WAVELETS

by

VEGARD AMUNDSEN

THESIS
for the degree of
MASTER OF SCIENCE

(Master in Computational Physics)

Faculty of Mathematics and Natural Sciences
Department of Physics
University of Oslo

January 2010

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Contents

Chapter 1

Introduction

Bla bla... Traditionally only looked at levels, we look at shape. Hypothesis: There exist a parameter describing the shape of the non-stationary EDR signals which is correlated to opplevd stress. The motivation for this thesis is to create a suitable model, parameter and compute its numerical value and test the hypothesis. , choice of model (we model as fBm, more details about fBm given in next chapter)

Chapter 2

Sweat activity

2.1 Measuring sweat activity

General about the gland... (Figure showing a response)

2.2 Our stress measurements

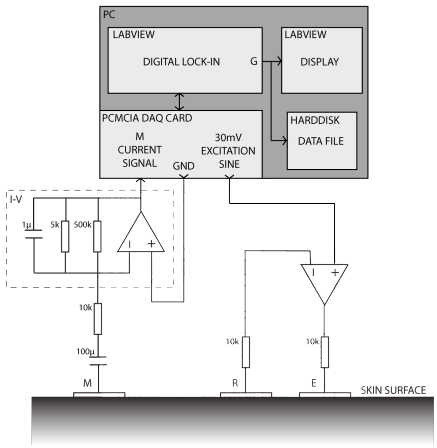

Figure 2.1: Measuring system for skin AC conductance. The system is composed of three main parts: 1.Skin surface electrodes creating the interface between electronic and ionic conductance. 2. Front-end electronics consisting of one dual op-amp IC, 5 resistors and two capacitors built into a small box (60x35x40mm). 3. A PC with a DAQ-card and software for signal processing, display and data storage. A 30mV sine voltage generated by the PC is applied to the skin through the E and R electrodes. Remote sensing at the current-less R electrode causes the right op-amp to drive a current through E which keeps the potential under R equal to the excitation sine, thus eliminating the stratum corneum impedance(SC) below E. Because the impedance of the wet tissue is negligible compared to that of the SC, the only current opposing element that is left is the impedance of the SC below the M electrode. Thus, the measurement is monopolar and confined to the SC of the effective electrode area below M, and the major contributor to changes in this impedance is the filling and reabsorption of sweat in the sweat ducts. The I-V dashed box on the left constitute a current to voltage converter with a Fc=200Hz RC lowpass filter. The 100 capacitor blocks DC potentials generated by the skin and electrodes. The digitized current signal is demodulated by the phase-corrected excitation sine in software, giving the AC conductance output to the display and the data file. For the electrodermal activity (EDA) measurements in this study, the M electrode was placed on the hypothenar area of the palm, while the R and E electrodes were placed on the underarm, but the placement of these two are not critical.

Chapter 3

Stocastic processes and synthetic time series

Let us now make a giant leap from the real world to the world of theory. In order to be able to parameterize our measurements as outlined, we have to have an understanding of synthetic "measurements", realized by stochastic processes. The most fundamental class of such processes are the so-called stationary processes. We will review some basics about stationary processes and their spectral density function (SDF). But as mentioned in the introduction, these processes do not suite our task well as a model for SA. Therefore we have to take a look at how we can develop non-stationary processes from a stationary processes. Finally we will briefly look at some popular synthetic time series and discuss how to make a parameterization of them.

3.1 Stationary stochastic processes

A process is said to be stationary if its statistical features do not change over time. Different segments of such processes are more or less identical. To be more precise, we can define the real valued, discrete parameter stochastic process $\{X_t : t = ..., -1, 0, 1, ...\}$ as a sequence of random variables. In order for $\{X_t\}$ to be stationary, it must satisfy the following properties

$$E\{X_t\} = \mu_X \forall t$$

$$cov\{X_t, X_{t+\tau}\} = s_{X,\tau} \forall t, \tau$$

The first property tells us that the expected value of the process equals a finite constant μ_X at any point t. The second property tells us that the covariance between any two components of $\{X_t\}$ equals a finite constant, $s_{X,\tau}$, that only depends on the separation between the components, τ. The sequence of these constants $\{s_{X,\tau} : \tau = ..., -1, 0, 1, ...\}$ is called the auto covariance sequence (ACVS). The second property has two important implications of the elements in the ACVS

$$s_{X,0} = cov\{X_t, X_t\} = E\{(X_t - \mu_X)^2\} = var\{X_t\}$$

and

$$s_{X,-\tau} = s_{X,\tau}$$

The ACVS is very useful to know because we can define stochastic processes entirely by its ACVS. But for our purpose it is more convenient to consider the Fourier transform of the ACVS. If the ACVS is square summable, we have

$$S_X(f) = \sum_{\infty}^{\infty} s_{X,\tau} e^{-i2\pi f\tau} \quad , -\frac{1}{2} \leq f \leq \frac{1}{2} \tag{3.1}$$

This is called the spectral density function (SDF) of $\{X_t\}$. From our knowledge of the ACVS, we can state two important facts about the SDF:

$$S_X(-f) = S_X(f) \tag{3.2}$$

and

$$\int_{-1/2}^{1/2} S_X(f)\, df = s_{X,0} = var\{X_t\} \tag{3.3}$$

To summarize these two properties in a few words, we claim that the SDF is an even function that decomposes the process variance with respect to frequency. Another important property of the SDF is that the SDFs of two stationary processes $\{X_t\}$ and $\{Y_t\}$, $\{Y_t\}$ created by linearly filtering $\{X_t\}$, are related by the following formula

$$S_Y(f) = |T(f)|^2 S_X(f) \tag{3.4}$$

where $T(f)$ is the filter's transfer function. Because the integral of the SDF is the process variance, we must also have

$$var\{Y_t\} = \int_{-1/2}^{1/2} S_Y(f)\, df = \int_{-1/2}^{1/2} |T(f)|^2 S_X(f)\, df \tag{3.5}$$

The SDF and its properties are crucial for the theory reviewed in the remaining of this chapter and chapter 5.

3.2 Non-stationary processes

In the previous section we looked at stationary processes and an important function, the SDF, characterizing them. As we know, the nature of the SA signals is non-stationary, so it is of interest to define a class of non-stationary processes which can be used as a model for the SA signals. We will now look at a way to ensure a well-defined SDF for a class of non-stationary processes, the so-called $1/f$-type processes.

Let us consider the stochastic process $\{X_t\}$ whose dth order backward difference

$$Y_t \equiv (1-B)^d X_t = \sum_{k=0}^{d} \binom{d}{k} (-1)^k X_{t-k} \quad , d \in \mathbb{N}_0 \tag{3.6}$$

is a stationary process with SDF S_Y and mean μ_Y. The backward shift operator B is defined as

$$B^k X_t = X_{t-k} \tag{3.7}$$

If $\{X_t\}$ is stationary, the SDFs are related through equation (3.4). If $\{X_t\}$ is not stationary, however, we can define

$$S_Y(f) = \left|D(f)^d\right|^2 S_X(f) \tag{3.8}$$

where $D(f)$ is the transfer function of the backward difference filter. This definition ensures a well defined SDF for the non-stationary process because (3.6) tells us that the stationary process, having a well-defined SDF, can be constructed by backward shifts of the non-stationary process. For the special case $d = 1$, we have

$$|D(f)|^2 = 4\sin^2(\pi f)$$

3.3 Long memory processes

We will now define the concept of having memory built into a process and redefine the hypothesis given in the introduction according to this. A process is said to hold long range dependency, or memory, if it satisfies

$$\lim_{f \to 0} \frac{S_X(f)}{C|f|^\alpha} = 1 \tag{3.9}$$

where $C > 0$ and $\alpha < 0$ is constant. We can assign a parameter like α to any long memory process (LMP). The SDF for a LMP must be a function of this parameter, so this parameter, which we will call the LMP parameter, is related to the appearance of the LMP itself. In the view of our SA measurements, we can imagine a model where a provocation forces the signal to be a LMP. Then the signal will "remember" this provocation and this will affect the appearance of the measured signal until it is completely relaxed. We can then assume that the LMP parameter is constant for a relatively short period of time[1]. Our hypothesis is then

The LMP parameter inherent in the EDR-measurements of SA at the hypothenar is correlated to the experienced stress.

