UPMC Research Intern. at DI ENS Ulm with Stéphane MALLAT
Scattering decomposition for massive signal classification : from theory to fast algorithm and implementation with validation on international bioacoustic benchmark

Randall BALESTRIERO
randallbalestriero@gmail.com

Pierre et Marie Curie University, Paris 6

Contents

0.1 Introduction

With the computational power available today, machine learning is becoming a very active field finding its applications in our everyday life. One of its biggest challenge is the classification task involving data representation (the preprocessing part in a machine learning algorithm). In fact, classification of linearly separable data can be easily done. The aim of the preprocessing part is to obtain well represented data by mapping raw data into a "feature space" where simple classifiers can be used efficiently. For example, almost everything around audio/bioacoustic uses MFCC features until now. This toolbox gives the basic tools for audio representation using the C++ programming language by providing an implementation of the Scattering Network [**?**] which brings a new and powerful solution for these tasks. The toolkit of reference in scattering analysis is SCATNET from Mallat et al. [1]. This tool is an attempt to have some of the scatnet features more tractable in large dataset. 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.
This toolbox gives out of the box executables that can be used by simple bash commands. Finally, for each presented algorithm, a graph is provided in order to summarize how the computation is done in this toolbox.

As we will review in the next part, we will need to perform data manipulation on huge dataset. It becomes important to have fast and efficient implementations in order to deal with this new "Big Data" era. The first descriptions concern mandatory utilities, such as audio file IO and basic tools (fft,...) that are necessary for the scattering network implementation.

[1] `http://www.di.ens.fr/data/software/scatnet/`

Part I

Fast Scattering

Chapter 1

Scattering Network

The Scattering Network aims to find a better data representation after numerous transformations of a raw input. It's been developed by Stéphane MALLAT and its team and the only available implementation is in Matlab which provides high-level programming but is not optimal in term of execution time. In fact, this algorithm just started to be applied in concrete challenges and finds its limitation in the required time to complete the transformation on massive dataset. We will review its core ideas and the implementation architecture chosen in C++.

1.1 Introduction

The basic idea is to perform series of linear and non linear operations on a $1D$ given input signal x. The linear operations are done through the convolutions while the non linear ones are the use of the absolute value on these convolutions. The use of the latter allows fast convergence by the contractive property. The convolutions of the first layer $|x \star \psi_{\lambda_1}|$ are basically decomposing the signal into a wavelet basis. A parallel can be made with the FFT and the complex sine decomposition. In addition, the application of the low-pass filter ϕ brings time invariance (of length depending on the ϕ support/bandwidth).

The structure itself of the network can be compared to a Convolutionnal Neural Network where the filters are given and fully determined by the meta parameters and not learned during a training phase. This is a huge difference in term of computation time allowing good representation without training. We have to keep in mind that filters generation is also complex and time consuming.

The mapped data into the feature space can be used for simple data analysis or data learning but it finds its best use in classification. In fact, this feature space is much more suited for the use of linear classifier. 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.

In this case the scattering network is made of 3 layers. Each layer has a specific low-pass filter $(\phi_i, i = 0, 1, 2, 3)$ and high-pass filter banks (ψ_1, ψ_2, ψ_3). 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 $|x \star \phi_0| := S_0 x$ and a high-decomposition $x \star \psi_{1,\lambda_1}$. Note that the convolutions are then sub-sampled leading to a output size of $|\lambda_1| \times 2N/T$ for U_1. The sub-sampling only reduce coefficient redundancy and depends on the T coefficients. The i^{th} filter of a given high-pass filter bank is generated by the meta parameters of the corresponding layer.
Finally on these convolutions is applied the absolute value operation leading to the high-decomposition $U_1 := |x \star \psi_{1,\lambda_1}|$ of the first scattering layer. The scattering coefficients are then easily computed with the application of the low-pass filter ϕ_1 leading to the scattering coefficient per say $U_1 \star \phi_1 := S_1$.
Then for the second layer, each one of the previous high-decomposition row is treated as an input signal and the same algorithm is performed again. Details about this will be given in the scattering layer section5.3 but we can already note that the meta parameters are specific to each scattering layer. Finally let's review what the meta parameters are about :

- T determines the ϕ support/bandwidth, this is basically the invariant time window. Taking a great T leads to huge time invariance but at the cost of variance (and information) loss with respect to the time dimension.
- Q determines the quality factor (the number of filters per octave)
- J determines the number of octave to go through.
- PE (Periodization Extent) is the constant used in the filter periodization (1 by default)

1.2 Filter Bank

Filters are created given the meta-parameters, the support size an the filter position in the filter bank. Using this, the coefficients σ and ξ are deduced.
For the ψ generation, it is efficient to loop through γ since the meta parameters and the total size of the filters are the same (only the filter number in the filter bank changes). Note that the support of these filters expands when λ increases. On the other hand, creating a specific filter with just the bandwidth and frequency max. position is more efficient and is thus used for the ϕ generation. In fact, since only one low-pass filter is made per layer, we just have to compute ξ and σ for this filter.

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 transformed through a scaling coefficient with exponential rate. We have then as mother coefficients :

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

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

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 (non-zero-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 increase resolution of the filters, we can compute them on a bigger interval than the one we are interested in and then periodize them in the Fourier domain :

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

In practice nothing assures the convergence for any function f but our filters are generated through Gaussian functions which actually guarantees the convergence. In practice, we use $n \in \{-PE, -PE+1, ..., PE, PE+1\}$ with $x \in \{x \in \mathbb{R}, i = 0, ..., T-1 : x = i * 2\pi/T\}$ which is similar to $x \in \{0, 2\pi/T, 2 * 2\pi/T, ..., (T-1)2\pi/T\}$. With this definition x covers $[0, 2\pi[$ with T points linearly separated by a distance of $1/T$. In all the examples presented here a periodization extent of 1 is used.
The n coefficients affect the range on which the wavelet is evaluated which grows with bigger n :

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

It is then shrunk into the desired support size by the periodization process.

1.3 Scattering Layer

The scattering layer performs the decomposition of the raw input given the meta parameters and the filter banks by performing the decomposition process.
With the filter bank, all the ψ filters are available and we can apply Littlewood-Paley normalization necessary because of the logarithmic spaced filters.

Decomposition Implementation The core of the algorithm lies in this decomposition. Firstly, the convolution defined in the section 1.1 is redundant and so is only performed on every $T/2$ spaced points. This implies a reduced output length and faster computation. Thus, it is necessary to perform a periodization before computing the IFFT (allowing a time sub-sampling). The output length must then be InputSize $* 2/T$. Doing this for each ψ filter gives the output of the layer. Here is a simple scheme to emphasize the algorithm :

Data: Input,inputN,inputM,Meta Parameters
Result: Output,outputN,outputM
NumberOfPsis=J*Q;
outputN=inputN*NumberOfPsis;
outputM=inputM$*2/T$;
ψ Filter Bank creation;
ϕ creation;
for $i = 0 \rightarrow inputN$ **do**
    inputFFT=FFT(input[i]);));
    for $j = 0 \rightarrow NumberOfPsis$ **do**
        HighDecomposition[i*NumberOfPsis+j]=abs(IFFT(inputFFT. $*$ $\psi_{L,j}$);
        Scattering[i*NumberOfPsis+j]=IFFT(periodize(
        FFT(HighDecomposition[i*NumberOfPsis+j]. $* \phi_L$)))
    end
end

Algorithm 1: Decomposition Algorithm of the layer L

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. For example the array $[0, 1, 2, 3, 4, 5]$ is equal to $[3, 5, 7]$ after a periodization by a factor of 2.

The degrees of freedom of the model are :

- T
- Q
- J
- h_1

Where we have :

$$U_1 = \begin{pmatrix} |x \star \psi_{1,1}| \\ |x \star \psi_{1,2}| \\ ... \\ |x \star \psi_{1,|\lambda_1|}| \end{pmatrix} \in \mathbb{R}^{|\lambda_1| \times N_1}$$

$$S_1 = \begin{pmatrix} |x \star \psi_{1,1}| \star \phi_1 \\ |x \star \psi_{1,2}| \star \phi_1 \\ ... \\ |x \star \psi_{1,|\lambda_1|}| \star \phi_1 \end{pmatrix} \in \mathbb{R}^{|\lambda_1| \times N_2}$$

$$U_2 = \begin{pmatrix} ||x \star \psi_{1,1}| \star \psi_{2,1}| \\ ||x \star \psi_{1,2}| \star \psi_{2,1}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,1}| \\ ||x \star \psi_{1,1}| \star \psi_{2,2}| \\ ||x \star \psi_{1,2}| \star \psi_{2,2}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,2}| \\ \dots \\ ||x \star \psi_{1,1}| \star \psi_{2,|\lambda_2|}| \\ ||x \star \psi_{1,2}| \star \psi_{2,|\lambda_2|}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,|\lambda_2|}| \end{pmatrix} \in \mathbb{R}^{|\lambda_1|*|\lambda_2| \times N_3}$$

$$S_2 = \begin{pmatrix} ||x \star \psi_{1,1}| \star \psi_{2,1}| \star \phi_2 \\ ||x \star \psi_{1,2}| \star \psi_{2,1}| \star \phi_2 \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,1}| \star \phi_2 \\ ||x \star \psi_{1,1}| \star \psi_{2,2}| \star \phi_2 \\ ||x \star \psi_{1,2}| \star \psi_{2,2}| \star \phi_2 \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,2}| \star \phi_2 \\ \dots \\ ||x \star \psi_{1,1}| \star \psi_{2,|\lambda_2|}| \star \phi_2 \\ ||x \star \psi_{1,2}| \star \psi_{2,|\lambda_2|}| \star \phi_2 \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,|\lambda_2|}| \star \phi_2 \end{pmatrix} \in \mathbb{R}^{|\lambda_1|*|\lambda_2| \times N_4}$$

Chapter 2

Choice for Fast Audio File Manipulation

2.1 File Structure

The WAV file is an instance of a Resource Interchange File Format (RIFF) defined by IBM and Microsoft. The 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 :

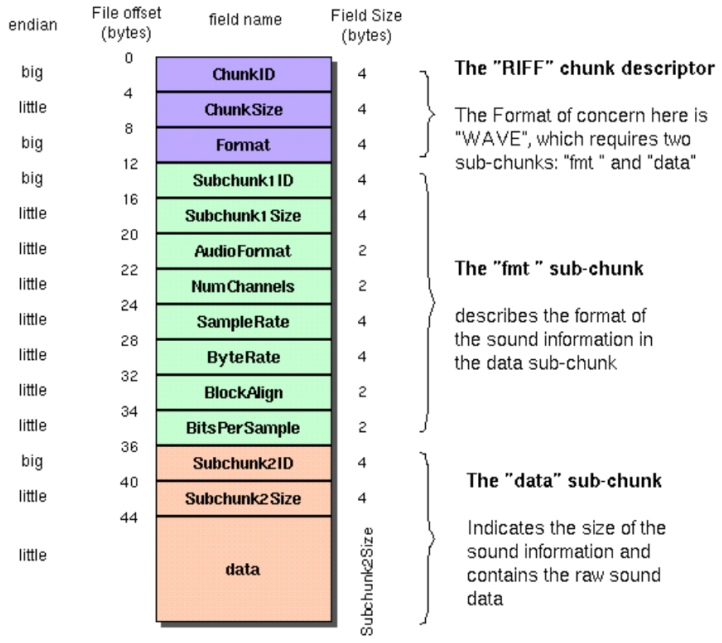

Figure 2.1: The Canonical WAV file format `https://ccrma.stanford.edu/courses/422/projects/WaveFormat/`

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

Chunk

- 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).

Sub Chunk 1

- 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 equal 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 : $\text{SampleRate} * \text{NumChannels} * \frac{\text{BitsPerSample}}{8}$.
- BlockAlign won't be necessary for us. It can be computed with the formula : $NumChannels * \frac{\text{BitsPerSample}}{8}$.
- BitsPerSample can either be 8 or 16 but in general the later is used.

Sub Chunk 2

- 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.
- Data is the block containing the values of the signal in the standard pulse-code modulation representation.

2.2 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 or at any other time. Let's look at an example.

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

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[i];//return the ith 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 exports the signal data disregarding all the other informations. This loss can be avoided using a special method which exports 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 no normalization is used. In fact, only the user can define the normalizing constant he is interested in (max, L^2-norm,...) and so has to apply it after the import. 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.

Chapter 3

Fast Fourier Transform implementation

3.1 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 of 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 and it is half of this 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 by the use of convolutions.

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

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

3.2 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 $N-1$.
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 $O(N^2)$ complexity.
Note that 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 which is not needed in this toolbox thus not used.
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

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). 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 $N = 8$:

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

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 $N = 2$:

$$W_2^0 = W_2^2 = W_2^4 = ...$$
$$W_2^1 = W_2^3 = W_2^5 = ...$$

And for $N = 4$

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

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.

Butterfly Scheme Finally, the last brick is the butterfly scheme that can be seen in the next diagram3.4 allowing an in-place FFT which is memory friendly.

3.3 Inverse Fourier Transform

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

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

3.4 Algorithm

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.

