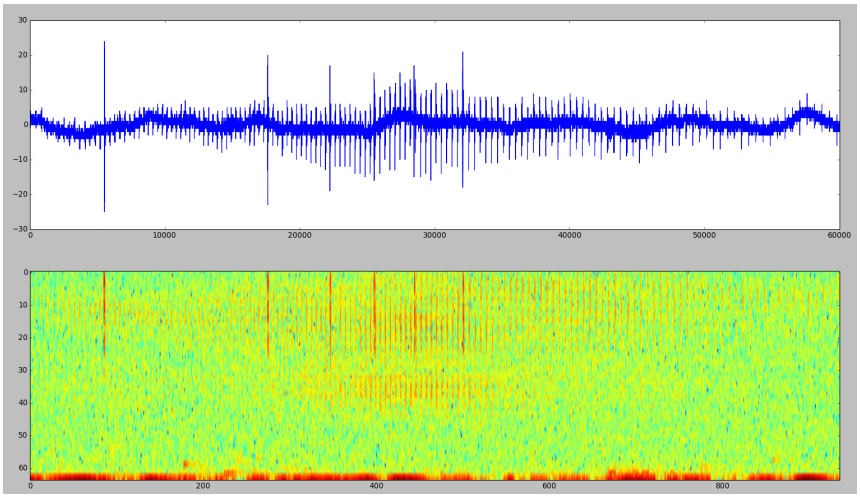


Figure 5: Spectrogram 128 50%

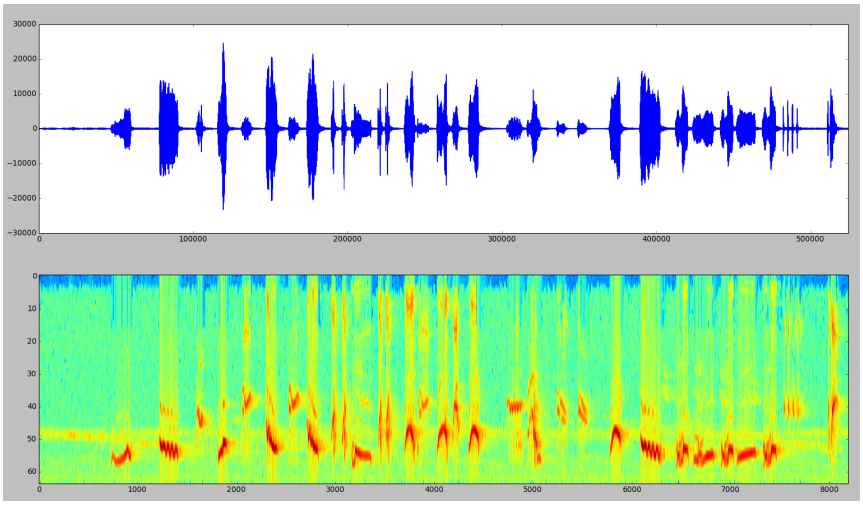


Figure 6: Spectrogram 512, 50%

4 Scattering Network

4.1 Introduction

The Scattering Network aims to find a better data representation after numerous transformations of a raw input.

The basic idea is to perform series of linear and non linear operations. The linear operations are the convolutions while the non linear ones are the applications of absolute values. The use of the latter allows fast convergence by the contractive property. The convolutions are basically decomposing the signal into a wavelet basis representation. A parallel can be make with the FFT and the complex sine decomposition, we just decompose an input into a feature vector where here features are wavelet basis. The structure itself of the network can be compared to a Convolution Neural Network where the filters are computed and fully determined by the meta parameters while in a CNN they are learned during training. This is a huge difference in term of computation time allowing an good representation without training even though filters generation is also complex.

The better data representation can be used for simple data analysis or data learning but it finds its best use in classification. In fact, this new feature space is much more suited for the use of linear classifier for example. Note that in this implementation we won't look at the reconstruction problems since our main goal is not to use the Scattering Network for compression,reconstruction,... Let's look at the general picture of the scattering network and analyse it briefly.