3.4 Synthetic time series

As we have redefined our hypothesis, we are now on the hunt for a model of a non-stationary LMP. Luckily, we are able to construct non-stationary stochastic processes with well-defined SDFs through stationary stochastic processes, so a suitable model is in sight. In this section we will look at some widely used stochastic processes and choose a suitable processes as a model for our SA-measurements. Let us start out by examining some popular stationary LMPs studied in the literature.

[1] Relatively short compared to the time it takes to completely relax the system.

3.4.1 Stationary models

Pure power law processes
The discrete-parameter, stationary stochastic process $\{X_t\}$ is said to be a pure power law (PPL) process if its SDF has the form

$$S_X(f) = C|f|^{\alpha} \quad , -\frac{1}{2} \leq f \leq \frac{1}{2} \tag{3.10}$$

where $C > 0$ and $-1 < \alpha < 0$. This is perhaps the simplest model of a LMP because of its close relationship to the definition of LMP given in equation (3.9). As a consequence of this definition, a PPL is not a LMP for $\alpha \geq 0$. The SDF of the PPL process is plotted in figure ???.

Fractional Gaussian noise
The historically oldest model of an LMP is the fractional Gaussian noise (FGN). Although aging, it is a widely used model for physiological signals (Eke et al.(2002)). It is typically defined through its ACVS. The discrete, stationary process $\{X_t\}$ is a FGN if its ACVS is

$$s_{X,\tau} = \frac{\sigma_X^2}{2}\left(|\tau+1|^{2H} - 2|\tau|^{2H} + |\tau-1|^{2H}\right) \quad , \tau = ..., -1, 0, 1, ... \tag{3.11}$$

where $\sigma_X^2 = var\{X_t\}$, and H is the Hurst exponent satisfying $0 < H < 1$. The SDF of FGN is a bit more complicated and non-intuitive. It is given as

$$S_X(f) = \sigma_X^2 C_H 4\sin(\pi f) \sum_{j=-\infty}^{\infty} \frac{1}{|f+j|^{2H+1}} \quad , -\frac{1}{2} \leq f \leq \frac{1}{2} \tag{3.12}$$

where $C_H = \Gamma(2H+1)\sin(\pi f)/(2\pi)^{2H+1}$. For small f we have $S_X(f) \propto |f|^{1-2H}$. From equation (3.9) we must require $1 - 2H < 0$ in order for the FGN to be an LMP. This implies that FGN is an LMP when $0.5 < H < 1$. In this domain, low frequency components are becoming increasingly dominant as H increases. For $0 < H < 0.5$ it is the other way around. In this domain high frequency components become increasingly dominant as H decreases. In the special case $H = 0.5$, equation (3.11) reduces to

$$s_{X,\tau} = \begin{cases} \sigma_X^2, & \tau = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

This often referred to as a white noise process, $\{\epsilon_t\}$.

Fractionally differenced processes
This white noise process is the mother of another popular time series model, the fractionally differenced (FD) process. We can define a discrete FD process $\{X_t\}$ as

$$(1-B)^{\delta} X_t = \sum_{k=0}^{\infty} \binom{\delta}{k} (-1)^k X_{t-k} = \epsilon_t \tag{3.14}$$

where B is the backward shift operator of equation (3.7). Because an FD with LMP parameter $\delta = 0$ is equivalent to an FGN with LMP $H = 0.5$, we must require $0 < \delta < 0.5$ to ensure that the FD process is stationary and an LMP. The SDF of such process can be obtained by applying equation (3.8) to the SDF of the white noise process. If we denote σ_ϵ^2 as the variance of the white noise process, the SDF of an FD process has the form

$$S_X(f) = \frac{\sigma_\epsilon^2}{\left[4\sin^2(\pi f)\right]^\delta} \quad , -\frac{1}{2} \leq f \leq \frac{1}{2} \tag{3.15}$$

We have now looked at three popular models for stationary LMPs. They are similar in the sense that they depend on two parameters only, the variance of the process and the LMP parameter. For us this exponent is of particular interest. The following figure will give us an idea how to extract the LMP parameter from the SDF. As we can see, the SDF of a PPL process is completely linear in a log-log plot, and we can easily extract the LMP parameter by the slope of its SDF. The FGN and FD processes have an approximately linear SDF below about $f = 0.2$ and $f = 0.1$ respectively. By skipping the highest frequencies, we can extract the LMP parameters by the slope of this segment of the SDF.

3.4.2 Non-stationary models

We can extend the theory of synthetic time series to include non-stationary LMPs as well. Let us generalize the theory reviewed above.

Pure power law processes (PPL)
From the definition of a PPL process given in equation (3.10), there is nothing restricting us from allowing other values of α than the ones ensuring the PPL process to be stationary. The constant α is in fact defined over the entire real axis, but because of the definition of an LMP, we must have $\alpha < 0$ in order to have a PPL process with long range memory. Given the range of α defining a stationary process, we can argue that $\alpha \leq -1$ yields a non-stationary PPL process. The non-stationary PPL process will still have a well-defined SDF which is completely linear in a log/log plot.

Fractionally differenced processes (FD)
In a similar manner, we can argue that there is nothing in the definition of a FD process given in equation (3.15) restricting the LMP parameter δ to be in the range $0 < \delta < 0.5$, as we needed to construct a stationary FD process. δ is, like α, defined over the entire real axis. So in order to have a non-stationary FD process with long range memory, it is sufficient to choose $\delta \geq 0.5$.

Fractional Brownian motion (FBM)
In contrast to the LMP parameters of PPL and FD processes, being defined over the entire real axis, the LMP parameter of an FGN process is restricted to a finite interval. So we cannot generalize the FGN to be a non-stationary process. It is stationary by definition! Luckily, we

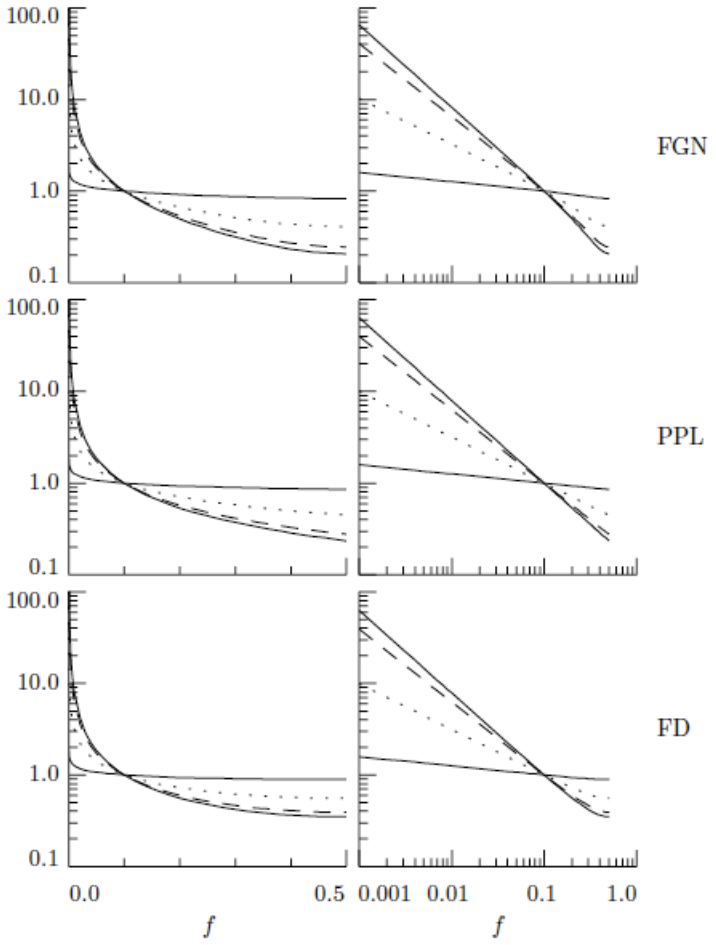

Figure 3.1: SDFs for FGN, PPL and FD processes plotted linearly and as log-log plots (right panels). They are normalized such that $S_X(0.1) = 1$. The SDFs are plotted with LMP parameter values equivalent to $\alpha = -0.1$ (thick solid line), $\alpha = -0.5$ (dotted line), $\alpha = -0.8$ and $\alpha = -0.9$. The relationship between the LMP parameters are given in Figure ???.