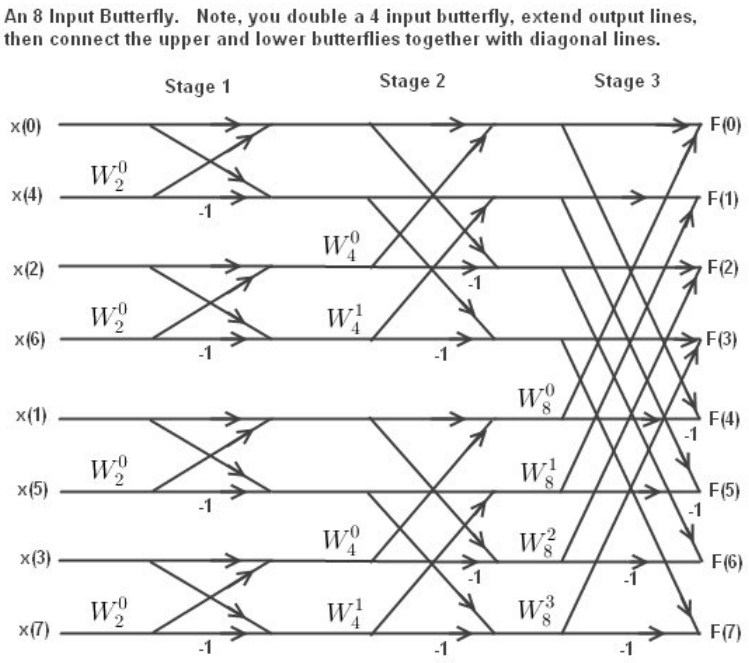

Figure 3.1: Full FFT computation with $N = 8$ [**?**]

This simple example performs the following computation in the implemented algorithm :

Listing 3.1: Computation done during the FFT computation of a signal of size 8. Each column represents the overall loop $9log(N)$ iterations. W is the Twiddle Factor and each block is the inner loop performing the butterfly scheme.

```
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]
```

The 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 type signal for example.

3.5 Graph

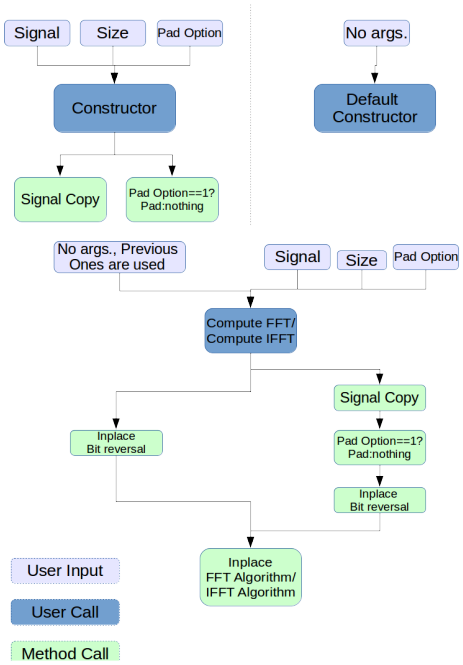

Figure 3.2: FFT Summary Diagram

Chapter 4

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 lesser 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 decomposition and thus the scattering network since this window size is not constant any more.

4.1 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 chunk 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 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 is simply the complex conjugate of the first half, since in the spectrogram we display the absolute value of the coefficients, we get symmetry about the middle point.
Most window functions afford more influence to the data at the center of the window than to the data at the edges, which represents a loss of information. To mitigate that loss, it is common to use overlapping in time (usually 50%).

4.2 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
    //and default parameters with the already declared spectrogram variable
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 one) this has to be done 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 user defined function passed as last argument when calling the Perfom method.

4.3 Graph

The spectrogram graph is simple and emphasizes the default parameters of the class/methods.

4.4 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 of the BIRDLIFE CLEF Challenge 2014 [1]. and a Inia dolphin.

%endfigure

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

Chapter 5

Scattering Network Fast Implementation

5.1 Meta-Parameters

In order to respect the scattering network architecture, this toolbox uses a specific structure : 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=4
```

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

5.2 Filter Bank Creation

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 is 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 σ and a ξ.

The first is used to generated the ψ filter bank and the second one for the ϕ 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 : Let's now look at the high-pass filter bank generated through a scattering layer :

```
MetaParam AA;
AA._Q=8;
AA._J=5;
AA._T=4;
ScatteringLayer LL;
LL.CreatePsis(pow(2,10),AA);
ofstream F("psis.txt");
F<<LL._Psis;
F.close();
```

Giving the following result :

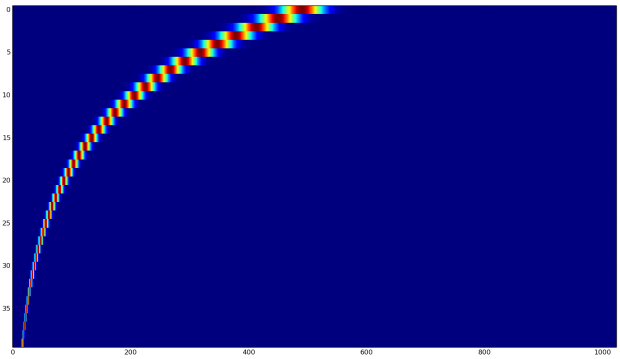

Figure 5.1: On 1024 bins, Q=8, J=5 without optimization on the support

5.3 Layer Implementation

The role of this class is to be the link between the raw input, the meta parameters, and the filter banks by performing the decomposition process. Firstly, this class takes a $2D$ input (the input signal has to be transformed for the first layer). This allows an easy link between layers by directly setting the input of the next one as the output of the previous one.
Given the input, private variables are computed determining 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 ψ filters :$Q * J$). Then when all the ψ 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.
Note that the decomposition is stored as a $2D$ matrix for every layer. Normally,

layer i has a dimension of $i+1$ which is not true in term of memory management. In fact, in this toolbox, the graph structure of the scattering network?? has been kept through $2D$ matrices. For example for the second layer if we have $|\lambda_1|$ filters for the first layer and $|\lambda_2|$ filters for the second layer, the U_2 matrix of will be :

$$U_2 := \begin{pmatrix} ||x \star \psi_{1,1}| \star \psi_{2,1}| \\ ||x \star \psi_{1,2}| \star \psi_{2,1}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,1}| \\ ||x \star \psi_{1,1}| \star \psi_{2,2}| \\ ||x \star \psi_{1,2}| \star \psi_{2,2}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,2}| \\ \dots \\ ||x \star \psi_{1,1}| \star \psi_{2,|\lambda_2|}| \\ ||x \star \psi_{1,2}| \star \psi_{2,|\lambda_2|}| \\ \dots \\ ||x \star \psi_{1,|\lambda_1|}| \star \psi_{2,|\lambda_2|}| \end{pmatrix}$$

5.4 Scattering Network Implementation

Finally here is how to perform the Scattering Network on a signal and to 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("signal.wav",opt,3);

ofstream file;
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 uses its overloaded operator to export the coefficients.

5.5 Optimizations and Computation Time

5.5.1 Filter Bank Optimizations

Since we want to use this implementation on massive dataset, it is mandatory to optimize the implementation.

The best way to improve the required time to perform the decomposition is around the filters (allocation, creation, application,...). In fact, FFT/IFFT and the periodization algorithms already use efficient algorithms.

I present here only the improvements concerning the filter bank, other details can be found in the C++ file comments.

The first big improvement is to only save in memory the coefficients of the filters that are not zero. So by only keeping the support of ψ we can save memory but more importantly when we apply the operation $inputFFT. * \psi_{L,i}$ we only need

to perform it on the support and set all the other coefficients to 0. Doing this is really efficient by the natural sparsity of the filters as shown here : 5.1.
Now that we only save part of the coefficients, the dimension is reduced and it is even possible to optimize again the filters creation. We know that filters are completely determined by the size of the input N and the meta parameters Q and J for the ψ filters. So in other word, it is possible to create a filter bank dictionary and save them in .txt files thus when we need some filters we only need to load them and not explicitly compute them over and over. This is particularly true since the input size must be a power of 2 so there are not a huge number of input size that we can take as input. By doing so, when we treat for example thousands of files, we will only load a few files without having to recompute the filters coefficients every single time. Now, if we use the implementation in a loop over several signals, it is even possible to first load the filters and then perform all the decomposition without having to load filters every time too !
Note however that this optimization is not always usable. In fact, the use of this toolbox on other platform/servers or just because the user doesn't want IO operations for example, requires to actually compute the filters every time. In this case, the generating algorithm has been optimized using standard C++ optimization techniques such as loop-unrolling, inplace methods,...

5.5.2 Computation Time

Using all the available optimization without the use of the filter bank IO (thus generating the filters every time) this toolbox needs about 30% of the signal duration (in second) to perform the loading of the wav, scattering decomposition and saving of the scattering coefficients into a .txt files). This required time obviously varies with the meta-parameters, the ones use here are the common ones (also used for the classification challenge presented later). Note that using the filter bank IO will lead to better performance on huge dataset by not recomputing the filters for every signal. The Scatnet implementation available in matlab requires 400% of the signal time to perform the decomposition algorithm.
Note that this toolbox is now used as the reference implementation for the scattering decomposition on ENS and UTLN servers for massive dataset decomposition (such as bioacoustic challenges).

5.6 Examples

In the examples below we did not apply any operation (logarithm, re-normalization,...). The sub-plots are from top to bottom the signal, and the U_1, U_2, U_3.

Part II

Benchmark and Bioacoustic Classification

Part III

Function parser

5.7 Motivation

Throughout this library, we will need to parse functions in order to compute them explicitly. By functions we mean user-defined function given as a string and retrieved by a terminal interaction or through a Qt widget. In order to do this, multiple approaches can be used.
We want to be able to interpret any given function such as the string $f(x) = x^3 - cos(x)$, in order to parse it, a binary tree is generated recursively on the given string with some simple rules (method used in this library). Another method (more related to compiler and language theory) implies the use of automaton.
Note however that the parser is very basic and doesn't take into account operator priority so it is necessary to add parenthesis (as shown in the examples).

5.8 Tree Generation

5.8.1 Rules

The basic idea is to read a string from left to right and deduce what the current character stands for (this is done letter by letter). This will be really simple since the only unknowns are x and y so any other letter which is not an operator can only be a function $((e)xp, (s)in, ...)$. To test if a character is an operator it is enough to test if the character is $+$, $-$, $*$, / or power, thus the need of the two functions

- bool IsOp(char)
- bool IsFunc(char)

Finally, note that if the character is a number, a utility function GetNumber will extract the number by taking all the following and adjacent characters that are numbers too. In fact, if the character is 5 the next one can be another number if the user input was 58 for example thus GetNumber("58+4*x") will return 58.

5.8.2 Structure

The generated tree will be made of nodes and leaves (the final nodes). A leaf can either be a constant (a number) or an unknown x or y.
A node can be a function or an operator and will have two children (not necessarily leaves) for an operator node and only one for a function node. All of these structures inherit from a node class which provides an access operator() that will be accessed recursively to evaluate the parsed string (the generated tree).

So let's look for example at the string $555+x$. First we read 5, it is a number so we look if the next character is also a number and so on to extract the whole plain number $\rightarrow$ extract 555.
Now analyse the next character + which generate an operator node where its two children are the string 555 and x. The analysis is done recursively thus the string 555 becomes a constant leaf and x a variable leaf. the access operator

will then be used as $f(t) = 555 + t$ allowing the computation of the given string. This was a really simple example but one can write for example

$$exp^{(cos(x)^2)/(sin(x)^2)} + N(0,1)$$

It is important to note that the limits are taken into account, thus, plotting the tangent $sin(x)/cos(x)$ will given the right asymptotes.

The parenthesis are also important for operation priority since the operators are taken from right to left. This means that the multiplication is not, without parenthesis, computed first compared to an addition for example unless it is to the left of the addition. For example, $5 * 3 + 4$ will give 35 and not 19 so one has to write $(5 * 3) + 4$. it is interesting to know that this kind of priority and syntax (left-to-right) exists in arithmetic.

5.9 Tree Evaluation

When calling the access operator ()(float) there is a recursive call on all the children and the recursion stops on the leaves where an actual value is returned. Given our last example : $f(5) \rightarrow 555 + (5) \rightarrow 560$. Obviously, only the variable leaf node will take into account the given value to the access operator, so the constant leaf 555 will always return 555 whereas the variable leaf x will return the given value.

5.10 Implemented functions

Some basic functions are already implemented. In addition, some random variables are also present and can be used really easily, for example, one can write $f(x) = x^2 + N(0,1)$ meaning that for each given point, a standard normal distribution value will be generated and added to the value x^2. This can be useful to simulate some white noise for example or to add randomness...
On the other hand, it is really easy to add some custom functions into the parser. For example one simply needs to add the function name (as a string) into the function list of the parser. This can allow use of custom functions. For example, if one wants to define g a function returning x^2 it is possible to add it and then use in the parser the following string $f(x) = 4 * g(x + 3) - 2$ for example.
The already supported functions are :

- exp
- sin
- cos
- ln

It is also possible to export the parsed function back into a string using some methods defined for the leaves/nodes.

5.11 Example

Let's look at some examples :

```
PlainMatrix<> X;
X=arange(-10,10,100);
string h("45+x+225-20");
string g("1+N(0,100)");
Tree a(g);
PlainMatrix<> Y;
Y=a(X);
```

Part IV

Solver

We will study the form

$$Ax = b$$

$$A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}$$

Note that all the presented solvers are specialized for each shaped matrix when this implies a reduced computation time (by taking advantage of the special matrix shape).

5.12 Direct Method

5.12.1 LU decomposition

The most common method for solving the system $Ax = b$ is the Gauss pivot. When we save the pivot coefficient, it is called a LU decomposition and is the most efficient direct method for exactly solving a matrix or finding its inverse (not taking count of round-off errors). It is provided for plain and sparse matrices. In order to solve a system with this method we need first to decompose a matrix into two triangular matrices and then use forward/backward substitution for solving.

Algorithms : Here are the substitution algorithms (forward and backward) :

Data: L,b
Result: x
$x = b$;
$x[0] /= L(0,0)$;
for $i = 1 \rightarrow N-1$ **do**
  for $j = 0 \rightarrow i-1$ **do**
    $x[i] -= x[j] * L(i,j)$;
  end
  $x[i] /= L(i,i)$;
end

Algorithm 2: Forward Substitution with plain Matrix $O(N^2)$

Data: U,b
Result: x
$x = b$;
$x[N-1] /= U(N-1,N-1)$;
for $i = N-2 \rightarrow 0$ **do**
  for $j = N-1 \rightarrow i+1$ **do**
    $x[i] -= x[j] * L(i,j)$;
  end
  $x[i] /= L(i,i)$;
end

Algorithm 3: Backward Substitution with plain Matrix $O(N^2)$

And now the decomposition algorithm storing the results into triangular matrices for efficient memory management. It would also be possible to store all the coefficient into one plain matrix with the upper triangular part and lower

triangular being the two triangular matrices but since we have a triangular matrix class already implemented this will bring more flexibility.

Data: A
Result: L,U
$U = A$;
$L = diag(N)$;
for $k = 0 \rightarrow N - 1$ **do**
  for $i = k \rightarrow N - 1$ **do**
    $L(i,k) = A(i,k)/A(k,k)$;
    for $j = k \rightarrow N - 1$ **do**
      $A(i,j) = A(i,j) - L(i,k) * A(k,j)$;
    end
  end
end

Algorithm 4: LU $O(N^3)$

Finally, for solving a system, the algorithm is as follow : first decompose the matrix and then solve 2 systems with the triangular decompositions :

Data: A,b
Result: x
$L, U = LU(A)$;
solve $Ly = b$;
solve $Ux = y$;

Algorithm 5: Solve $O(N^3)$

Let's look at the following example, showing how to solve a linear system with this method.