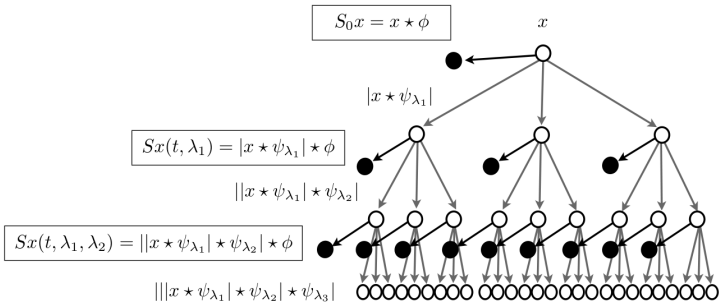


Figure 7: Scattering Summary. Source : `http://www.di.ens.fr/data/publications/papers/1304.6763v1.pdf`

In this case the scattering network is made of 3 layers. Each layer has low-pass filters (ϕ) and high-pass filters (ψ). In our specific case of 1D signals, there is only one ϕ per layer. Given an input signal x of size N we perform a low-pass decomposition (S_0x) by performing the convolution $x \star \phi$ and a high-decomposition leading to a output size of $\times 2N/T \times$ NumberOfPsisFilters by performing NumberOfPsisFilters convolutions $x \star \psi_{i,\lambda 1}$ where $\psi_{i,\lambda 1}$ is the i^{th} filter of the ψ-filter bank generated by the meta parameters $\lambda 1$. Finally on this high-decomposition is applied the absolute value operation.
Then for the second layer, each one of the previous high-decomposition is treated as a input signal and the same algorithm is performed. Details about this will be given in the scattering layer section but we can already note that the meta parameters are specific to a scattering layer Finally let's review what the meta parameters are about :

- T determines the convolution process between a filter and the signal by giving the time resolution (low T gives better time resolution which might be better for first layers)
- Q determines the quality factor (the number of filters per octave)
- J determines the number of octave to go through.
- PE (Periodization Extent) constant used in the filter periodization (1 by default)

In order to respect this architecture, this toolbox uses a specific class : MetaParam using default parameters and a TtoJ method :

```
MetaParam L1param(500);
//L1param._T=500, L1param._Q=1, L1param._J=8, L1param._PE=1
L1param=MetaParam(500,2);
//L1param._T=500, L1param._Q=2, L1param._J=6, L1param._PE=1
L1param=MetaParam(500,2,4);
//L1param._T=500, L1param._Q=2, L1param._J=4, L1param._PE=1
L1param=MetaParam(500,4,4,4);
//L1param._T=500, L1param._Q=4, L1param._J=4, L1param._PE=1
```

Let's now see the details of each implementation level and emphasize the implementation architecture used.

4.2 Filter Bank implementation

Filters are created through the constructor of the Filter1D class. Given meta-parameters and a support size, the constructor will initialize all the wanted variables and compute the actual filters. Note that the Filter1D class has two children : the MorletFilter1D and GaborFilter1D. These two specializations have their own filter generation algorithm. This also means that if one wants to implement a new filter, the only thing to do it to create another class of the name of this filter, inherit from the Filter1D class and implement the coefficients generation method.
Note that the constructor can be used in two different ways :

- Giving support size, meta parameters, and the position of the filter in this configuration (*gamma*)
- Giving a support size, a sigma and a xi.

The first one is more practical for the ψ generation since the size and the meta parameters are fixed for a layer, we just have to loop through γ. On the other hand, the second constructor is simpler for the ϕ filter generation, in fact, since only one filter is made per layer, we just have to compute ξ and σ for this filter. Here is an example with arbitrary coefficients :

```
Filter1D* BankFilter=new Filter1D[5];
BankFilter[0]=GaborFilter1D(500,0,1,2);//500 points, xi=0,sigma=1,PE=2
for(int i=1;i<5;++i)
    BankFilter[i]=MorletFilter1D(500,2+0.5*i,0.2*i);//500 points, xi=f(i),
                                                    //sigma=g(i),PE=1 (default)
ofstream file("filters.txt");
for(int i=0;i<5;++i){
    file<<BankFilter[i];// use of the overloaded operator
    file<<"\n";
}
delete[] BankFilter;
file.close();
```

Giving the following result :

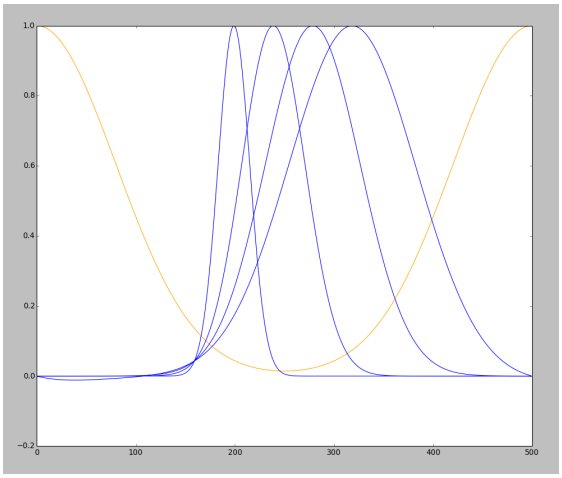


Figure 8: Filters generation example, orange : Gabor filter, blue : Morlet wavelets.