are able to construct non-stationary processes from stationary processes by backward difference (3.6). A very popular model of a non-stationary process is constructed by taking the first order backward difference of the FGN process. This is called fractional brownian motion (FBM). The SDF of a discrete, real valued FBM process $\{X_t\}$ is given as

$$S_X(f) = \sigma_X^2 C_H \sum_{j=-\infty}^{\infty} \frac{1}{|f+j|^{2H+1}} \quad , -\frac{1}{2} \leq f \leq \frac{1}{2} \tag{3.16}$$

where $C_H = \Gamma(2H+1)\sin(\pi f)/(2\pi)^{2H+1}$ as before. The FBM is a model that has a lot of different applications. It has a natural link to fractal theory, and the fractal dimension D is related to the Hurst exponent of an FBM process as

$$D = 2 - H \tag{3.17}$$

It has been widely used as a model for physiological signals too. Eke et al. (2002) suggest that any non-stationary physiological signal should be modeled as an FBM. Hence the FBM model, with its LMP parameter H, is a natural choice of model for our SA measurements. To test our hypothesis, we need a mathematical tool to extract the Hurst exponent from our measurements. The theory needed to do so will be described in the following two chapters.

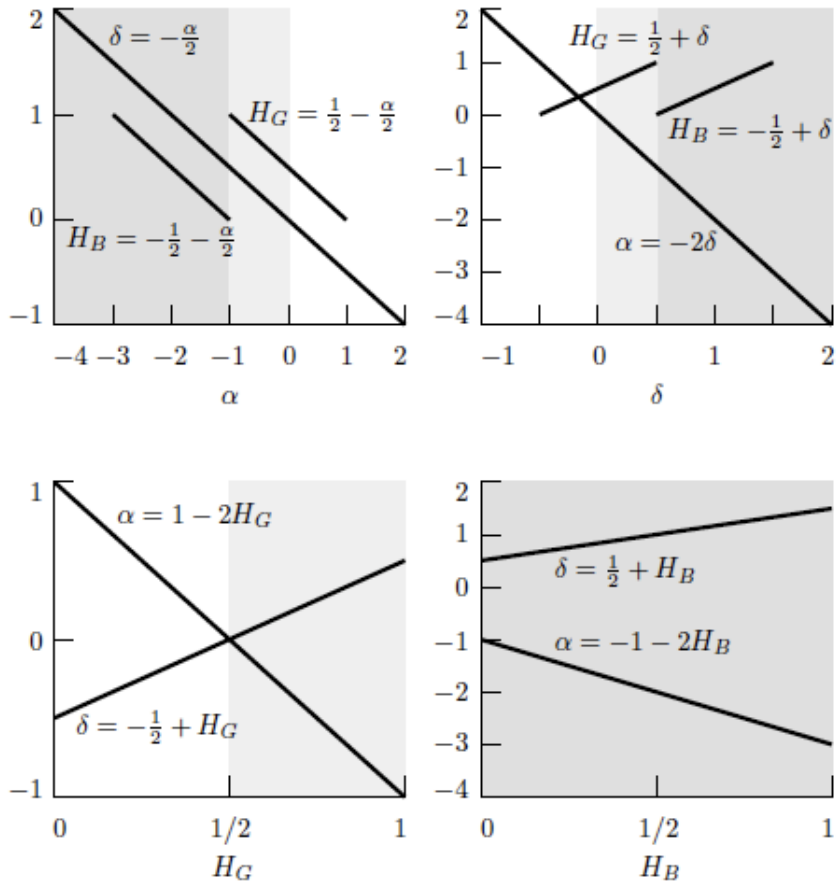

Figure 3.2: Relationships between LMP parameters of the models reviewed here. In order to separate between the Hurst parameters of FGN and FBM, we denote the Hurst exponent of an FGN process as H_G and the Hurst exponent of an FBM process as H_B. The white areas in the figure represent parameter values corresponding to processes without long memory, the lightly shaded areas correspond to processes with "short" memory and the the heavily shaded areas correspond to an LMP.

Chapter 4

The wavelet transform

We constructed a model where a provocation sets a value to the LMP parameter, yielding a characteristic shape of the measured signal for a short period of time. We need to decompose the signal so that we finally can fit a regression line, skipping the highest frequencies. The purpose of this thesis is to assign a parameter to the measured SA-signals. In order to do this, we will need a mathematical tool well suited to analyze non-stationary time series as realized by the SA measurements. I will start out by defining the Fourier transform and the wavelet transform in the continuous representation.

As we have seen, the time series are discrete sampled in time. It is often, however, more convenient to . to look for discrete transform. FT decompose over scale yield a SDF...? then the main objective of this chapter is to find a fast implementation of the discrete transform.
Will stick to 1D throughout
MRA = scale instead of frequency
I will assume real wavelets (although a generalization is no problem (thus innerproduct -> integral form no *)

4.1 The Fourier transform

To extract the frequency information of a signal f in the time domain, a standard technique is to use the Fourier transform (FT):

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{4.1}$$

Where $\hat{f}(\omega)$ is a representation of f in the frequency domain. We can transform f back to the time domain by the inversion

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega t} d\omega \tag{4.2}$$

The FT measures "how much" oscillations are at the frequency ω there is in f, but does not give much information about at which times the frequencies occures. Time-localization can be achieved by first windowing the signal f by a function g, so as to cut off only a well-localized slice of f, and then take its FT:

$$\hat{f}(\omega, t) = \int_{-\infty}^{\infty} f(s)g(s-t)e^{-i\omega s}ds \tag{4.3}$$

This is known as the windowed Fourier transform (WFT). The transform is highly redundant, representing a one dimensional signal in the time-frequency plane (ω, t). It can be interpreted as the "content" of f near time t and near frequency ω. Although the WFT yields a time-frequency decomposition of the signal, it's not necessarily a good tool for an analysis of an arbitrary signal. The WFT decomposes signals over waveforms that have the same time and frequency resolution. If the signal of interest includes structures having different time-frequency resolutions we will need another tool that adress this issue. This leads to the wavelet transform.

4.2 The continuous wavelet transform

To define a transformation that can handle a varying time-frequency resolution, we will need an analyzing function that is localized in time and frequency as opposed to the complex exponentials (or equivalently sine/ cosine) used in FT and WFT. We can obtain this using wavelets.

4.2.1 Wavelets

Let us define a function ψ(t) of the real variable t with properties

$$\int_{-\infty}^{\infty} \psi(t)dt = 0 \tag{4.4}$$

$$\int_{-\infty}^{\infty} \psi^2(t)dt = 1 \tag{4.5}$$

If (4.4) and (4.5) is satisfied, ψ(t) resembles a wave localized in time, a wavelet. A family of wavelets can then be constructed by translating and dilating our mother wavelet

$$\psi_{a,b}(t) = |a|^{-\frac{1}{2}}\,\psi\left(\frac{t-b}{a}\right) \quad , \quad a, b \in \mathbb{R}, a \neq 0 \tag{4.6}$$

where a is the dilation parameter and b is the translation parameter. Let us take a closer look at how typical wavelets may look like.