```
PlainMatrix<> A(2,2);A(0,0)=1;A(0,1)=2;A(1,0)=-1;A(1,1)=2;
cout<<A<<endl;

/*
1     2
-1    2

*/

Vector<> b(2);b[0]=1;

LU_Decomposition<> LU(A);
cout<<LU._L<<endl;
/*
1     0
-1    1
*/
cout<<LU._U<<endl;
/*
1     2
0     4
*/
cout<<LU.Solve(b)<<endl;
//0.5     0.25
```

5.12.2 Matrix Invert Computation

Using the LU-decomposition can be used to invert a matrix. In order to explicitly compute the inverse, the LU decomposition is done then followed by $2N$ forward-backward substitutions in order to solve N systems (for each basis vector e_i). In fact we fill the i^{th} column of the matrix inverse by computing the exact solution of

$$Ax = e_i$$

This leads to a possible computation of the inverse in $O(N^3)$

5.12.3 Computing the Matrix Determinant

In order to compute the determinant of large matrices, we compute the LU-decomposition and simply use the fact that :

$$A = LU$$

$$det(A) = det(L)det(U)$$

$$\left(\prod_{i=1}^{N} L_{i,i}\right)\left(\prod_{i=1}^{N} U_{i,i}\right)$$

5.13 Iterative Methods

Direct solvers can become hard to use when dealing with large matrices, especially when sparsity is great and thus optimized algorithms are available. In order to reduce the computation cost (time and computational resources) one can use some iterative methods. Even though several methods exist, they all require some conditions (more strict than just A being invertible) in order to converge almost surely.

5.13.1 Jacobi

One of the simplest method is the Jacobi method. The idea is to decompose A into two matrices.

$$A = D + R$$

$$D = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ 0 & a_{22} & ... & 0 \\ & & ... & \\ 0 & 0 & ... & a_{nn} \end{pmatrix}, R = \begin{pmatrix} 0 & a_{12} & ... & a_{1n} \\ a_{21} & 0 & ... & a_{2n} \\ & & ... & \\ a_{n1} & a_{n2} & ... & 0 \end{pmatrix}$$

$$Ax = b \implies (D+R)x = b \implies Dx = b - Rx \implies x^{k+1} = D^{-1}(b - Rx^k)$$

$$\rho(D^{-1}R) < 1$$

Data: A,b,x_0
Result: x_i
$i = 0$;
$D = D^{-1}$;
$a = Db$;
$b = -DR$;
while *Criteria* **do**
  $x^{i+1} = a + bx^k$;
end

Algorithm 6: Jacobi

this algorithm converges if the matrix is strictly diagonally dominant. Let's look at the approximated solution generated through this algorithm for solving a simple 2×2 system :

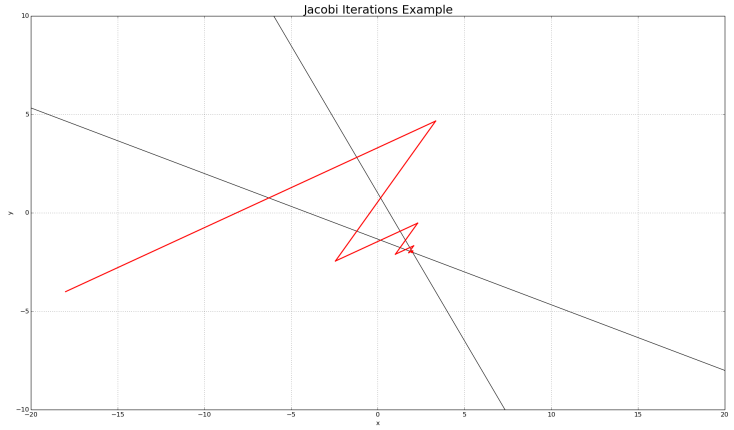

Figure 5.2: starting from $(-18, -4)$ to $(2, -2)$ in 24 iterations for an error of 1^{-10}

And now the convergence of the method :

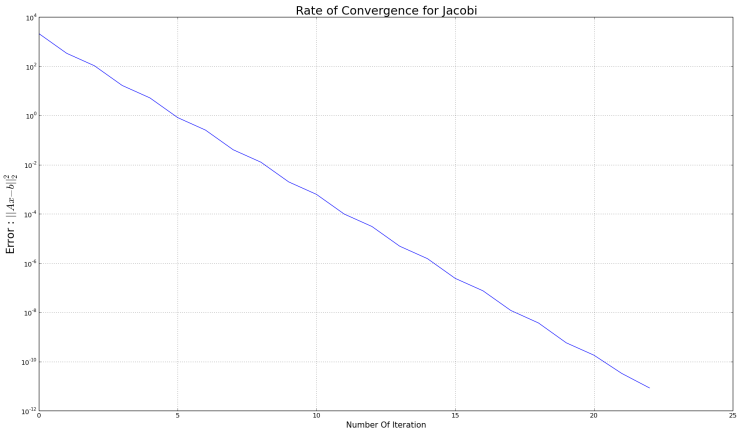

Figure 5.3: Error rate of the method which is plotted with a logarithmic y axis

5.13.2 Gauss-Seidel

We now decompose the matrix into two matrices and perform basically a similar algorithm to the Jacobi method but inside each iteration we use the already computed values for the computation of the other ones.

$$A = L + U$$

$$L = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ a_{21} & a_{22} & ... & 0 \\ & & ... & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}, U = \begin{pmatrix} 0 & a_{12} & ... & a_{1n} \\ 0 & 0 & ... & a_{2n} \\ & & ... & \\ 0 & ... & ... & 0 \end{pmatrix}$$

$$Ax = b \implies (L+U)x = b \implies Lx = b - Ux \implies x^{k+1} = L^{-1}(b - Ux^k)$$

Data: A,b,x_0
Result: x_i
$i = 0$;
$L = L^{-1}$;
$a = Lb$;
$b = -LU$;
while *Criteria* **do**
  $x^{i+1} = a + bx^k$;
end

Algorithm 7: Gauss Seidel 1

This algorithm performs a plain $matrix * vector$ and a $vector + vector$ operation. The needed storage is for a plain matrix and a vector. Another algorithm could be

Data: A,b,x_0
Result: x_i
$i = 0$;
$L = L^{-1}$;
while *Criteria* **do**
  $x^{i+1} = L(b - Ux^k)$;
end

Algorithm 8: Gauss Seidel 2

In this case, we don't store a plain matrix but two triangular matrices and we perform two triangular $matrix * vector$ operations plus a $vector - vector$ operation which is about the same as the previous algorithm. However in this case there will be more cache miss. To guarantee convergence, the matrix A needs to be strictly diagonally dominant or symmetric positive definite. It is the same as SOR method with $\omega = 1$.

Let's look at the approximated solution generated :

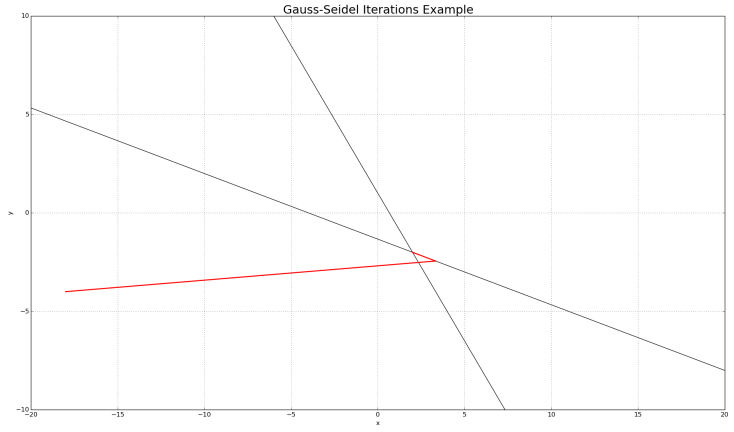

Figure 5.4: starting from $(-18, -4)$ to $(2, -2)$ in 13 iterations for an error of 1^{-12}

And now the convergence rate :

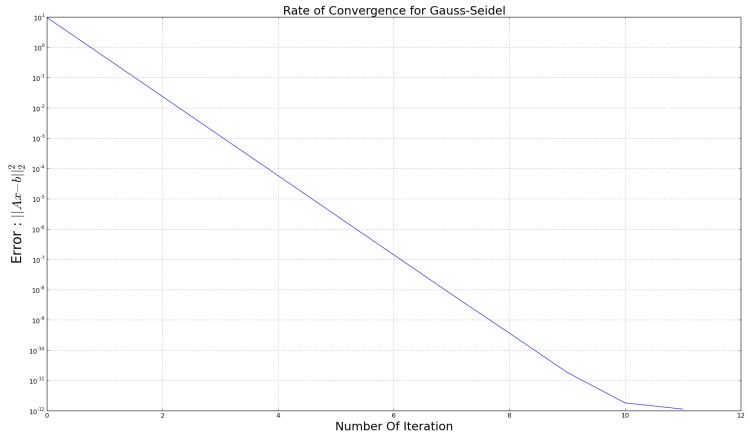

Figure 5.5: Error rate of the method which is plotted with a logarithmic y axis

here is the example used for this generation :