The filters are directly computed in the Fourier domain to speed up the decomposition algorithm, indeed we only have to compute the FFT of the input

to perform the decomposition algorithm now. Here ξ corresponds to the central frequency and so to the global maximum position. It can be seen as a position parameter while σ is a scale parameter. In practice, in order to generate the filters we always take the mother coefficients that are exponentially transformed through a scale coefficient. We have

$$\Xi = \frac{\pi}{2} * (2^{-1/Q} + 1)$$

$$\Sigma = \sqrt{3} * (1 - 2^{-1/Q})$$

And so the scaling factor for the filter i is :$\lambda_i = 2^{-i/Q}$ which leads to the following coefficients for any given filter i for a specific layer having the same meta parameters :

$$\xi_i = \Xi * \lambda_i$$

$$\sigma_i = \Sigma * \lambda_i$$

Filters In this implementation, high-pass filters are Morlet wavelets while low-pass filters are Gabor filters. Note that Morlet filters are actually another name for Gabor kernels. The difference between the Gabor function (nonzero-mean function) and the Gabor kernel (zero-mean function) is that the Gabor kernel satisfies the admissibility condition for wavelets (integral equals to 0), thus being suited for multi-resolution analysis. The admissibility condition ensures that the inverse transform and Parseval formula are applicable.

Filter Periodization In order to always have 2π-periodic filters, we use the PE (periodization extent) coefficient in a special case of the following formula :

$$f(x) = \sum_{n \in \mathcal{Z}} f(x + 2\pi n)$$

In practice nothing assures the convergence but since we use gaussian functions quickly decreasing to 0 this is not a problem. Also in practice, we use $n \in \{-PE, -PE+1, ..., PE, PE+1\}$ with $x \in \{x \in \mathbb{R} : x = i*2\pi/T, i = 0, ..., T-1\}$ which is similar to $x \in \{0, 2\pi/T, 2 * 2\pi/T, ..., (T-1)2\pi/T\}$ So x covers $[0, 2\pi[$ with T points linearly separated by $1/T$. In practice a periodization extent of 1 is already enough.
The n coefficients implies the range on which the wavelet is evaluated :

$$[-2\pi * PE, 2\pi * (1 + PE) - 1/T]$$

4.2.1 Graph

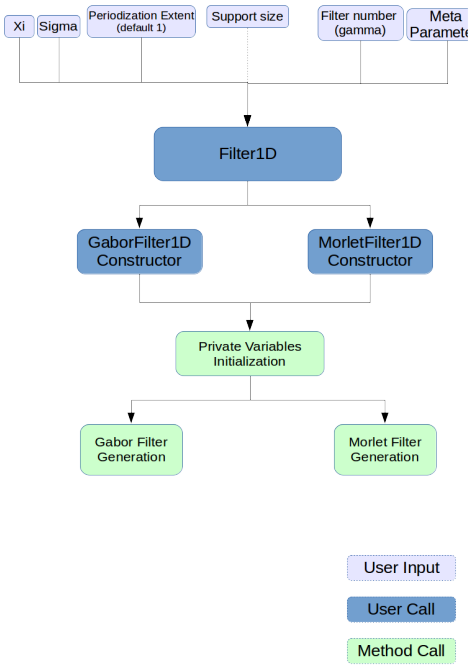


Figure 9: Filter1D Summary Diagram

4.3 Layer Implementation

The role of this class is to be the link between the raw input, the meta parameters, and the bank filters by performing the decomposition process. Firstly, this class takes a $2D$ input. This force to transform the 1D input signal for the first layer but then allow an easy link between layers by directly setting the input of the next one as the output of the previous one. Given the set of inputs, private variables are computed which determine the structure of the class by computing variables that will be passed to the next layer such as the size of the output (given the input size and the number of psis filters :$Q * J$). Then when all the psis filters are available a Littlewood-Paley normalization is performed (due to the logarithmic spaced filters). After this the filters are generated using the Filter1D class. The Decomposition can now be performed.