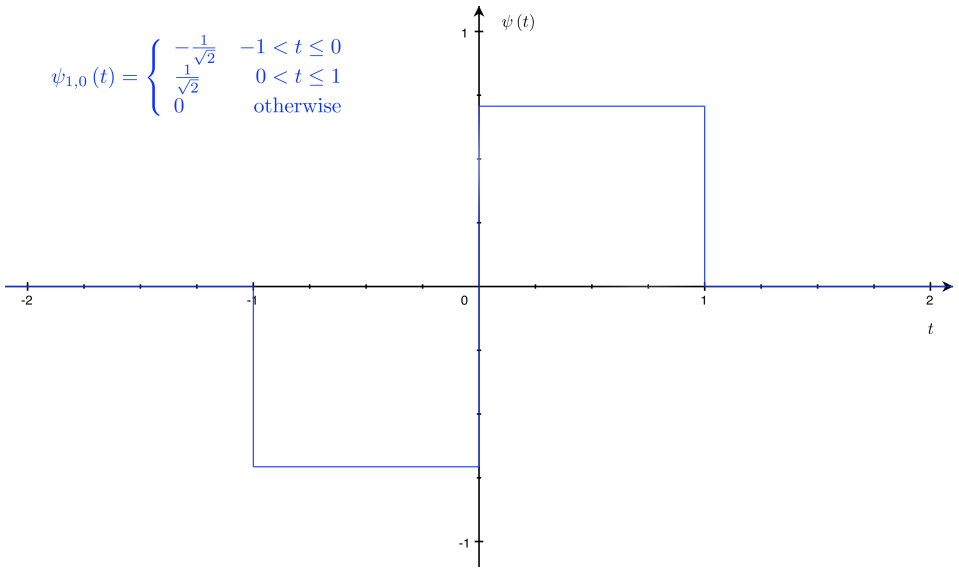

Figure 4.1: The Haar wavelet. This was the first wavelet to be discovered and it had nothing to do with wavelet theory in the first place. Alfred Haar used this function to create a orthonormal system in $L^2(\mathbb{R})$. It was "rediscovered" when the wavelet theory was developed in the 1980s. As we will see later, this is the simplest wavelet of a class of wavelets discovered by Daubechies.

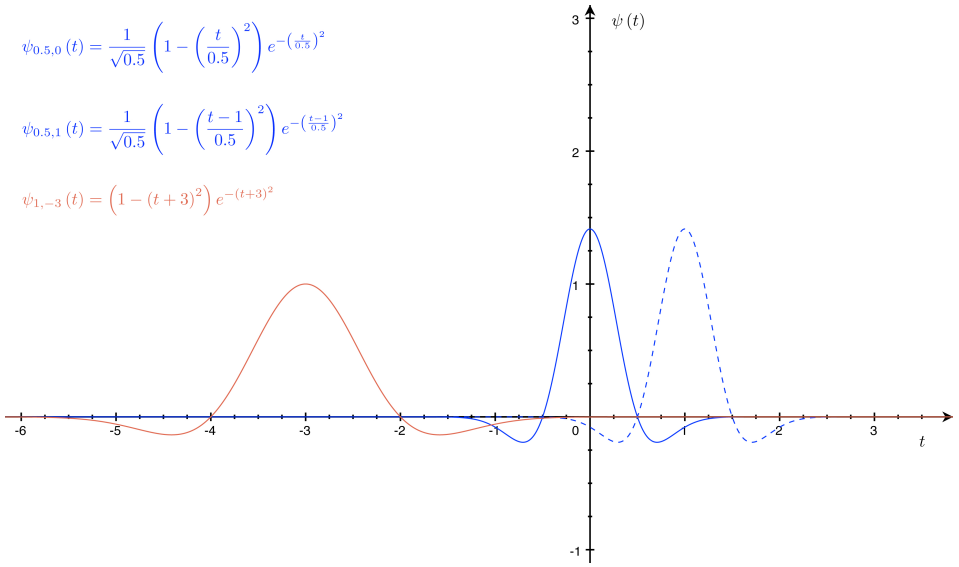

Figure 4.2: Mexican Hat wavelets. The second derivative of a Gaussian. I have plotted...Nanana

4.2.2 CWT

Now that we have an idea of what a wavelet is, we can finally define the continuous wavelet transform (CWT). The transform analogous to (4.3) using wavelets is

$$W(a,b) = \int_{-\infty}^{\infty} f(t)\psi_{a,b}(t)dt \tag{4.7}$$

Which indeed is a highly redundant transform like the WFT. Basically the only thing we have done is to substitute the analyzing functions $g_{\omega,t}(s) = g(s-t)e^{-i\omega s}$ with $\psi_{a,b}(t)$. By doing so we achieve what we were looking for. With the functions $g_{\omega,t}$ we were stuck with the envelope function g, which regardless of ω had the same width. $\psi_{a,b}$, on the other hand, are able to adapt its width to the frequency in the signal. High frequency $\psi_{a,b}(t)$ are very narrow, while low frequency $\psi_{a,b}(t)$ are much broader.Thus the wavelet transform is better able to "zoom in" on high frequency phenomena than theWFT.

Equation (4.7) is the analysis of the signal f(t), and our motivation for constructing the family of wavelets $\psi_{a,b}$. A synthesis of the signal is possible and given by the formula

$$f(t) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} W(a,b)\psi_{a,b}(t)\frac{dadb}{a^2} \tag{4.8}$$

Here we require a further restriction of our wavelets in order to allow reconstruction of f, the so-called admissibility condition. A wavelet $\psi(t)$ is said to be admissible if its FT is such that

$$C_\psi \equiv \int_0^{\infty} \frac{\left|\hat{\psi}(t)\right|^2}{|t|} < \infty \tag{4.9}$$

4.3 Discretization of the wavelet transform

In order to obtain an efficient numerical implementation of the wavelet transform, we have to derive a discrete version of it. Let us consider a discretization of the CWT in the (a,b) plane introducing discrete wavelets.

4.3.1 Discrete wavelets

We can define a family of discrete wavelets by allowing the dilation parameter a and the translation parameter b both to take discrete values only. If we choose $a = a_0^j$ and $b = kb_0a_0^j$, where $b_0 > 0$ and $j, k \in \mathbb{Z}$, we ensure that the discretized wavelets $\psi_{j,k}$ "cover" t in the same way for all j. The family of discrete wavelets is then

$$\psi_{j,k}(t) = a_0^{-\frac{j}{2}}\psi(a_0^{-j}t - kb_0) \tag{4.10}$$

The analysis of a signal f is given by

$$W(j,k) = \int_{-\infty}^{\infty} f(t)\psi_{j,k}(t)dt \tag{4.11}$$

This description yields yet again a redundant transform. However, the discrete wavelets defined in (4.10) does not in general give rise to a synthesis equation like (4.8) even if the admissibility condition (4.9) is satisfied in its discrete form. A solution to this problem can be approached in two ways:

- Construct wavelets $\psi_{j,k}$ who constitute a frame [1]
- Construct an orthonormal wavelet basis with compact support

The first approach is carefully studied in (daub and mallat). If we follow this path, we will typically end up choosing wavelets who are well localized in time and frequency, but defined over the entire real axis. To implement such a transform, we will have to make a cut-off of the wavelets at some point. It would be more convenient to implement a transform whose wavelets have compact support. This can be achieved by constructing an orthonormal wavelet basis. Our first step towards a numerical implementation of the wavelet transform is thus to create such basis. To get there, we will have to define the concept of a multiresolution approximation.

4.3.2 Multiresolution approximation

Let us consider a function space decomposed into a ladder of subspaces where each subspace is a scaled version of the central space. If we look at our signal f this way, it implies that there is a piece of f in each subspace. An orthogonal projection of f on subspace V_j will give an approximation f_j of f at resolution a_0^{-j}. If we let $\{W_j\}$ be another family of subspaces containing the differences between V_{j-1} and V_j for all j, then f can be approximated at an arbitrary precision by $\{V_j\}$ and $\{W_j\}$. Let us give these ideas a more mathematical framework. For simplicity, we choose $a_0 = 2$ and $b_0 = 1$, which means that (4.10) takes the form

$$\psi_{j,k}(t) = 2^{-\frac{j}{2}}\psi(2^{-j}t - k) \tag{4.12}$$

A multiresolution approximation is then a sequence of closed subspaces $\{V_j\}$ satisfying the following properties

$$V_j \subset V_{j-1} \quad ,\forall j \in \mathbb{Z} \tag{4.13}$$

$$\overline{\bigcup_{j\in\mathbb{Z}} V_j} = L^2(\mathbb{R}) \tag{4.14}$$

$$\bigcap_{j\in\mathbb{Z}} V_j = \{0\} \tag{4.15}$$

$$f \in V_j \Longleftrightarrow f(2^j\cdot) \in V_0 \quad ,\forall j \in \mathbb{Z} \tag{4.16}$$