```
    PlainMatrix<> A(2,2);
    BandMatrix<> Band(4,4,1);Band[0]=-1;Band[1]=2;Band[2]=-1;
//      cout<<Band<<endl;
    Vector<> B(2);B[0]=2;B[1]=-8;
    Vector<> X(2);X[0]=-18;X[1]=-4;
    A(0,0)=3;
    A(0,1)=2;
    A(1,0)=2;
    A(1,1)=6;
    cout<<GaussSeidel(A,B,X,0.000000000001)<<endl;
```

5.13.3 Successive Over Relaxation

Finally another simple method based on Gauss-Seidel is the SOR-method which is not implemented in this library :

$$A = D + L + U$$

$$\begin{aligned}
& Ax = b \\
\Longrightarrow\ & \omega Ax = \omega b \\
\Longrightarrow\ & \omega \left(D + L + U\right) x = \omega b \\
\Longrightarrow\ & \left(D\omega + L\omega + U\omega\right) x = \omega b \\
\Longrightarrow\ & \left(D - (1-\omega)D + L\omega + U\omega\right) x = \omega b \\
\Longrightarrow\ & \left(D + \omega L\right) x^{k+1} = \omega b + \left((1-\omega)D - \omega U\right) x^k \\
\Longrightarrow\ & x^{k+1} = \left(D + \omega L\right)^{-1} \left(\omega b + \left((1-\omega)D - \omega U\right) x^k\right)
\end{aligned}$$

This leads to the following algorithm :

Data: A,b,ω,x_0
Result: x_i
$i = 0$;
$L = (D + \omega L)^{-1}$;
$a = \omega L b$;
$b = L\left[(1-\omega)D - \omega U\right]$;
while *Criteria* **do**
  $x^{i+1} = a + bx^k$;
end

Algorithm 9: SOR Relaxation

The difficult part in this algorithm is to find the optimal ω leading to the fastest convergence rate.

5.14 Gradient Based Methods

In order to solve the linear system $AU = F$ that will be generated in the application, we need to use an iterative method which will be useful since A will be sparse.
in the following cases, we aim to solve the system $Ax = b$ by minimizing the function

$$f(x) = x^T Ax - b^T x + c$$

Thus finding the point where the gradient is 0 :

$$\nabla f(x) = Ax - b = 0$$

$$\Delta f(x) = A$$

In order to do this we will use two methods : Gradient Descent and Conjugate Gradient Descent. In order to apply these methods, A must be symmetric positive definite. So there exists a unique minimum since the Hessian of the considered function is A. Let's first illustrate these approaches using some $2D$ examples.

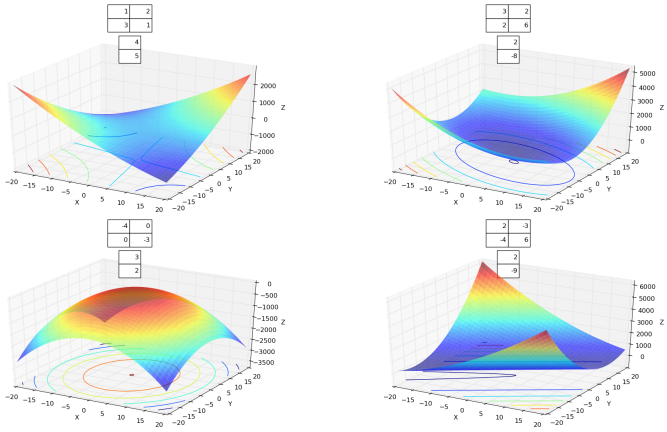

Figure 5.6: Examples of the function to minimize with different matrices

The first matrix shows a saddle point, the second one is symmetric positive definite thus has one global minimum, the third one is symmetric negative definite thus having one global maximum. Finally, the last one has no solution thus no extrema. This is why this condition is important.
Note that for all the methods, the mandatory argument is only the function to generate F but optional arguments are : diagonal preconditioning, ϵ : the stop criteria and C the number of iterations between criteria check. By default $\epsilon = 10^{-12}$ and $C = 10000$ for the GD and 100 for the CGD.

5.14.1 Gradient Descent

The main idea of the gradient descent is to find the nearest local (or in this case global) minimum by using the gradient. At each iteration we follow the gradient opposite direction and perform a line search. This line search provides the minimum of f in this direction. The scheme is as follow, go from x_k to x_{k+1} with the relation $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$ with α_k being the result of the line search :

$$\begin{aligned}
&\frac{\partial f}{\partial \alpha_k}(x_k - \alpha_k \nabla f(x_k) = 0 \\
\implies &- \nabla f(x_k) \nabla f(x_k - \alpha_k \nabla f(x_k))) = 0 \\
\implies &- \nabla f(x_k) \left(A(x_k - \alpha_k \nabla f(x_k)) - b\right)) = 0 \\
\implies &- \nabla f(x_k)^T \nabla f(x_k) + \alpha_k \nabla f(x_k)^T A \nabla f(x_k) = 0 \\
\implies &\alpha_k = \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_k)^T A \nabla f(x_k)}
\end{aligned}$$

Note that the consecutive descent directions are orthogonal.

Data: A,b,x_0
Result: x_i
$i = 0$;
while *Criteria* **do**
  $\alpha_i = \frac{\nabla f(x_i)^T \nabla f(x_i)}{\nabla f(x_i)^T A \nabla f(x_i)}$;
  $x_{i+1} = x_i - \alpha_i \nabla f(x_i)$;
  $i = i + 1$;
end

Algorithm 10: Deepest Descent Algorithm

With $\nabla f(x_k) = Ax_k - b$ and obviously $\nabla f(x_{k+1}) = Ax_{k+1} - b = \nabla f(x_k) - \alpha_i A \nabla f(x_i)$ to save one matrix vector product. So an optimized algorithm could be

Data: A,b,x_0
Result: x_k
$g = \nabla f(x_0) = Ax_0 - b$;
while *Criteria* **do**
  $p = Ag$;
  $\alpha = \frac{g^T g}{g^T p}$;
  $x- = \alpha_i g$;
  $g- = \alpha_i p$;
end

Algorithm 11: Deepest Descent Algorithm

With the last two inplace subtractions being made simultaneously in order to not compute two times the same value. Note that in this case we avoid to explicitly compute $\nabla f(x^{k+1})$ by using the recursive formula, however this can lead to more round-off errors. Also, stop criteria can be checked every C iterations and not every time. When it is being check, we also update the gradient with the standard formula to reset the accumulated round-off errors. Illustration of the method applied to the second matrix of the previous image :

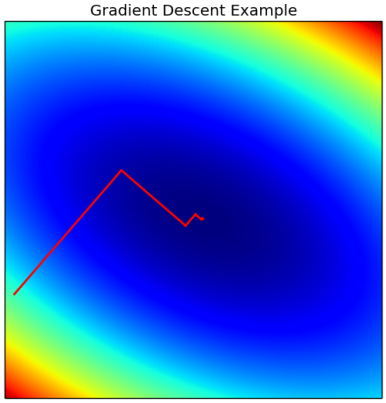

Figure 5.7: Gradient Descent iterations

Here is how to use the method (one can actually use the function GD as a stand alone by providing the input variables). This method will take a function pointer as the first input in order to create the F vector (here the function is a simple $sin(3x)$). The second (and optional) input is the diagonal preconditioning, described in 8.9.3.