4.3.1 Graph

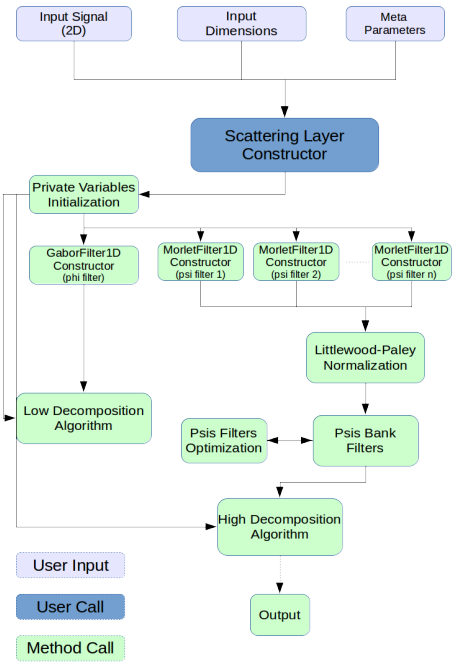


Figure 10: Scattering Layer Summary Diagram

4.4 Decomposition Implementation

The core of the algorithm lies in this decomposition. Firstly, the convolution defined in the section 4.1 are redundant and so are performed only every $T/2$ points. This result in a reduced output length and faster computation. Secondly, the filters are 0 almost everywhere and so the product between the FFT of the signal and the filters is mostly a waste of time. To solve this the resulting coefficients are only computed for the filter's non negligible coefficients (and set to 0 everywhere else). These informations are computed by the Filter1D class during the call of OptimizeFilter. In addition, it is necessary to perform a periodization since we perform in time domain a convolution every $T/2$ points we must do the product of the FFT followed this before computing the IFFT (allowing a time sub-sampling). The output length must then be InputSize $* 2/T$. Doing this for each psis filter gives the output of the layer. Here is a simple scheme not taking into account these optimizations in order to emphasize the algorithm :

Data: Input,inputN,inputM,Meta Parameters
Result: Output,outputN,outputM
NumberOfPsis=J*Q;
outputN=inputN*NumberOfPsis;
outputM=inputM∗2/T;
output (outputN,outputM);
BankPsis;
Phi;
for $i = 0 \rightarrow inputN$ **do**
  inputFFT=FFT(input[i]);
  LowDecomposition[i]=IFFT(periodize(inputFFT.*Phi));
  for $j = 0 \rightarrow NumberOfPsis$ **do**
    HighDecomposition[i*NumberOfPsis+j]=abs(IFFT(periodize(inputFFT.*BankPsis[j])));
  end
end

Algorithm 1: Decomposition Algorithm

With periodize being the function that will periodize the result in order to sub-sample in the time domain to obtain the desired output size.

4.5 Scattering Network Implementation

Finally here is how to perform the Scattering Network on a signal and save The outputs :

```
MetaParam* opt=new MetaParam[3];
opt[0]=MetaParam(8,30,4,1);
opt[1]=MetaParam(64,1,1,1);
opt[2]=MetaParam(1024,1,1,1);
ScatteringNetwork decomposition("lifeclef.wav",opt,3);

ofstream file;
cout<<"DONE MKOT"<<endl;
file.open("layer1.txt");
file<<decomposition[0];
file.close();
file.open("layer2.txt");
file<<decomposition[1];
file.close();
file.open("layer3.txt");
file<<decomposition[2];
file.close();
delete[] opt;
```

In fact the operator [] is overloaded to return the specific layer which itself use its overloaded operator to export the coefficients.

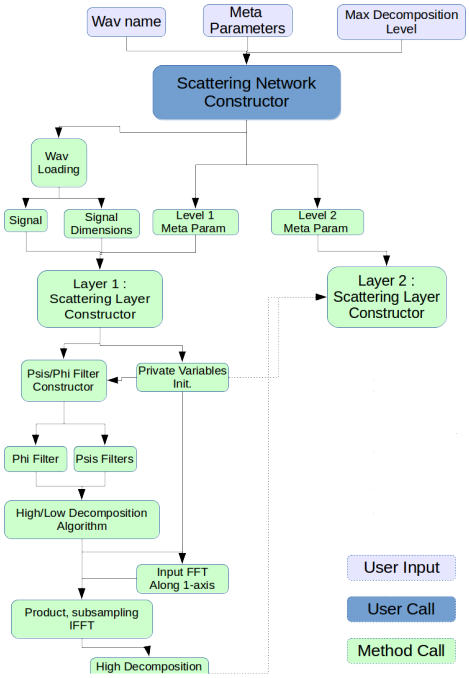


Figure 11: Scattering Network Summary Diagram

4.6 Examples

No operations applied (logarithm of renormalization or else)