[1] A frame of a vector space can be seen as a generalization of the idea of a basis to sets which may be linearly dependent

$$f \in V_0 \Rightarrow f(\cdot - k) \in V_0 \quad , \forall k \in \mathbb{Z} \tag{4.17}$$

$$\exists \phi \in V_0 : \{\phi_{0,k}; k \in \mathbb{Z}\} \text{ is an orthonormal basis} \tag{4.18}$$

To summarize these equations in a few words, equations (4.13), (4.14), (4.15) ensures completeness of $\{V_j\}$ in $L^2(\mathbb{R})$, equation (4.16) require V_j to be scale invariant and equation (4.17) require V_j to be shift-invariant. Equation (4.18) combined with (4.16) tells us that there for all $j, k \in \mathbb{Z}$ exist an orthonormal basis $\{\phi_{j,k}\}$ where

$$\phi_{j,k}(t) = 2^{-\frac{j}{2}} \phi(2^{-j}t - k) \tag{4.19}$$

We will call V_j the scaling space, hence $\phi_{j,k}$ is called the scaling function of the multiresolution approximation. Let W_j be the wavelet space with an orthonormal wavelet basis $\{\psi_{j,k}; j, k \in \mathbb{Z}\}$. For all $j \in \mathbb{Z}$ we define

$$V_{j-1} = V_j \bigoplus W_j \tag{4.20}$$

with the constraint that $W_j \perp W_{j'}$ if $j \neq j'$. We see that W_j contain the difference between the approximations with resolution 2^{j-1} and 2^j. We will call this the detail at level j. Using equation (4.14) we can "telescope" the union to write

$$L^2(\mathbb{R}) = \bigoplus_{j \in \mathbb{Z}} W_j \tag{4.21}$$

In other words, the wavelet subspaces yields a wavelet decomposition of $L^2(\mathbb{R})$! Equations (4.14), (4.15) and (4.21) then implies that $\{\psi_{j,k}; j, k \in \mathbb{Z}\}$ is a basis for $L^2(\mathbb{R})$. As a consequence of (4.20), W_j inherit the scaling property (4.16) from V_j. Our search for an orthonormal wavelet basis can then be reduced to find $\psi \in W_0$ such that $\psi(\cdot - k)$ constitute an orthonormal basis for W_0.

4.3.3 The scaling function

Husk dette med averages/differences og hvilket nivaa naa!!! Before we can construct ψ, we need to study the scaling function ϕ in more detail. With $\phi \in V_0 \subset V_{-1}$ and $\{\phi_{-1,k}; k \in \mathbb{Z}\}$ an orthonormal basis in V_{-1}, we can write

$$\phi(t) = \sum_{k=-\infty}^{\infty} \langle \phi_{0,k} | \phi_{-1,k} \rangle \phi_{-1,k}(t) \equiv \sum_{k=-\infty}^{\infty} g_k \phi_{-1,k}(t) \tag{4.22}$$

Where $\{g_k\}$ is a set of scaling coeffisients. Since $\phi_{-1,k}(t) = \sqrt{2}\phi(2t - k)$, equation (4.22) can be rewritten as

$$\phi(t) = \sqrt{2} \sum_{k=-\infty}^{\infty} g_k \phi(2t - k) \tag{4.23}$$

To ensure that $\phi(t)$ is normalized, we must have

$$1 = \int_{-\infty}^{\infty} \phi(t) dt = \sqrt{2} \sum_{k=-\infty}^{\infty} g_k \int_{-\infty}^{\infty} \phi(2t - k) dt$$

The substitution $t' = 2t - k$ results in the requirement

$$\sum_{k=-\infty}^{\infty} g_k = \sqrt{2} \tag{4.24}$$

Equation (4.23) is a so-called two-scale difference equation.Taking its Fourier transform, the two-scale difference equation takes the form

$$\hat{\phi}(\xi) = \frac{1}{\sqrt{2}} \sum_{k=-\infty}^{\infty} g_k e^{-ik\xi/2} \hat{\phi}\left(\frac{\xi}{2}\right) \equiv G\left(\frac{\xi}{2}\right) \hat{\phi}\left(\frac{\xi}{2}\right) \tag{4.25}$$

where

$$G(\xi) = \frac{1}{\sqrt{2}} \sum_{k=-\infty}^{\infty} g_k e^{-ik\xi} \tag{4.26}$$

is the transfer function of g. We can place a constraint on the form of $G(\xi)$, apart from being periodic in $L^2([0, 2\pi])$,by taking advantage of the orthonormality of $\phi(\cdot - k)$

$$\delta_{k,0} = \int_{-\infty}^{\infty} \phi(t)\phi^*(t-k)\, dt = \int_{-\infty}^{\infty} \left|\hat{\phi}(\xi)\right|^2 e^{ik\xi} d\xi = \int_0^{2\pi} \left(\sum_{l=-\infty}^{\infty} \left|\hat{\phi}(\xi + 2\pi l)\right|^2 \right) e^{ik\xi} d\xi$$

In the second step, we applied Parseval's theorem. It can be reviewed in for example (percival/walden and mallat). The last expression is nothing but the inverse FT of the sum in the parenthesis. This implies

$$\sum_{l=-\infty}^{\infty} \left|\hat{\phi}(\xi + 2\pi l)\right|^2 = \frac{1}{2\pi} \tag{4.27}$$

With reference to (4.25) and with the substitution of variables $\zeta = \xi/2$, equation (4.27) can be rewritten as

$$\sum_{l=-\infty}^{\infty} |G(\zeta + \pi l)|^2 \left|\hat{\phi}(\zeta + \pi l)\right|^2 = \frac{1}{2\pi}$$

Because of the periodicity of G, we can split the sum in this expression into even and odd l which gives us

$$|G(\zeta + \pi)|^2 \sum_{l=-\infty}^{\infty} \left|\hat{\phi}(\zeta + 2(l+1)\pi)\right|^2 + |G(\zeta)|^2 \sum_{l=-\infty}^{\infty} \left|\hat{\phi}(\zeta + 2\pi l)\right|^2 = \frac{1}{2\pi}$$

By equation (4.27) this can be simplified to

$$|G(\zeta + \pi)|^2 + |G(\zeta)|^2 = 1 \tag{4.28}$$

4.3.4 The wavelet function

Now that we have derived some properties for the scaling function, let us try to describe f in the wavelet space in order to find a counterpart to the scaling function, namely the wavelet

function ψ, as a canditate for being an orthonormal basis in W_0. Let us write out equation (4.20) for $f \in W_0$.

$$V_{-1} = V_0 \bigoplus W_0$$

This tells us that a description of $f \in W_0$ is equivalent to a description of $f \in V_{-1}$ where $f \perp V_0$. For $f \in V_{-1}$ we can write

$$f = \sum_{k=-\infty}^{\infty} \langle f|\phi_{-1,k}\rangle \phi_{-1,k} \equiv \sum_{k-\infty}^{\infty} c_k \phi_{-1,k} \tag{4.29}$$

where $\{c_k\}$ is a set of constants. Since $\phi_{-1,k}(t) = \sqrt{2}\phi(2t-k)$ still, the Fourier transform of f is given by

$$\hat{f}(\xi) = \frac{1}{\sqrt{2}} \sum_{k=-\infty}^{\infty} c_k e^{-ik\xi/2} \hat{\phi}\left(\frac{\xi}{2}\right) \equiv C\left(\frac{\xi}{2}\right) \hat{\phi}\left(\frac{\xi}{2}\right) \tag{4.30}$$

where

$$C(\xi) = \frac{1}{\sqrt{2}} \sum_{k=-\infty}^{\infty} c_k e^{-ik\xi} \tag{4.31}$$

We can find a relation between $C(\xi)$ and $G(\xi)$ of equation (4.26) by the fact that $f \perp V_0 \Longrightarrow f \perp \phi_{0,k}$ for all k.

$$0 = \int_{-\infty}^{\infty} f(t)\,\phi^*(t)\,dt = \int_{-\infty}^{\infty} \hat{f}(\xi)\,\hat{\phi}^*(\xi)\,e^{ik\xi} d\xi = \int_0^{2\pi} \left(\sum_{l=-\infty}^{\infty} \hat{f}(\xi + 2\pi l)\,\hat{\phi}^*(\xi + 2\pi l) \right) e^{ik\xi} d\xi$$

hence

$$\sum_{l=-\infty}^{\infty} \hat{f}(\xi + 2\pi l)\,\hat{\phi}^*(\xi + 2\pi l) = 0 \tag{4.32}$$