```
double F(double const& x){
        return sin(3*x);
}
Mesh1D<double> mesh(0,2*M_PI,20000);
MatriceTridiagonale<double> A(20000,Triple<double> (-1,2,-1));
A._Mesh=&mesh;
A.GDSolve(F);
```

5.14.2 Conjugate Gradient Descent

The idea now is not to follow orthogonal descent directions but $A-$orthogonal directions. A vector x is $A-$orthogonal (or conjugate) if $x^T Ax = 0$. We will denote these new directions by p. The first p is the gradient, but then p_{i+1} is computed so it is a conjugate of p_i.

Data: A, b, x_0
Result: x_i
$p_0 = -\nabla f(x_0)$;
while *Criteria* **do**
  $\alpha_i = \frac{\nabla f(x_i)^T p_i}{p_i^T A p_i}$;
  $x_{i+1} = x_i - \alpha_i p_i$;
  $\lambda = \frac{\nabla f(x_{i+1})^T \nabla f(x_{i+1})}{\nabla f(x_i)^T \nabla f(x_i)}$;
  $p_{i+1} = \lambda p_i - \nabla f(x_{i+1})$;
end

Algorithm 12: CGD algorithm

With $\nabla f(x^{k+1}) = Ax^{k+1} - b = \nabla f(x^k - \alpha^k p^k) = \nabla f(x^k) - \alpha^k A p^k$

Data: A, b, x_0
Result: x_i
$x = x_0$;
$g_0 = \nabla f(x) = Ax - b$;
$p = -g$;
while *Criteria* **do**
    $t = Ap$;
    $\alpha_i = \frac{g_0^T p}{p^T t}$;
    $x- = \alpha p$;
    $g_1 = g_0 - \alpha t$;
    $\lambda = \frac{g_1^T g_1}{g_0^T g_0}$;
    $p = \lambda p - g_1$;
    $g_0 = g_1$;
end

Algorithm 13: Optimized CGD algorithm

Once again by using the recursive formula for the gradient, errors might accumulate, thus it is refreshed when checking for the stop criteria. Note that the CGD converges much faster than GD.
Illustration of the method on the same 2D matrix :

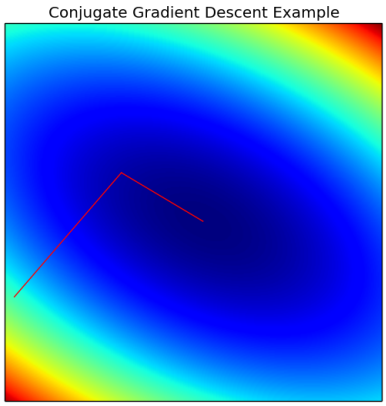

Figure 5.8: CGD iterations

5.14.3 Preconditioning

In order to have a solving method with faster convergence, one can use preconditioning on the matrix A and the vector F so that A is better conditioned. The simplest technique is to use a diagonal matrix which is the inverse of the diagonal of A and so transforming the problem into solving :

$$P = diag(A); P^{-1}x = P^{-1}F$$

Thus in the GD and CGD the optional second argument is a vector which will multiply A and the vector F element-wise as if P was a diagonal matrix with this vector of values.
Note that using the preconditioning implies changes in the matrix, thus if one wants to consecutively solve systems, one has to check the matrix values.

5.14.4 Example

Let's look at an example by solving a tridiagonal matrix given by :

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \dots & & \\ & & -1 & 2 \end{pmatrix}$$

using gradient based methods (with a size of 20000×20000. We will first see the convergence difference between the Conjugate Gradient Descent and the Gradient Descent with and without preconditioning. I present here some plots of the errors for the given problem given the specific solving algorithms presented before. We can clearly see the huge impact of preconditioning on the CGD and the GD. However the CGD is assured to reach the solution in at most N iterations but the GD can need far more iterations to reach the stop criteria.

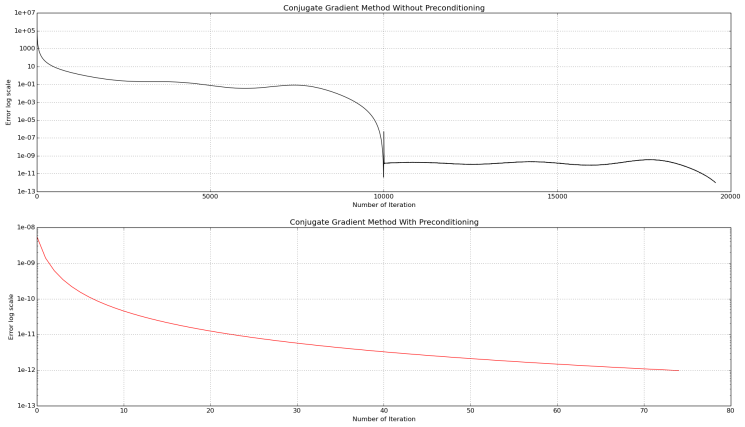

Figure 5.9: Errors vs Number of iterations for the CGD with and without preconditioning, log scale for Y axis

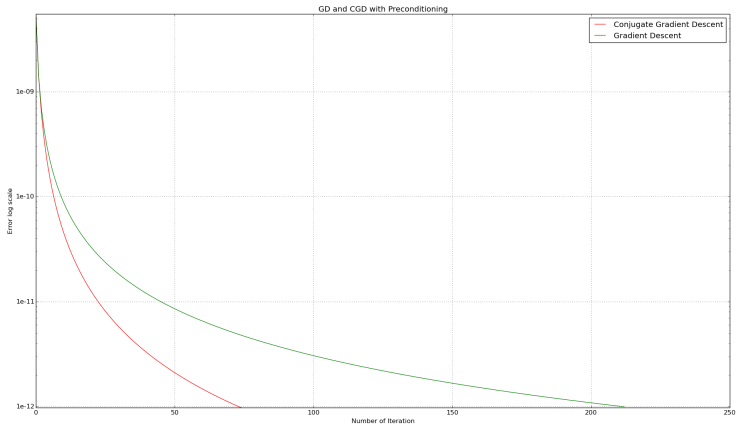

Figure 5.10: Difference between CGD and GD with preconditioning on solving the system

Part V

Validation on a Finite Element Approximation

Chapter 6

Initial Problem

The goal is to study the heat distribution in a 2-dimensional squared domain Ω through Laplace's equation. The domain boundary is divided into four parts ($\Sigma_1 := \{0\}\times]0,1[$, $\Sigma_2 :=]0,1[\times\{1\}$, $\Sigma_3 := \{1\}\times]0,1[$, $\Sigma_4 :=]0,1[\times\{0\}$) corresponding to the four sides of the domain.
The heat conductivity of the material on the domain is given by the function α which is defined by parts.
There are two boundary conditions : one part with a homogeneous Dirichlet condition (on the bottom side of the domain) meaning that the temperature of the material is kept to 0 on this side; the remaining boundary has a Neumann condition, symbolizing the heat transfer between the material and the exterior (another material for example).

- The Initial Problem (PI) :

$$\begin{cases} \text{Find } u \in H^2_D(\Omega) \text{ such that} \\ -\alpha\Delta u = 0 \text{ on } \Omega \text{ with } \alpha \in L^\infty(\Omega) \\ \frac{\partial u}{\partial n} = g \text{ on } \Gamma_N := \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \text{ with } g \in L^\infty(\Gamma_N) \\ u = 0 \text{ on } \Gamma_D := \Sigma_4 \end{cases} \tag{6.1}$$

  Where $H^2_D(\Omega) := \{v \in H^2(\Omega) : v|_{\Gamma_D} = 0\}$ with $.|_{\Gamma_D}$ being the trace operator.

- The function g defined on Γ_N with $g_i : \mathbb{R} \to \mathbb{R}, i = 1,2,3$:

$$g : \Gamma_N \subset \mathbb{R}^2 \to \mathbb{R}$$
$$(x,y) \to \begin{cases} g_1(y), \text{ if } (x,y) \in \Sigma_1 \\ g_2(x), \text{ if } (x,y) \in \Sigma_2 \\ g_3(y), \text{ if } (x,y) \in \Sigma_3 \end{cases}$$

- The function α defined on Ω with $\alpha_i : \mathbb{R} \to \mathbb{R}, i = 1,...,9$:

$$\alpha : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$$
$$(x,y) \to \alpha_i(x,y), \text{ if } (x,y) \in \Omega_i, i = 1,...,9$$

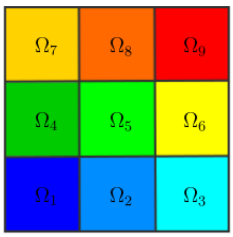

Figure 6.1: Sub-domains corresponding to Ω_i representing the different conductivity levels.

Chapter 7

Variational Problem

7.1 Weak Form Derivation

We will now derive the variational formulation of the initial problem 9.1

$$
\begin{aligned}
&\forall v \in H^1_D(\Omega) \\
&\int_\Omega (-\alpha \Delta u) v dx dy = 0 \\
\implies &-\int_{\partial\Omega} (\alpha|_{\partial\Omega} \nabla u . \overrightarrow{n}) v|_{\partial\Omega} d\sigma + \int_\Omega (\alpha \nabla u).(\nabla v) dx dy = 0 \text{ (Green)} \\
\implies &-\int_{\Gamma_N} \alpha|_{\Gamma_N} \frac{\partial u}{\partial n} v|_{\Gamma_N} d\sigma - \int_{\Gamma_D} \alpha|_{\Gamma_D} \frac{\partial u}{\partial n} v|_{\Gamma_D} d\sigma + \int_\Omega (\alpha \nabla u).(\nabla v) dx dy = 0 \\
\implies &\int_\Omega (\alpha \nabla u).(\nabla v) dx dy = \int_{\Gamma_N} \alpha|_{\Gamma_N} g v|_{\Gamma_N} d\sigma \text{ (Boundary Conditions)}
\end{aligned}
$$

Leading to the variational problem

$$
\begin{cases} \text{Find u} \in H^1_D(\Omega) \text{ such that} \\ a(u,v) = l(v), \forall v \in H^1_D(\Omega) \end{cases} \tag{7.1}
$$

We now have 9.1 $\implies$ 10.1 with :

$$a(u,v) := \int_\Omega (\alpha \nabla u).(\nabla v) dx dy$$

$$l(v) := \int_{\Gamma_N} \alpha|_{\Gamma_N} g v|_{\Gamma_N} d\sigma$$

7.2 Problem Equivalence

We will now check that 10.1 $\implies$ 9.1 assuming that $u \in H^2(\Omega)$

$$
\begin{aligned}
&\forall \phi \in \mathcal{D}(\Omega) \subset H_D^1(\Omega), \\
&\int_\Omega (\alpha \nabla u).(\nabla \phi) dxdy = \int_{\Gamma_N} \alpha|_{\Gamma_N} g\phi|_{\Gamma_N} d\sigma \\
\implies &\int_\Omega (\alpha \nabla u).(\nabla \phi) dxdy = 0 \text{ (Compact Support of } \phi \\
\implies &\int_{\partial\Omega} (\alpha \nabla u.\overrightarrow{n})\phi|_{\partial\Omega} d\sigma - \int_\Omega \alpha \Delta u \phi dxdy = 0 \\
\implies &-\int_\Omega \alpha \Delta u \phi dxdy = 0, \forall \phi \in \mathcal{D}(\Omega) \\
\implies &-\alpha \Delta u = 0 \text{ almost everywhere on } \Omega
\end{aligned}
$$

Let's find the Neumann boundary condition (the Dirichlet condition is already included into $H_D^1(\Omega)$) :

$$
\begin{aligned}
&\forall v \in H_D^1(\Omega), \\
&\int_\Omega (\alpha \nabla u).(\nabla v) dxdy = \int_{\Gamma_N} \alpha|_{\Gamma_N} gv|_{\Gamma_N} d\sigma \\
\implies &\int_{\partial\Omega} (\alpha|_{\partial\Omega} \nabla u.\overrightarrow{n})v|_{\partial\Omega} d\sigma - \int_\Omega (\alpha \Delta u) v dxdy = \int_{\Gamma_N} \alpha|_{\Gamma_N} gv|_{\Gamma_N} d\sigma \\
\implies &\int_{\Gamma_N} \alpha|_{\Gamma_N} \nabla u.\overrightarrow{n} v|_{\Gamma_N} d\sigma = \int_{\Gamma_N} \alpha|_{\Gamma_N} gv|_{\Gamma_N} d\sigma, \forall v \in H_D^1 \\
\implies &\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N
\end{aligned}
$$

Thus we have : 9.1 $\iff$ 10.1

7.3 Existence and Uniqueness of the Solution

We will now prove the uniqueness of the solution through the Lax-Milgram's Theorem :

- $H_D^1(\Omega)$ is a Hilbert space with the scalar product $< .,. >_{H^1(\Omega)}$. In fact, $H_D^1(\Omega) := ker(.|_{\Gamma_D})$. Since the trace operator is a linear and continuous operator, $H_D^1(\Omega)$ is closed. In addition, $H_D^1(\Omega) \subset H^1(\Omega)$ thus $H_D^1(\Omega)$ is a Hilbert space.
- a and l are bilinear and linear respectively by definition of the integral.
- l is continuous :

$$
\begin{aligned}
&\forall v \in H_D^1(\Omega) : \\
&|l(v)| \leq ||\alpha|_{\Gamma_N}||_{L^\infty(\Gamma_N)} ||g||_{L^2(\Gamma_N)} ||v|_{\Gamma_N}||_{L^2(\Gamma_N)} \\
&|l(v)| \leq ||\alpha|_{\Gamma_N}||_{L^\infty(\Gamma_N)} ||g||_{L^2(\Gamma_N)} K||v||_{H^1(\Omega)} \\
&|l(v)| \leq C||v||_{H^1(\Omega)}
\end{aligned}
$$

With K being the continuity constant of the trace operator :

$$||u|_\Gamma||_{L^2(\Gamma)} \leq K||u||_{H^1(\Omega)}$$

- a is continuous :

$$\begin{aligned}
&\forall u, v \in H^1_D(\Omega) : \\
&|a(u,v)| \leq ||a||_{L^\infty(\Omega)}||\nabla u||_{L^2(\Omega)}||\nabla v||_{L^2(\Omega)} \\
&|a(u,v)| \leq ||a||_{L^\infty(\Omega)}||u||_{H^1(\Omega)}||v||_{H^1(\Omega)} \\
&|a(u,v)| \leq C||u||_{H^1(\Omega)}||v||_{H^1(\Omega)}
\end{aligned}$$

- a is coercive, if we require $\alpha \geq \alpha_0 > 0$:

$$\begin{aligned}
&\forall u \in H^1_D(\Omega) : \\
&a(u,u) = \int_\Omega \alpha|\nabla u|^2 dxdy \\
&a(u,u) \geq \alpha_0||\nabla u||^2_{L^2(\Omega)} \\
&a(u,u) \geq \frac{\alpha_0}{1+C_P^2}||u||^2_{H^1(\Omega)} \\
&a(u,u) \geq C||u||^2_{H^1(\Omega)}
\end{aligned}$$

Thus using Poincare's inequality leads to the coercivity through the condition $\alpha > 0$ which is not a problem since in our case α stands for the heat conductivity of the material. Note that we can use the Poincare's inequality because of the Dirichlet condition :

$$||f||_{L^2(\Omega)} \leq C_p||\nabla f||_{L^2(\Omega)}$$

Thus we can derive :

$$\begin{aligned}
||u||^2_{H^1(\Omega)} &= ||u||^2_{L^2(\Omega)} + ||\nabla u||^2_{L^2(\Omega)} \\
&\leq C_p^2||\nabla u||^2_{L^2(\Omega)} + ||\nabla u||^2_{L^2(\Omega)} \\
&\leq (1+C_p^2)||\nabla u||_{L^2(\Omega)} \\
||u||^2_{H^1(\Omega)}\frac{1}{(1+C_p^2)} &\leq ||\nabla u||^2_{L^2(\Omega)}
\end{aligned}$$

Therefore all the conditions for the Lax-Milgram's Theorem are checked, we thus have :

$$\exists! u \in H^1_D(\Omega) : a(u,v) = l(v), \forall v \in H^1_D(\Omega)$$

Chapter 8

Discretized Problem

8.1 Existence and Uniqueness of the Solution