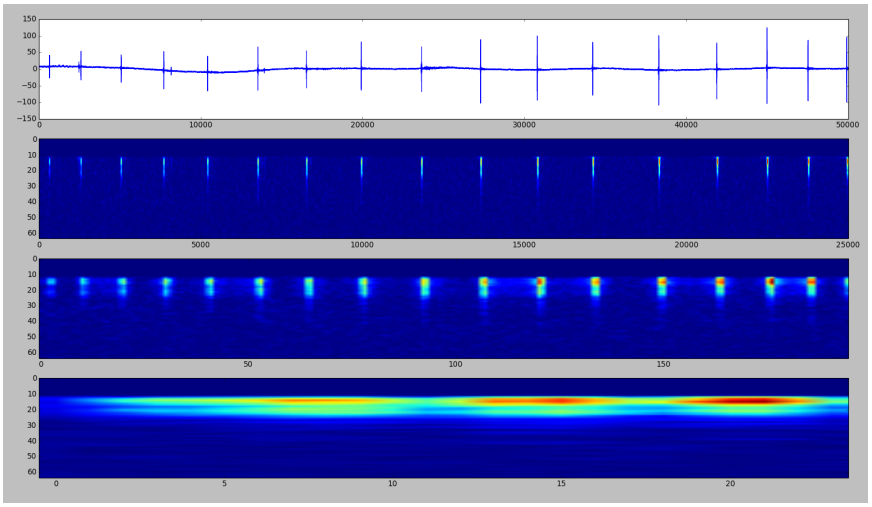


Figure 12: L1, L2, L3 of Inia Perou (slow clicks) T1:4 Q1:32 J1:2 T2:256 Q2:1 J2:1 T3:16 Q3:1 J3:1