Manipulating this sum in a similar way that we manipulated (4.27) we end up with a constraint on $C(\xi)$ (remember $\zeta = \xi/2$)

$$C(\zeta)\,G^*(\zeta) + C(\zeta + \pi)\,G^*(\zeta + \pi) = 0 \tag{4.33}$$

Because of (4.28), $G^*(\zeta)$ and $G^*(\zeta + \pi)$ cannot vanish togheter. We can then rewrite (4.33) as

$$C(\zeta) = -\frac{C(\zeta + \pi)}{G^*(\zeta)} G^*(\zeta + \pi) \equiv \lambda(\zeta)\,G^*(\zeta + \pi)$$

By inserting $\lambda(\zeta)$ and $\lambda(\zeta + \pi)$ in (4.33), we get

$$\lambda(\zeta) + \lambda(\zeta + \pi) = 0$$

It can be easily verified that

$$\lambda(\zeta) = e^{i\zeta} \nu(2\zeta) \tag{4.34}$$

satisfies this condition for ν being 2π-periodic. We can then write $C(\xi/2) = e^{i\zeta}\nu(2\zeta)G^*(\xi+\pi)$ and rewrite (4.30) as

$$\hat{f}(\xi) = e^{i\xi/2}G^*(\xi/2+\pi)\,\nu(\xi)\,\hat{\phi}(\xi/2) \tag{4.35}$$

Let us now assume that

$$\hat{\psi}(\xi) = e^{i\xi/2}G^*(\xi/2+\pi)\,\hat{\phi}(\xi/2) \tag{4.36}$$

is a Fourier-transformed wavelet. (4.35) can then be written

$$\hat{f}(\xi) = \left(\sum_{k=-\infty}^{\infty}\nu_k e^{-ik\xi}\right)\hat{\psi}(\xi) \Longleftrightarrow f = \sum_{k=-\infty}^{\infty}\nu_k\psi(\cdot-k)$$

where $\psi(\cdot-k)$ is a basis of W_0.[2] Finally we can define the wavelet function as the inverse Fourier-transform of (4.36)

$$\psi(t) = \sqrt{2}\sum_{k=-\infty}^{\infty}h_k\phi(2t-k) \tag{4.37}$$

where

$$h_k = (-1)^{k-1}g_{-k-1} \tag{4.38}$$

From equation (4.4) we required the integral of $\psi(t)$ to be 0. From this condition we can derive a requirement for the set of coefficients $\{h_k\}$

$$0 = \int_{-\infty}^{\infty}\psi(t)\,dt = \sqrt{2}\sum_{k=-\infty}^{\infty}h_k\int_{-\infty}^{\infty}\phi(2t-k)\,dt$$

The substitution of variable $t' = 2t-k$ gives

$$\sum_{k=-\infty}^{\infty}h_k = 0 \tag{4.39}$$

Since the construction of an orthonormal wavelet-basis, $\psi_{0,k}$ for W_0 leads to an orthonormal basis $\{\psi_{j,k}; j,k\in\mathbb{Z}\}$ for $L^2(\mathbb{R})$, we can now define the multiresolution analysis (MRA) of a signal f

$$f = P_j f + \sum_{l=-\infty}^{j}Q_l f \tag{4.40}$$

where P_j is an orthogonal projection of f onto V_j and Q_l is an orthogonal projection of f onto W_l.

4.3.5 Construction of compactly supported wavelets

In this section we will use our knowledge about multiresolution to construct orthonormal wavelet bases with compact support. We will primarly focus on the Daubechies' wavelets, but also mention a few other related wavelet bases.

[2] The details prooving that $\psi(\cdot-k)$ really is an orthonormal basis for W_0 can be found in (daubechies)

To ensure compact support, it is sufficient to make sure that the sequences $\{g_k\}$ and $\{h_k\}$ have a finite number of nonzero coefficients. This follows naturally from the multiresolution approximation where we defined $\{g_k\}$ as $\langle \phi_{0,k} | \phi_{-1,k} \rangle$. This means that the transfer function G from equation (4.26) now is given by

$$G(\xi) = \frac{1}{\sqrt{2}} \sum_{k=0}^{L-1} g_k e^{-ik\xi} \tag{4.41}$$

In order to construct Daubechies' wavelets, we need a stronger condition on our wavelets than (4.4). Let us introduce the condition of vanishing moments

$$\int_{-\infty}^{\infty} t^l \psi(t)\, dt = 0 \quad , \quad l = 0, 1, 2, ..., m-1 \tag{4.42}$$

This puts the restriction on $\psi(t)$ that the first m moments vanish. Let us find out what this implies for $\hat{\psi}(\xi)$

$$\hat{\psi}(\xi) = \int_{-\infty}^{\infty} \psi(t)\, e^{-i\xi t} dt$$

The first derivative with respect to ξ yields

$$\frac{d\hat{\psi}}{d\xi} = \int_{-\infty}^{\infty} \psi(t)\, (-it)\, e^{-i\xi t} dt$$

Continued differentiation gives

$$\frac{d^l \hat{\psi}}{d\xi^l} = (-i)^l \int_{-\infty}^{\infty} t^l \psi(t)\, e^{-i\xi t} dt$$

If we insert $\xi = 0$ in the expression above, we get

$$\frac{d^l \hat{\psi}}{d\xi^l} \mid_{\xi=0} = (-i)^l \int_{-\infty}^{\infty} t^m \psi(t)\, dt$$

Now, by the condition of vanishing moments, we must have

$$\frac{d^l \hat{\psi}}{d\xi^l} \mid_{\xi=0} = 0 \quad , \quad l < m \tag{4.43}$$

We know from (4.36) that $\hat{\psi}(\xi)$ is related to $G(\xi)$. We can therefore show that (4.43) put the same constraint on $G(\xi)$ in $\xi = \pi$

$$\frac{d^l G}{d\xi^l} \mid_{\xi=\pi} = 0 \quad , \quad l < m \tag{4.44}$$

This means that $G(\xi)$ has a multiplicity m of zero in $\xi = \pi$. It is now a good idea to factor out these vanishing moments. G can then be written as

$$G(\xi) = \left(\frac{1 + e^{-i\xi}}{2}\right)^m \mathcal{L}(\xi) \tag{4.45}$$

where $\mathcal{L}(\xi)$ is a trigonometric polynomial. We are now able to develop a trigonometric polynomial for $G(\xi)$ such that equation (4.28) holds. The first step is to take the square of (4.45)

$$|G(\xi)|^2 = \left(\frac{1 + e^{-i\xi}}{2}\right)^{2m} |\mathcal{L}(\xi)|^2$$

Now we can express as in cosine. Obviously G itself is a cosine, so L must also be cosine. But for convenience we will use the identity blabla andr write EQ as

4.3.6 Phase properties of orthonormal wavelets

4.4 The pyramid algorithm

We have so far used the multiresolution approximation to construct families of orthonormal wavelets. As we shall see in this section, the multiresolution approximation also leads to a fast numerical scheme for computing the wavelet transform - the pyramid algorithm[3]. In the following subsections, we will derive the pyramid algorithm for two different choices of the translation parameter b_0 of the discrete wavelets $\psi_{j,k}$.