We will use the Galerkin method in order to discretize the problem and solve it. Given a triangulation of Ω, we define :

- P^1 is the set of all the degree one polynomials $(a + bx + cy)$.
- $v|_{T_i}$ will be the restriction of v on the i^{th} triangle.
- $V_h(\Omega) := \{v_h \in C^0(\bar{\Omega}) : v_h|_{T_i} \in P^1(T_i), v_h|_{\Gamma_D} = 0\}$
- $V_h(\Omega) \subset V(\Omega) := H^1_D(\Omega)$

Since we know that :

- $V_h(\Omega)$ is a finite dimension space, thus is closed
- $V_h(\Omega) \subset H^1_D(\Omega)$

we can use again the Lax Milgram's theorem giving the existence and uniqueness of the solution to the discretized problem with 11.1 $\iff$ 10.1.

$$\begin{cases} \text{Find } u_h \in V_h \text{ such that} \\ a(u_h, v_h) = l(v_h), \forall v_h \in V_h \end{cases} \tag{8.1}$$

8.2 Mesh Description

Given a discretization of the domain into a $2D$ mesh with nodes x_i (note necessarily equidistant) the degrees of freedom are $\{x_i \in \Omega : x_i \notin \Gamma_D\}$. In the specific case of a uniform mesh and with

- $\Omega = [0, 1]^2$
- N points in x, $h_x = \frac{1}{N-1}$
- M points in y, $h_y = \frac{1}{M-1}$
- $\dim(V_h) = MN - N$ because of the Dirichlet condition

- $x_i = ((i \mod N)h_x, floor(i/N)h_y), 0 \leq i \leq MN - 1$

Thus leading to the following node labelling where we list the nodes x_i from left to right and from bottom to top.

$$\begin{array}{cccc} x_{(M-1)N} & x_{(M-1)N+1} & \dots & x_{MN-1} \\ & \dots & & \\ x_N & x_{N+1} & \dots & x_{2N-1} \\ x_0 & x_1 & \dots & x_{N-1} \end{array}$$

Here is a little mesh example with $N = M = 3$:

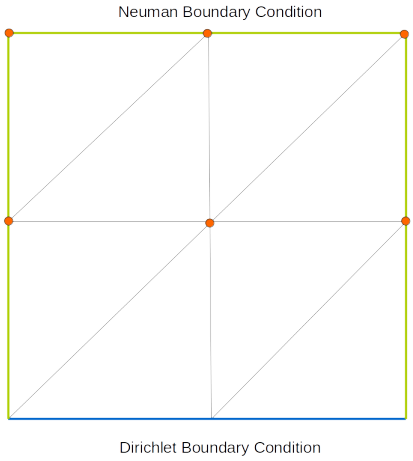

Figure 8.1: Mesh example, Orange : degrees of freedom

We use a uniform mesh but the used algorithms are not specialized. Each node needs a label in order to know if it belongs to the boundary or not. We also assume the Dirichlet condition. The labels are necessary for example to know if a triangle belongs to the boundary (to know how many of its three defining points belong to the boundary) for the linear form computation.

Since we will use P^1 Lagrange polynomials, the basis functions are the hat functions w^i with

$$w^{(i)}(x_k) = \delta_{ik} = \begin{cases} 1, \text{if } i = k \\ 0, \text{otherwise} \\ i, k \in \{N, ..., NM - 1\} \end{cases}$$

Those functions form a basis of $V_h(\Omega)$, in fact

- $w^{(i)} \in V_h(\Omega), \forall i$

- $|\{w^{(i)}, i = N, ..., NM-1\}| = (M-1)N = dim(V_h)$
- $(w^{(i)})_{i=N}^{NM-1}$ are linearly independent.

We just need to show that they are linearly independent :

$$\begin{aligned} & \sum_{i=N}^{NM-1} w^{(i)}(x,y)\lambda_i = 0 \\ \implies & \sum_{i=N}^{NM-1} w^{(i)}(x_k)\lambda_i = 0, \forall k \in \{N, ..., NM-1\} \\ \implies & \lambda_i = 0, \forall i \end{aligned}$$

8.3 System $Au = L$

We will now see how this discretized problem can be solved using a linear system. We already know that :

$$\exists! u_h \in V_h(\Omega) : a(u_h, v_h) = l(v_h), \forall v_h \in V_h(\Omega)$$

This implies directly that

$$\exists! u_h \in V_h(\Omega) : a(u_h, w^{(i)}) = l(w^{(i)}), \forall i \in \{N, ..., NM-1\}$$

Now by using the fact that we have a basis for $V_h(\Omega)$ we can rewrite any function as a linear combination of this basis leading to $u_h = \sum_{i=N}^{NM-1} u_h(x_i)w^{(i)}$.
Now by using the bilinearity of a we have :

$$\begin{aligned} & a(u_h, w^{(i)}) = l(w^{(i)}), \forall i \in \{N, ..., NM-1\} \\ \implies & a(\sum_{j=N}^{NM-1} u_h(x_j)w^{(j)}, w^{(i)}) = l(w^{(i)}), \forall i \in \{N, ..., NM-1\} \\ \implies & \sum_{j=N}^{NM-1} u_h(x_j)a(w^{(j)}, w^{(i)}) = l(w^{(i)}), \forall i \in \{N, ..., NM-1\} \\ \implies & Au = L \end{aligned}$$

With

$$A = \begin{pmatrix} a(w^{(N)}, w^{(N)}) & a(w^{(N+1)}, w^{(N)}) & ... & a(w^{(NM-1)}, w^{(N)}) \\ a(w^{(N)}, w^{(N+1)}) & a(w^{(N+1)}, w^{(N+1)}) & ... & a(w^{(NM-1)}, w^{(N+1)}) \\ & & ... & \\ a(w^{(N)}, w^{(NM-1)}) & a(w^{(N+1)}, w^{(NM-1)}) & ... & a(w^{(NM-1)}, w^{(NM-1)}) \end{pmatrix}$$

$$u = \begin{pmatrix} u_h(x_N) \\ u_h(x_{N+1}) \\ ... \\ u_h(x_{NM-1}) \end{pmatrix}, L = \begin{pmatrix} l(w^{(N)}) \\ l(w^{(N+1)}) \\ ... \\ l(w^{(NM-1)}) \end{pmatrix}$$

Thus we end up by simply having a linear system to solve of size $N(M-1) \times N(M-1)$ but we will see that the matrix A is sparse.

Chapter 9

Discretized Problem Computation

9.1 Quadrature

9.1.1 Triangle Area

In order to quickly compute the area of a triangle (T_k) given its three nodes, we use the following formula :

$$area(T_k) = \frac{|(y_3 - y_1)(x_2 - x_1) - (y_2 - y_1)(x_3 - x_1)|}{2}$$

In fact, if we have three points in a $2D$ space (defining the triangle) the determinant of the matrix

$$\begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$$

divided by 2 will give us the (signed) area of the triangle, thus we take the absolute value.

9.1.2 Hat Functions

By the nature of the basis functions ($w^{(i)}$), also called the "hat" functions because of their shape, we have some properties :

- $w^{(i)}|_{T_i} \in P^1(T_i), \forall i$
- $\nabla w^{(i)}|_{T_i} \in P^0(T_i), \forall i$
- $w^{(i)}(x, y) = 0$ and thus $\nabla w^{(i)}(x, y) = 0$ if $(x, y) \notin$ adjacent triangles of x_i

Let's compute the explicit expression of $w^{(i)}$ on any triangle T_k adjacent to x_i since otherwise $w^{(i)} = 0$. We know that $w^{(i)}(x_i) = 1$ and that a triangle is defined by its three nodes (where one of them is x_i) $(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)$. We will thus compute the plane equation given its normal vector and one of the

points.
The two vectors computed from the triangles are

$$\begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{pmatrix}, \begin{pmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \end{pmatrix}$$

We now compute the cross product by finding the determinant of the following matrix

$$\begin{pmatrix} i & j & k \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{pmatrix}$$

Leading to the normal vector :

$$\begin{pmatrix} (y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1) \\ -(x_2 - x_1)(z_3 - z_1) + (z_2 - z_1)(x_3 - x_1) \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \end{pmatrix} := \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

Thus the equation of the plane is of the form

$$a_1 x + a_2 y + a_3 z = d$$

where d is determined by plugging in one of the three nodes leading to the formula

$$\forall (x, y) \in T_k \subset supp(w^{(i)}), w^{(i)}(x, y) = \frac{-a_1 x - a_2 y + (a_1 x_i + a_2 y_i + a_3 z_i)}{a_3}$$

With T_k being a triangle having x_i as one of its node. This allows us to compute explicitly the gradient :

$$\forall (x, y) \in T_k \subset supp(w^{(i)}), \nabla w^{(i)}(x, y) = \begin{pmatrix} \frac{-a_1}{a_3} \\ \frac{-a_2}{a_3} \end{pmatrix} := \begin{pmatrix} g^i_{1,k} \\ g^i_{2,k} \end{pmatrix}$$

9.2 Bilinear Form Computation

We will now investigate the bilinear form which will be used to fill the matrix A. We will denote by $T_k[i], i = 1, 2, 3$ the node i of the triangle k.

$$\begin{aligned} a(w^{(i)}, w^{(j)}) &= \int_\Omega (\alpha \nabla w^{(i)}).(\nabla w^{(j)}) dxdy \\ &= \sum_{k=0}^{K-1} \int_{T_k} \alpha (\nabla w^{(i)}.\nabla w^{(j)}) dxdy \\ &= \sum_{k=0}^{K-1} (g^i_{1,k} g^j_{1,k} + g^i_{2,k} g^j_{2,k}) \int_{T_k} \alpha dxdy \\ &= \sum_{k=0}^{K-1} (g^i_{1,k} g^j_{1,k} + g^i_{2,k} g^j_{2,k}) * \frac{\alpha(T_k[0]) + \alpha(T_k[1]) + \alpha(T_k[2])}{3} * area(T_k) \end{aligned}$$

If x_i and x_j are not nodes of the triangle T_k it is equal to 0. Note that the last approximation is actually an exact quadrature except if T_k belongs to two

different sub-domains Ω_i. So we will loop over the triangles to fill A and add the new computed contribution to the right position. Note that since α is constant on each Ω_i we could have used a faster quadrature formula by simply using the middle point for example. However, in a case where we have triangles on two sub-domains, this quadrature reduces the error.
In the particular case of a uniform mesh, the matrix will be tridiagonal by tridiagonal blocks.

9.3 Linear Form Computation

$$l(w^{(i)}) = \int_{\Gamma_N} \alpha|_{\Gamma_N} g w^{(i)}|_{\Gamma_N} d\sigma$$

For most of the $w^{(i)}$ functions, we will have $l(w^{(i)}) = 0$. But we can efficiently compute it knowing that this will be non-zero if and only if $x_i \in \Gamma_N$. We can already say that

$$\begin{aligned} x_i \in \Gamma_N \iff i \in &\{N, 2N, ..., (M-1)N, \\ &(M-1)N+1, ..., NM-1, \\ &(M-1)N-1, (M-2)N-1, ..., 2N-1\} := X \end{aligned}$$

We can even see that by looking only at the nodes on the boundary, the node $x_{X[s]}$ has two adjacent nodes in X : $x_{X[s-1]}$ and $x_{X[s+1]}$ with $s = 1, ..., 2M + N - 5$ in fact $|X| = 2M + N - 4$. And so we can see that if $i \in X$ we will need to compute the area under two segments to compute $l(w^{(i)})$. We then have the following formula if we note $w^{(X[i])} := w$ and we assume $h_y = h_x = h$ for convenience (otherwise we need to distinguish the case $x_{X[i]} \in \Sigma_1 \cup \Sigma_3$ from $x_{X[i]} \in \Sigma_2$) :

$$\begin{aligned} l(w) =& h * \frac{\alpha(x_{X[i-1]}) * g(x_{X[i-1]}) * w(x_{X[i-1]}) + \alpha(x_{X[i]}) * g(x_{X[i]}) * w(x_{X[i]})}{2} + \\ & h * \frac{\alpha(x_{X[i+1]}) * g(x_{X[i+1]}) * w(x_{X[i+1]}) + \alpha(x_{X[i]}) * g(x_{X[i]}) * w(x_{X[i]})}{2} \\ =& h * \frac{\alpha(x_{X[i]}) * g(x_{X[i]}) * w(x_{X[i]})}{2} + h * \frac{\alpha(x_{X[i]}) * g(x_{X[i]}) * w(x_{X[i]})}{2} \\ =& h * \alpha(x_{X[i]}) * g(x_{X[i]}) * w(x_{X[i]}) \end{aligned}$$

By using the fact that $w(x_{X[k]}) = \delta_{X[i]X[k]}$. Since on each $\Omega_i \cap \Gamma_N$ we have $\alpha \in P^0$, $g \in P^0$, $w^{(i)} \in P^1$ the used quadrature formula is exact. This would not be the case without these properties. Let's look at an example with the 3×3 mesh and with $w^{(3)}|_{\partial\Omega}$

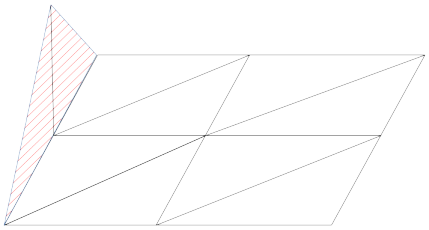

Figure 9.1: Example of $w^{(3)}|_{\Gamma_N}$

9.4 Implementation of Piecewise-Defined Functions

9.4.1 α function

First of all, we have the function α which is defined by parts with the functions $\alpha_i : \Omega_i \subset \mathbb{R}^2 \to \mathbb{R}$. This comes from the heat conductivity of the material which is not constant on the whole domain. This is for example the case if the studied material is actually itself a composite of some pieces each having different properties. In order to keep a general approach to this, we define a special class *alpha* which will hold all the sub-domains and the corresponding values. Then, when we use the access operator, for a specific $\mathbb{R}^2$ point, a special method will find to which sub-domain this point belongs to and then will return the right value. Here is an example :