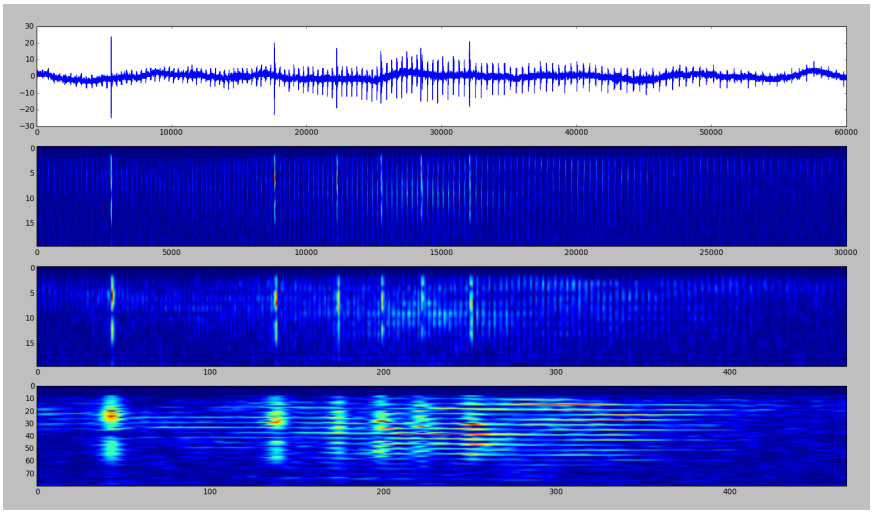


Figure 13: Inia Perou (fast clicks) 4 20 1 128 1 1 2 4 1

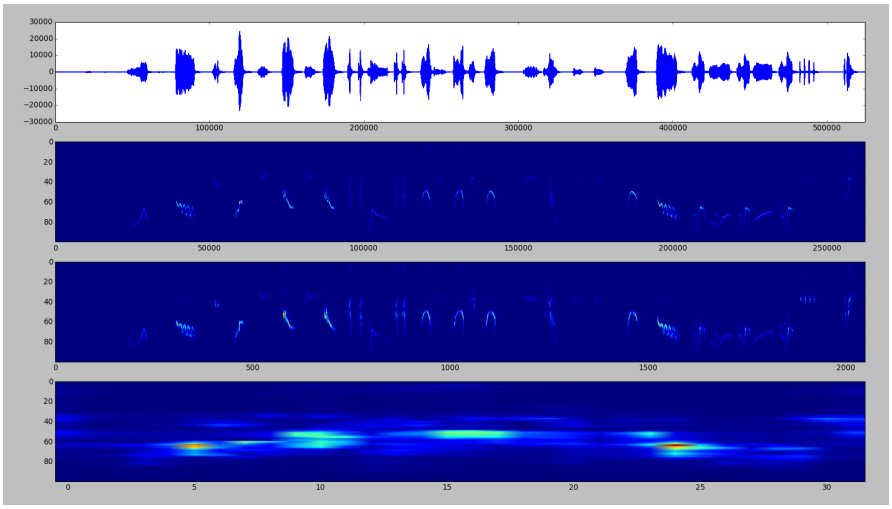


Figure 14: Bird (LIFECLEF Challenge) 4 25 4 256 1 1 128 1 1

5 imagesc TODO

5.1 Import data

Load data. First line uses push back method that expands the size of the vector by 2 when it's full meaning that for N points only $\log(N)$ expansions are made. Following lines directly have the right length. No other way since we don't know the size of the matrix to load.

5.2 Optimization

Rendering is done using openGL directly from C. Squares are used because no need for triangles since they all lie in the same plane in 2D, this also reduces the number of shapes to draw. The best optimization is the use of the array by allowing the good client states, in fact passing directly the vertex color and points arrays reduces functions calls by a lot.

5.3 Colormap

To render good spectrograms or scalograms is it necessary to have a good colormap. The one used here is the same as Matlab or Python.

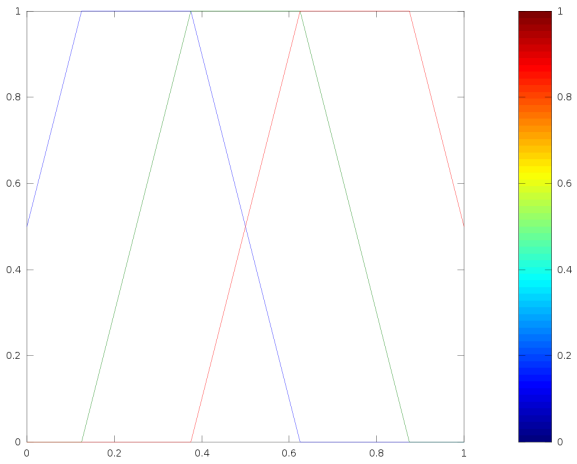


Figure 15: Colormap used in the toolbox

5.4 Implementation

Given a matrix $N \times M$ squares are computed using or not the original aspect ratio. If it is kept then a matrix of size $N \times 2N$ will be displayed into $[0, 1] \times [0, 0.5]$ and this is true for every possible ratio. Because this can become bad for really low ratio (few rows millions of columns) it is possible to disregard the original ratio drawing the matrix into the unitary square. Even it is not mandatory this part of the toolbox is made to be used in command line rather than inside a program, this can be easily done by :

```
./imshow myspectro.txt
./imshow 0 myspectro.txt
```

The 0 option specifies to no keep the original aspect ratio meaning that now the image will be displayed as a square.

References

[1] *Deep Scattering Spectrum.*

[2] A dft and fft tutorial. http://www.alwayslearn.com/dft%20and%20fft%20tutorial/DFTandFFT_BasicIdea.html, June 2014.

[3] Stéphane Mallat. *A wavelet tour of signal processing*. Academic press, 1999.

[4] Joo Martins. How to implement the fft algorithm. http://www.codeproject.com/Articles/9388/How-to-implement-the-FFT-algorithm, February 2005.

[5] Vlodymyr Myrny. A simple and efficient fft implementation in c++:part 1. http://www.drdobbs.com/cpp/a-simple-and-efficient-fft-implementatio/199500857, May 2007.

[6] Craig Stuart Sapp. Wave pcm soundfile format. https://ccrma.stanford.edu/courses/422/projects/WaveFormat/, December 2011.

[7] Laurent Sifre and Stephane Mallat. Rotation, scaling and deformation invariant scattering for texture discrimination. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2013.

[8] Wikipedia. Fast fourier transform. http://en.wikipedia.org/wiki/Fast_Fourier_transform, December 2014.

[9] Wikipedia. Wav. http://en.wikipedia.org/wiki/WAV, November 2014.