4.4.1 DWT

The "simplest" choice of b_0 is the one we made while treating the multiresolution approximation, $b_0 = 1$. This leads to discrete wavelets on the form given in (4.12)

$$\psi_{j,k}(t) = 2^{-j/2}\psi(2^{-j}t - k)$$

and a similar expression for the scaling function

$$\phi_{m,n}(t) = 2^{-m/2}\phi(2^{-m}t - n)$$

We also know, from equation (4.37), that the wavelet function can be expressed in terms of the scaling function.

$$\psi(t) = \sqrt{2} \sum_{n=-\infty}^{\infty} \bar{h}_n \phi(2t - n)$$

[3]Other schemes for implementing the wavelet- transform exists. The lifting scheme offers in-place memory operations and is twice as fast as the pyramid algorithm. An excellent introduction to wavelets and lifting is given in (ripples)

Combining these yields

$$\psi_{j,k}(t) = 2^{-j/2}\psi(2^{-j}t - k) = 2^{-(j-1)/2} \sum_{n=-\infty}^{\infty} \bar{h}_n \phi\left(2^{-(j-1)}t - (2k+n)\right) = \sum_{n=-\infty}^{\infty} \bar{h}_n \phi_{j-1,2k+n}$$

Now, taking the inner product with the signal of interest, f, we get the wavelet coefficients of equation (4.11)

$$\langle f, \psi_{j,k} \rangle = \sum_{n=-\infty}^{\infty} \bar{h}_n \langle f, \phi_{j-1,2k+n} \rangle$$

Remember that we defined the wavelet filter as $h_n \equiv \bar{h}_{-n}$. If we substitute this in the expression, we get

$$\langle f, \psi_{j,k} \rangle = \sum_{n=-\infty}^{\infty} h_n \langle f, \phi_{j-1,2k-n} \rangle \tag{4.46}$$

A similar derivation for the scaling function yields

$$\langle f, \phi_{j,k} \rangle = \sum_{n=-\infty}^{\infty} g_n \langle f, \phi_{j-1,2k-n} \rangle \tag{4.47}$$

We can now compute the first set of wavelet coefficients, $\langle f, \psi_{1,k} \rangle$, by convolving the sequence $\langle f, \phi_{0,2k-n} \rangle$ with the wavelet filter h_n. Then we compute the first set of scaling coefficients, $\langle f, \phi_{1,k} \rangle$, by convolving the sequence $\langle f, \phi_{0,2k-n} \rangle$ with the scaling filter g_n. This procedure can be repeated to compute the second set of wavelet and scaling coefficients and so on.

Matrix representation

The above briefly sketches the numerical scheme we are looking for. To get the "full picture" of the DWT, let us change the notation a bit so that we can express each step of the pyramid algorithm with matrices. Let us start out by modifying the wavelet and scaling filters to fit such representation.

$$\begin{bmatrix} h_1 & h_0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & h_3 & h_2 \\ h_3 & h_2 & h_1 & h_0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & h_3 & h_2 & h_1 & h_0 \end{bmatrix}$$

Filter representation

4.4.2 MODWT

Chapter 5

Applications of the wavelet transform on time series

In this chapter we will finally make use of the wavelet coefficients, obtained by the pyramid algorithm as described in the previous chapter, by either the DWT or the MODWT approach. The goal is to determine the LMP-parameter present in the measured signal. To get there, we must explore how the wavelet coefficients decompose variance over scales and how this is related to the SDF.

5.1 Wavelet decomposition of variance

Let us consider a sampled signal $\{X_t\}$ of length $N = 2^J$, with sample mean $\overline{X}$, sample variance σ_X^2 and energy $||\mathbf{X}||^2$. By performing a partial DWT to such a signal up to level $J_0 < J$, we obtain the energy decomposition

$$||\mathbf{X}||^2 = \sum_{j=1}^{J_0} ||\mathbf{W}_j||^2 + ||\mathbf{V}_{J_0}||^2 \tag{5.1}$$

Then it follows naturally from the definition of the sample variance, $\sigma_X = \frac{1}{N} ||\mathbf{X}||^2 - \overline{X}^2$, that the DWT decomposes the sample variance as

$$\sigma_X^2 = \frac{1}{N} \sum_{j=1}^{J_0} ||\mathbf{W}_j||^2 + \frac{1}{N} ||\mathbf{V}_{J_0}||^2 - \overline{X}^2 \tag{5.2}$$

If we now specialize to the case of a full DWT, that is a DWT up to level J, we can use the result from equation ??? ,which implies that $||\mathbf{V}_{J_0}||^2 = N\overline{X}^2$, and insert in the equation above. We can then decompose the sample variance as

$$\sigma_X^2 = \frac{1}{N} \sum_{j=1}^{J} ||\mathbf{W}_j||^2 \tag{5.3}$$

This can be interpreted as a wavelet decomposition of the sample variance, where $\frac{1}{N}||\mathbf{W}_j||^2$ represents the contribution to the sample variance of $\{X_t\}$ due to changes at scale τ_j.

Let us now review the decomposition of variance as a consequence of an MODWT. The energy decomposition (5.1) followed naturally for the DWT, the DWT being an orthonormal transform and hence isometric. In order to get a simular decomposition of the sample variance for the MODWT, we need to show that this transform is isometric too. Consider equation ??? and ???

$$\widetilde{W}_{j,t} = \sum_{l=0}^{N-1} \tilde{h}^o_{j,l} X_{t-l \bmod N} \quad , \quad \widetilde{V}_{j,t} = \sum_{l=0}^{N-1} \tilde{g}^o_{j,l} X_{t-l \bmod N} \quad , t = 0, 1, 2, ..., N-1$$

Taking the FT of the square of these expressions applying Parseval's theorem, we get

$$\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 = \frac{1}{N}\sum_{k=0}^{N-1}\left|\widetilde{H}_j\left(\frac{k}{N}\right)\right|^2\left|\hat{X}_k\right|^2 \quad , \quad \left|\left|\widetilde{\mathbf{V}}_j\right|\right|^2 = \frac{1}{N}\sum_{k=0}^{N-1}\left|\widetilde{G}_j\left(\frac{k}{N}\right)\right|^2\left|\hat{X}_k\right|^2$$

Adding the energy of the wavelet coefficients and the scaling coefficients we get

$$\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 + \left|\left|\widetilde{\mathbf{V}}_j\right|\right|^2 = \frac{1}{N}\sum_{k=0}^{N-1}\left|\hat{X}_k\right|^2\left(\left|\widetilde{H}_j\left(\frac{k}{N}\right)\right|^2 + \left|\widetilde{G}_j\left(\frac{k}{N}\right)\right|^2\right)$$

By the definitions of the transfer functions, ??? and ???, and equation ??? the paranthesis simplifies to

$$\left|\widetilde{H}_j\left(\frac{k}{N}\right)\right|^2 + \left|\widetilde{G}_j\left(\frac{k}{N}\right)\right|^2 = \left|\widetilde{H}\left(2^{j-1}\frac{k}{N}\right)\right|^2\prod_{l=0}^{j-2}\left|\widetilde{G}\left(2^l\frac{k}{N}\right)\right|^2 + \prod_{l=0}^{j-1}\left|\widetilde{G}\left(2^l\frac{k}{N}\right)\right|^2$$

$$= \left(\left|\widetilde{H}\left(2^{j-1}\frac{k}{N}\right)\right|^2 + \left|\widetilde{G}\left(2^{j-1}\frac{k}{N}\right)\right|^2\right)\prod_{l=0}^{j-2}\left|\widetilde{G}\left(2^l\frac{k}{N}\right)\right|^2$$

$$= \left(\left|\widetilde{G}\left(2^{j-1}\frac{k}{N}+\frac{1}{2}\right)\right|^2 + \left|\widetilde{G}\left(2^{j-1}\frac{k}{N}\right)\right|^2\right)\left|\widetilde{G}_{j-1}\left(\frac{k}{N}\right)\right|^2 = \left|\widetilde{G}_{j-1}\left(\frac{k}{N}\right)\right|^2$$

for any $j \geq 2$. We now have

$$\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 + \left|\left|\widetilde{\mathbf{V}}_j\right|\right|^2 = \frac{1}{N}\sum_{k=0}^{N-1}\left|\widetilde{G}_{j-1}\left(\frac{k}{N}\right)\right|^2\left|\hat{X}_k\right|^2 = \left|\left|\widetilde{\mathbf{V}}_{j-1}\right|\right|^2$$

which can be written

$$\left|\left|\widetilde{\mathbf{V}}_1\right|\right|^2 = \sum_{j=2}^{J_0}\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 + \left|\left|\widetilde{\mathbf{V}}_j\right|\right|^2$$

for any $J_0 \geq 2$. We now need to have $||\mathbf{X}||^2 = \left|\left|\widetilde{\mathbf{W}}_1\right|\right|^2 \left|\left|\widetilde{\mathbf{V}}_1\right|\right|^2$ for MODWT to be isometric. Given the derivation above, this is rather straight forward to show

$$\left|\left|\widetilde{\mathbf{W}}_1\right|\right|^2 + \left|\left|\widetilde{\mathbf{V}}_1\right|\right|^2 = \frac{1}{N}\sum_{k=0}^{N-1}\left|\widetilde{H}_j\left(\frac{k}{N}\right)\right|^2\left|\hat{X}_k\right|^2 + \frac{1}{N}\sum_{k=0}^{N-1}\left|\widetilde{G}_j\left(\frac{k}{N}\right)\right|^2\left|\hat{X}_k\right|^2$$

$$= \frac{1}{N}\sum_{k=0}^{N-1}\left|\hat{X}_k\right|^2\left(\left|\widetilde{H}\left(2^{j-1}\frac{k}{N}\right)\right|^2 + \left|\widetilde{G}\left(2^{j-1}\frac{k}{N}\right)\right|^2\right) = \frac{1}{N}\sum_{k=0}^{N-1}\left|\hat{X}_k\right|^2 = ||\mathbf{X}||^2$$