```
alpha al;
al.AddDomain(0,0,0.3,0.5,1);
al.AddDomain(0.3,0,0.6,0.5,6);
al.AddDomain(0.6,0,1,0.5,1);
al.AddDomain(0,0.5,1,1,1);
al(0.2,0.2);
```

Note that the domains are rectangular domains defined by the two extreme points and the corresponding return value. A generalisation for every shape is possible but then in order to check to which sub-domain a specific point belongs to we will need a more complex method. A default constructor will automatically set the coefficients and the sub-domains as given in the studied problem.

9.4.2 g function

Another function which is defined by parts is the function g. This function represents the heat flow coming in or out (depending on the sign of the value) the boundary of the material (except on $[0,1] \times \{0\}$ where the temperature is kept at 0. This function is defined by a specific $g_i : \mathbb{R} \to \mathbb{R}$ function for each side of Ω (except on Γ_D). In order to have a simple class for this function, we just store the three possible values (one for each side) and then when a point is requested, a simple condition will return the right value.
Note that it would be possible to generalize this to any given case (for example a non constant function g_i) by storing the function pointer and returning the specific function return value instead of simply returning the stored value.

9.5 Method to Fill the System $Au = L$

9.5.1 Optimized Approach

Since we already know that A will be sparse by the structure of the mesh we can avoid doing an algorithm of the form :

Data: Matrix, Vector
Result: A filled
for *i=N → NM-1* **do**
  for *j=0 → NM-1* **do**
    $A(i,j) = a(w^{(i)}, w^{(j)})$
  end
end

Algorithm 14: Filling the matrix A in $O(NM^2)$

Another approach is by computing the value of a on each triangle and then add the result to the right entry of the matrix A. Note that by doing this it is also possible to fill the vector L at the same time.

Data: Matrix, Vector
Result: A and L filled
for *k=0 → K-1* **do**
  for *i=0 → 2* **do**
    if $Triangle[k][i] - N \geq 0$ **then**
      $L[Triangle[k][i] - N] += b(Triangle[k], i, al, g)$;
      for *j=0 → 2* **do**
        if $Triangle[k][j] - N \geq 0$ **then**
          $A(T[k][i] - N, T[k][j] - N) += a(Triangle[k], i, j, al)$
        end
      end
    end
  end
end

Algorithm 15: Filling the matrix A in $O(K)$ (about $9 * K$ loops)

Let's see the resulting matrix for two different mesh structures to show the

special "tridiagonal by tridiagonal blocks" structure defined as :

$$\begin{pmatrix} B & I & & \\ I & B & I & \\ & & \ddots & \\ & & I & B \end{pmatrix}$$

with B being once again a tridiagonal matrix.

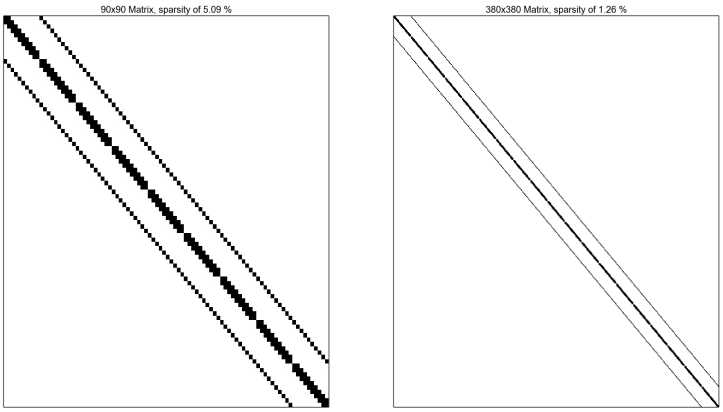

Figure 9.2: Example for two matrices.

To illustrate the fill-in induced by a LU decomposition on a sparse (tridiagonal) matrix, let's look at the LU decomposition of the matrix A :

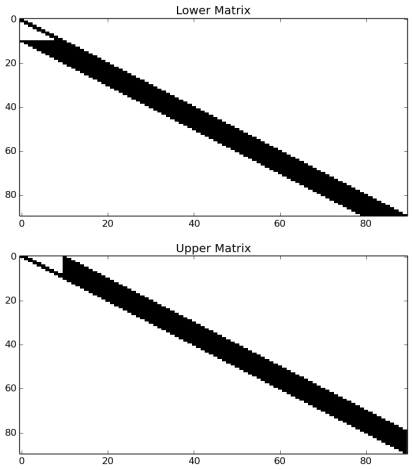

Figure 9.3: Example for two matrices.

9.5.2 Computation time,Convergence, Optimization

Since we have a uniform mesh we can optimize the computation by not calculating every time the area of the triangles. In fact, we have by definition, triangles with the same area, only a rotation is applied.
The time to fill the system is not problematic though. Here is a plot of the timed needed to fill the system compared to the number of triangles and the theoretical needed time :

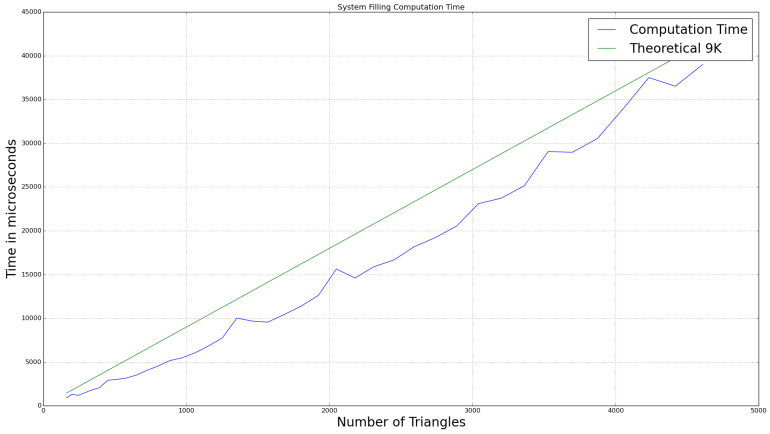

Figure 9.4: Time necessary to fill the system (actual time and theoretical $O(9 * K)$

Let's now look at the required number of iterations for the solvers to converge to a solution. The error criteria taken is 1^{-9}. Not that no preconditioning was use in this plot but this can be used with the gradient descent. however, when a preconditioning is applied to the conjugate gradient method, the convergence is not assured. This might be because the matrix A which structure depends on α is not always positive definite.

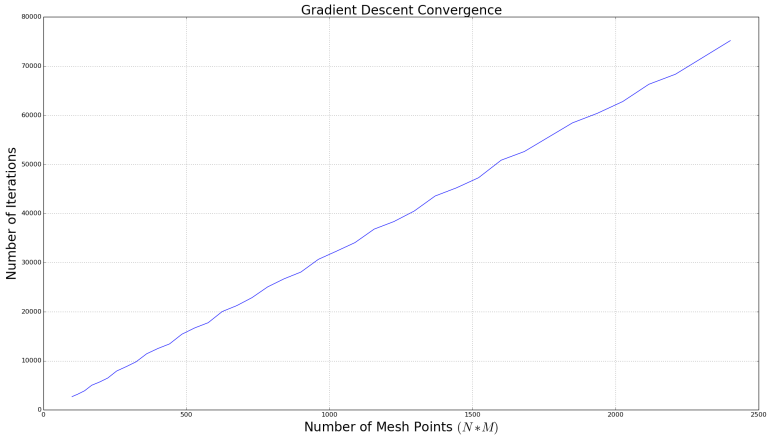

Figure 9.5: Number of iterations to reach an error of 1^{-9} with gradient descent

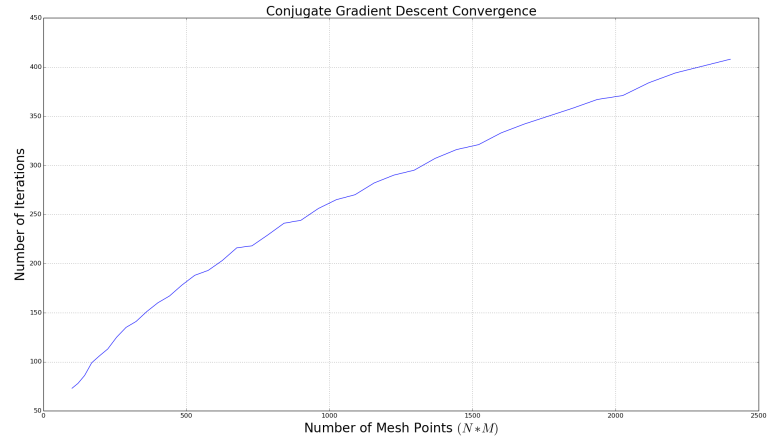

Figure 9.6: Number of iterations to reach an error of 1^{-9} with conjugate gradient descent

Chapter 10

Examples and GUI

10.1 GUI

In order to facilitate the user experience, a simple Graphical Interface is provided for this project allowing quick tests. All the main parameters can be modified :

- the α function
- the g function
- the mesh architecture
- the solver properties

The visualization is then done in this exact interface using Qt utilities (very similar to openGL).
Given the mesh points (x_i, y_i, z_i) for every points, the actual solution is plotted as a colored surface with a colormap (the color stands for the z-coordinate). For each triangle, the inside points color (needed to draw all the pixels inside the triangle) are interpolated using a bilinear interpolation according to the fact that $v_h|T_k \in P^1(T_k)$. This results in very nice looking visualization even with just a few mesh points as it can be seen below.
Let's now look at this interface and the provided utilities.

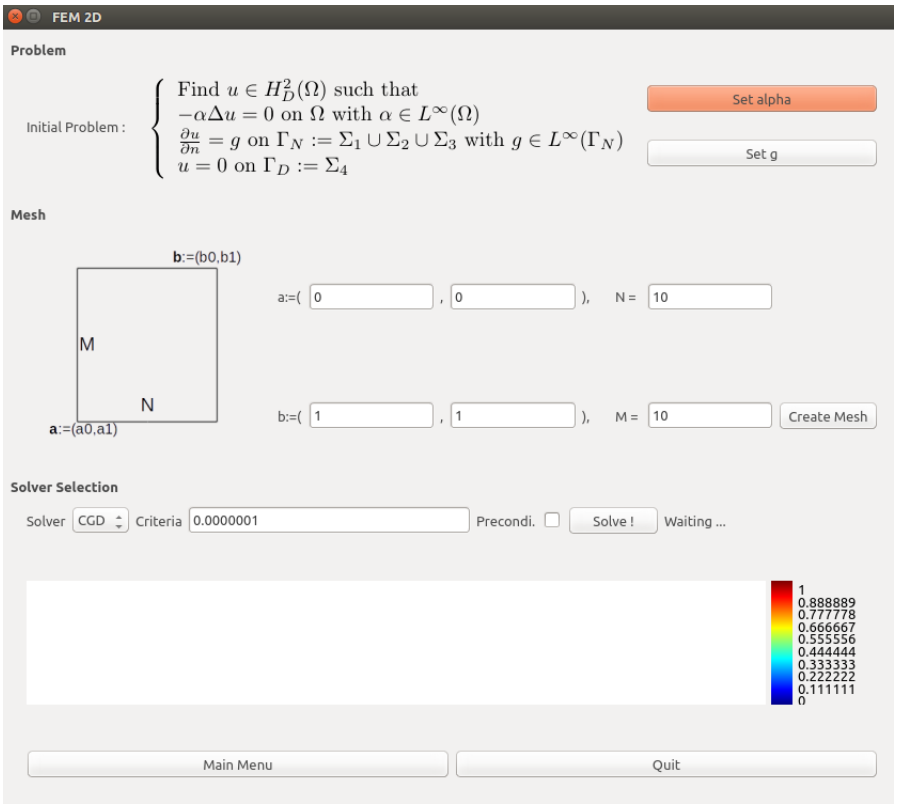

Figure 10.1: Here is the main interface where every parameter can be changed and the visualization is done.

When clicking on the "Set alpha" button, a window appears where it is possible to set the value of α on every Ω_i :

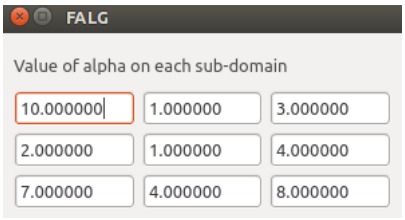

Figure 10.2: Pop-up window for setting the α function

Similarly, it is possible to set the g function :

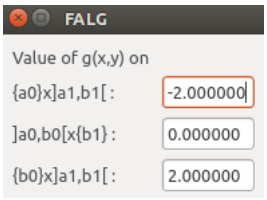

Figure 10.3: Pop-up window for setting the g function

Finally, the last parameter to set for the discretized problem is the number of points to use to discretize the domain. This is done by setting the parameters N, M. note that it is also possible to change the domain $]0,1[\times]0,1[$. When this is set, clicking on "Create Mesh" will generate the triangular mesh and draw it inside a pop up window :

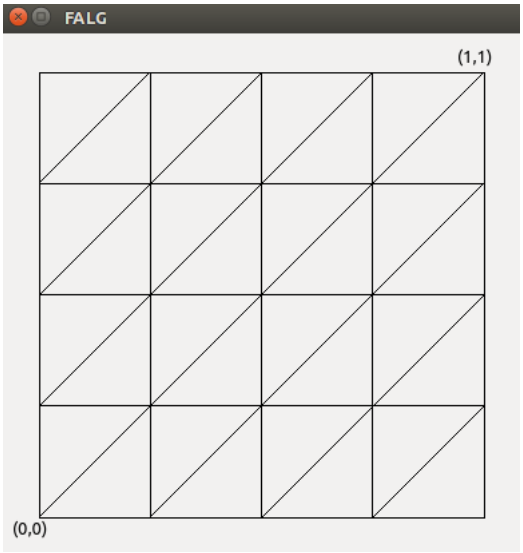

Figure 10.4: Pop-up window showing the generated mesh, for a 5×5 mesh in this case

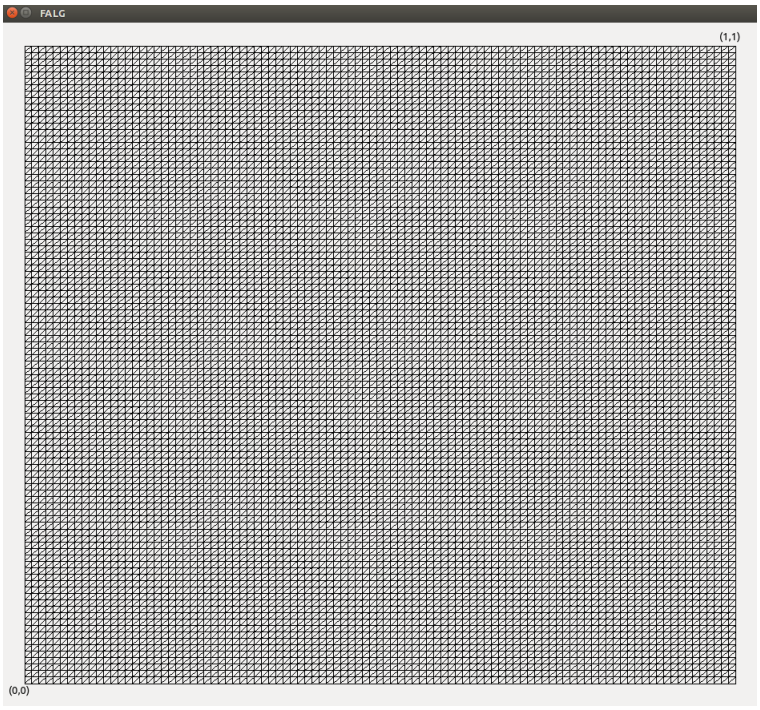

Figure 10.5: Same with a much finer mesh : 100×100

Now setting the solver (or using the default parameters) and clicking on "Solve" will launch the solver with the generated parameters and when the approximation will be computed, the rendering process will start displaying the solution in the bottom part as shown here :

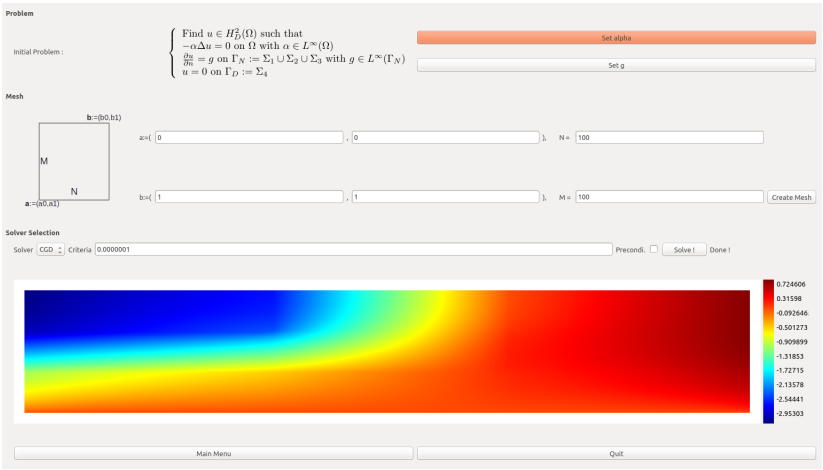

Figure 10.6: Finally the visualization of the solution after having launched the resolution process

10.2 Examples

Now we will look at some more examples with other parameters, for each example are provided α, g :

10.2.1 Example 1

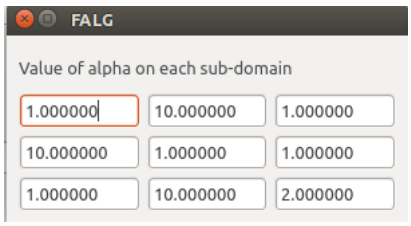

Figure 10.7: Example 1 α function

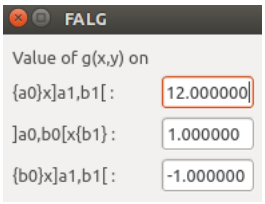

Figure 10.8: Example 1 g function

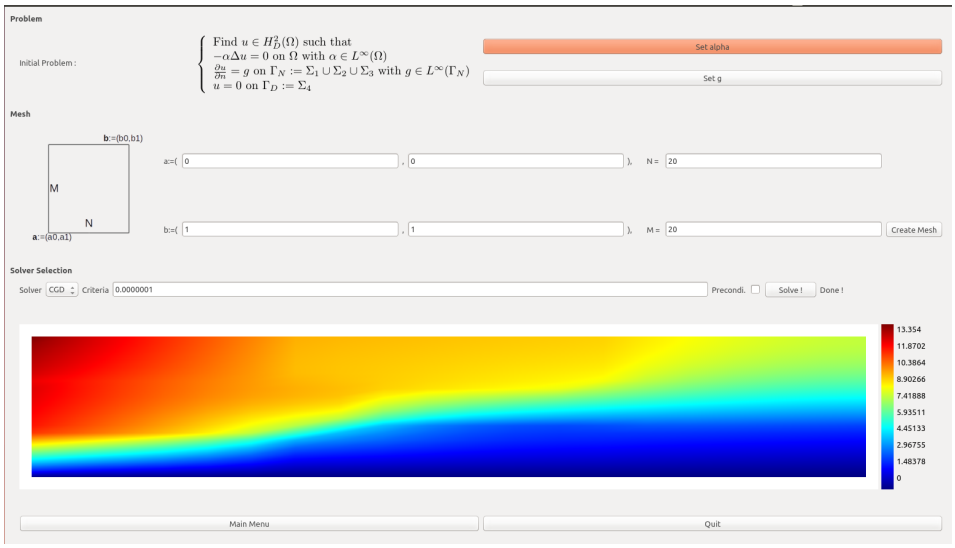

Figure 10.9: Example 1 approximated solution

10.2.2 Example 2

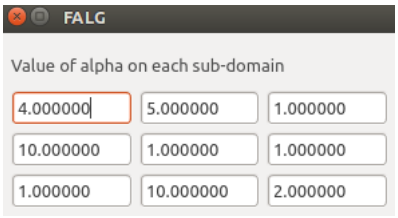

Figure 10.10: Example 2 α function

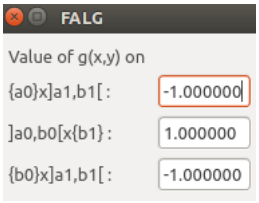

Figure 10.11: Example 2 g function

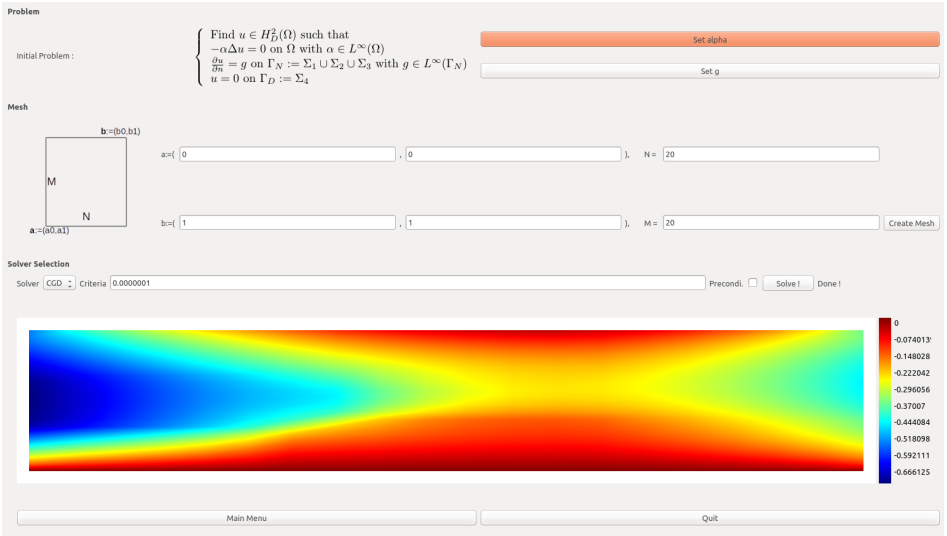

Figure 10.12: Example 2 approximated solution

Chapter 11

Error Analysis

11.1 Considered Problem

We will now analyse the error made by the approximation scheme by taking a reference (not exact) solution computed with a very fine mesh 150×150 and compare it with four other (less fine) meshes. We will first study the error approximation in the given interval $]0,1[\times]0,1[$ and then analyse the convergence rate.
The studied problem is the following :

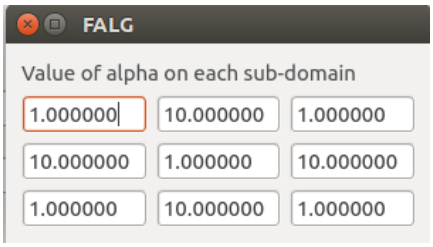

Figure 11.1: α function used

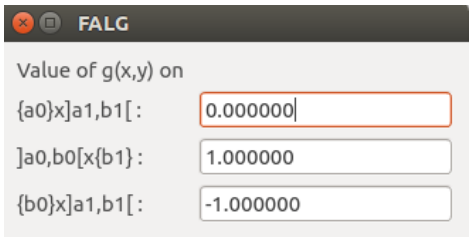

Figure 11.2: g function used

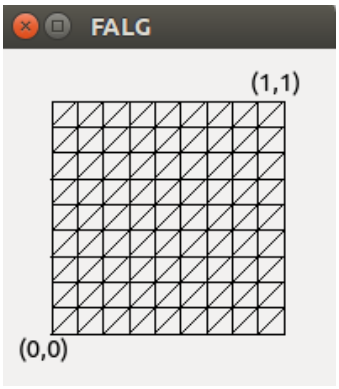

Figure 11.3: mesh used for this example (not the referenced mesh)

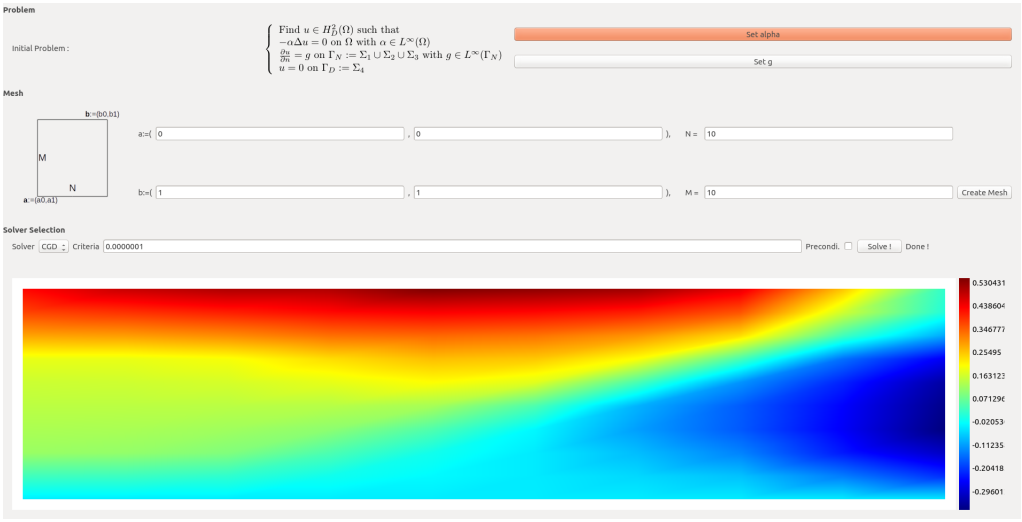

Figure 11.4: Solution example with this mesh

11.2 Analysis

The four considered meshes that will be used against the 150×150 mesh are :

- $9 \times 9 := M_1$
- $12 \times 12 := M_2$
- $24 \times 24 := M_3$
- $36 \times 36 := M_4$

Firstly, we will look at the error made for each mesh with a global colormap in order to emphasize the error reduction when the mesh gets smaller.

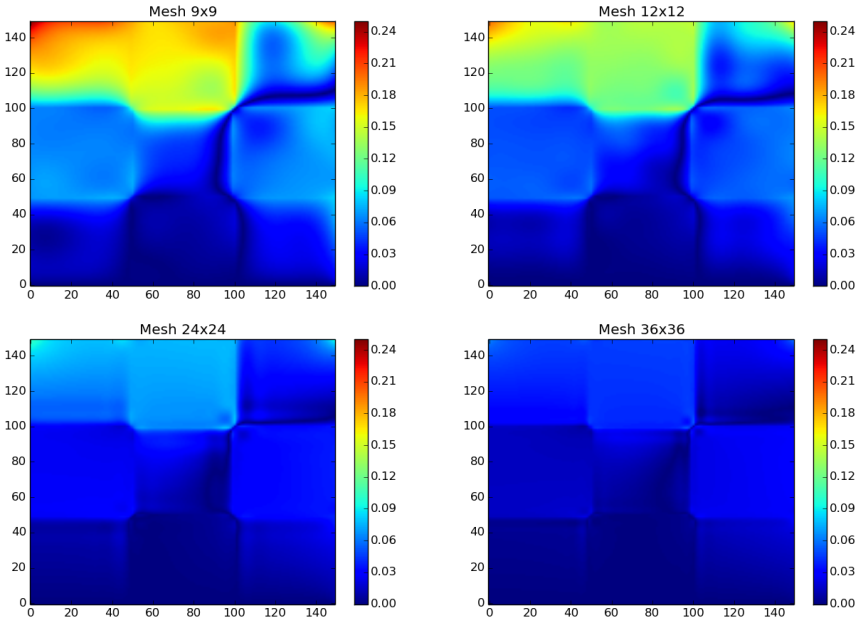

Figure 11.5: Reference 150x150 mesh, with global colormap

In this case, each subplot represents $|M_{ref} - M_i|$ but the key here is that the colors (corresponding to the error made on each mesh point) is using the biggest error of M_1 and the smallest error of M_4 as a range for the colors.
Thus it is clear that the error is reduced by taking a finer mesh.
Note that once again this result has been computed using bilinear interpolation when needed but this is exact by the definition of V_h.
The problem is that we can see the evolution of the error through the different meshes but not really the error repartition for each mesh, thus, here is the same plot but with each subplot having an independent error range and coloration :

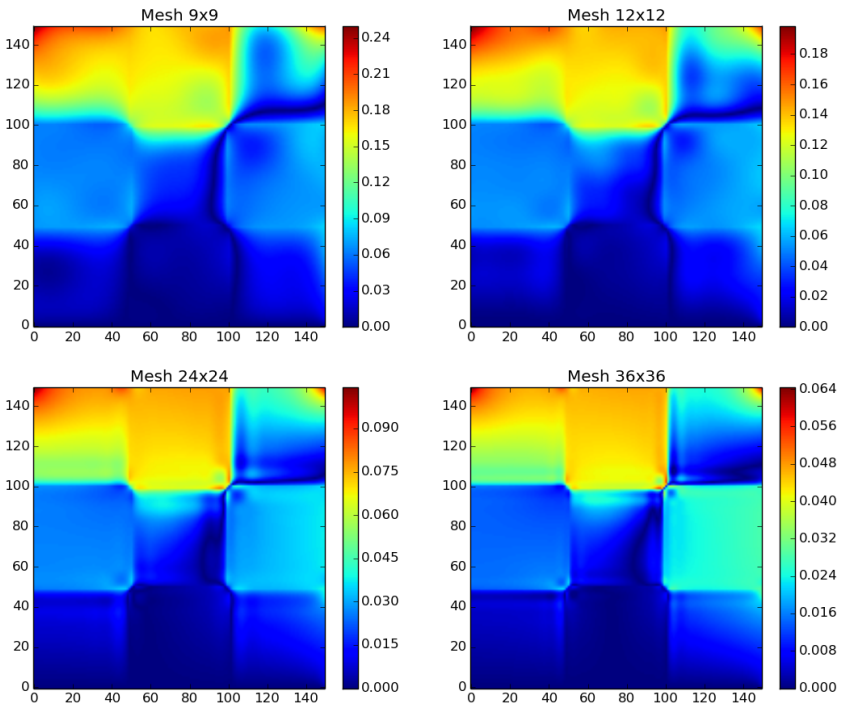

Figure 11.6: Reference 150x150 mesh, with individual colormap

Now we can clearly see that the middle-top sub-domain Ω_i is always problematic whatever the mesh we use. In addition, the top left corner is where the error is the biggest. This analysis could be an indicator that in the sub-domain in question (middle-top) we should use a finer mesh and maybe in the top corners too.
Let's now see the error ($||.||_{H^1}$) made by these approximations compared to the reference one :

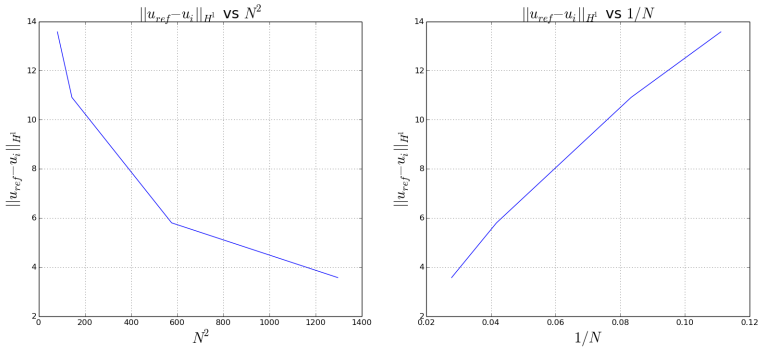

Figure 11.7: Reference 150x150 mesh showing the $O(h)$

When we plot the error versus $1/N$ we clearly see the error which is of order h with h being the mesh steps. This is in fact what was expected when using a P^1 Lagrange basis.

11.3 Conclusion

The presented challenge is really a unique experience for testing algorithms around bioacoustic classification. This task contains all the main difficulties and problems encountered in a real life problem while involving a huge dataset. We have seen that the scattering transform provides a wonderful new data representation but it is not yet easy to know how to extract features from its coefficients. We obviously didn't try every possible algorithm but the idea of thresholding/combination in the λ_2 dimension seems to be interesting. The key part is to be able to capture feature with a time-invariant algorithm in term of the position of the feature in a signal, but, we need to keep a "time" component to analyse frequencies evolution inside the features.

Given the massive dataset it has also been a unique experience to learn ways and methods to deal with computational limitations and the challenge deadline. more importantly, this has helped to deeply understand the scattering network and the pre-processing part in general when trying to apply machine learning algorithms for a given task.