Hence the energy is conserved by the MODWT coefficients, and we can write the sample variance decomposition of the signal $\{X_t\}$ as

$$\sigma_X^2 = \frac{1}{N}\sum_{j=1}^{J_0}\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 + \frac{1}{N}\left|\left|\widetilde{\mathbf{V}}_{J_0}\right|\right|^2 - \overline{X}^2 \tag{5.4}$$

If we now look at the special case where $J_0 = J$, we can again use the result of equation ???.hilewfiojd!!!!

$$\sigma_X^2 = \frac{1}{N}\sum_{j=1}^{J}\left|\left|\widetilde{\mathbf{W}}_j\right|\right|^2 \tag{5.5}$$

We have now looked at how the wavelet coefficients decompose the sample variance of a time series with respect to scales, τ_j. Let us now define the concept of wavelet variance using the notation of MODWT. Let $\{X_t : t = ..., -1, 0, 1, ...\}$ be a realization of an infinite discrete parameter real-valued stochastic process. Transforming this signal with the jth level MODWT filter, $\tilde{h}_{j,l}$ with length L_j, gives a finite set of wavelet coefficients

$$\overline{W}_{j,t} \equiv \sum_{l=0}^{L_j-1}\tilde{h}_{j,l}X_{t-l} \quad , t = ..., -1, 0, 1, ... \tag{5.6}$$

The time-independent wavelet variance at scale τ_j is then defined as the variance of a set of these coefficients

$$\nu_X^2(\tau_j) \equiv var\{\overline{W}_{j,t}\} \tag{5.7}$$

This is closely related to the sample variance. Given the fact that the wavelet coefficients decompose the sample variance across scales, and the definition of the wavelet variance above, we can relate the sample variance and the wavelet variance

$$\sum_{j=1}^{\infty}\nu_X^2(\tau_j) = \sigma_X^2 \tag{5.8}$$

This relation enables us to link the wavelet variance with the theory of stochastic processes given in chapter 3.

5.2 Regularizing the SDF by the wavelet variance

In the previous section, we reviewed how the wavelet variance is related to the sample variance. The theory reviewed in chapter 3 describes how the SDF decompose the sample variance with respect to frequency. Since frequency and scale are closely related, there should be a link between the wavelet variance and the SDF. The aim of this section is to establish this connection. In the following we will assume that the wavelet coefficients are obtained by using a MODWT wavelet filter based upon Daubechies wavelets.

Let $\{X_t\}$ be a stochastic process whose dth order backward difference , $\{Y_t\}$, is a stationary process with SDF S_Y and mean μ_Y. Calculating the wavelet coefficients according to equation (5.6), the wavelet coefficients themselves can resemble a stationary process, S_j, given the right conditions

$$S_j = \left|\widetilde{H}_j^d\right|^2 (f)\, S_X\,(f) \tag{5.9}$$

Assuming that this holds, we can use the fact that the variance of a stationary process equals the integral of its SDF. Since the variance of $\{\overline{W}_{j,t}\}$ is defined as the wavelet variance, we have

$$\nu_X^2\,(\tau_j) = \int_{1/2}^{1/2} S_j\,(f)\,df = \int_{1/2}^{1/2} \left|\widetilde{H}_j^d\right|^2 (f)\, S_X\,(f)\,df \tag{5.10}$$

This is true as long as the integral is finite, that is, the wavelet coefficients resemble a stationary process. To find out which restrictions ensure this, we can take a look at the form of the integral for $j = 1$:

$$\nu_X^2\,(\tau_1) = \int_{1/2}^{1/2} \left|\widetilde{H}_1^D\right|^2 (f)\, S_X\,(f)\,df$$

$$= \frac{1}{4^d} \sum_{l=0}^{\frac{L}{2}-1} \binom{\frac{L}{2}-1+l}{l} \int_{-1/2}^{1/2} \cos\,(\pi f)^{2l} \sin\,(\pi f)^{L-2d} S_Y\,(f)\,df$$

The condition $L \geq 2d$ ensures that the sinusoidal term is bounded by unity and that the integral is finite. A similar approach using the squared gain function for Coiflet wavelets require $L \geq 3$.

5.3 Wavelet variance estimation

Now that we have established the theory connecting the wavelet variance with the SDF, we need to construct an estimator for the wavelet variance itself. In the following we will construct estimators of the wavelet variance based on MODWT coefficients and DWT coefficients. We will consider a finite process $\{X_t : t = 0, 1, 2, .., N-1\}$ whose dth order backward difference form a stationary process. Due to the properties of the wavelet coefficients discussed above, they seem like a natural choice for constructing our estimators. From the definition of the

wavelet variance, we have

$$\nu_X^2(\tau_j) = var\{\overline{W}_{j,t}\} = E\{\overline{W}_{j,t}^2\} - \left(E\{\overline{W}_{j,t}\}\right)^2 = E\{\overline{W}_{j,t}^2\}$$

if we have $E\{\overline{W}_{j,t}\} = 0$. This holds for a stationary process given the constraints presented at the end of the last section. For a non-stationary process, such as the fBm process we use as our model, we require the stronger conditions

$$L > 2d \quad \text{, Daubechies filters} \tag{5.11}$$

$$L > 3d \quad \text{, Coiflet filters} \tag{5.12}$$

to ensure that $E\{\overline{W}_{j,t}\} = 0$. In the following we require that the wavelet coefficients are computed according to these requirements.

5.3.1 MODWT based estimators

Before we proceed, we remind the reader about the definitions of the MODWT wavelet coefficients and the wavelet coefficients forming the wavelet variance:

$$\widetilde{W}_{j,t} = \sum_{l=0}^{L_j-1} \tilde{h}_{j,l} X_{t-l \bmod N}$$

and

$$\overline{W}_{j,t} = \sum_{l=0}^{L_j-1} \tilde{h}_{j,l} X_{t-l} \quad , t = ..., -1, 0, 1, ...$$

We can now construct an unbiased estimator of the wavelet variance based upon the MODWT coefficients. We do so by picking the MODWT coefficients not affected by the circularity assumption baked into the pyramid algorithm only. That is,, we omit the modulo operation. By doing so, we have

$$\widetilde{W}_{j,t} = \overline{W}_{j,t}$$

which implies

$$E\{\widetilde{W}_{j,t}^2\} = E\{\overline{W}_{j,t}^2\}$$

This holds for all $t \geq L_j - 1$. The unbiased MODWT based estimator then takes the form

$$\tilde{\nu}_X^2(\tau_j) = \frac{1}{M_j} \sum_{t=L_j-1}^{N-1} \widetilde{W}_{j,t}^2 \tag{5.13}$$

where $M_j = N - L_j + 1$. In order to be able to construct the unbiased MODWT estimator, we must have

$$N - L_j \geq 0 \tag{5.14}$$

We could also construct a biased estimator in a similar matter, allowing the boundary coefficients to be included in the estimator. The unbiased MODWT estimator has the form

$$\tilde{\tilde{\nu}}_X^2(\tau_j) = \frac{1}{N}\sum_{t=0}^{N-1} \widetilde{W}_{j,t}^2 \tag{5.15}$$

One practical advantage of constructing a biased estimator, is that it gives us more freedom because equation (5.14) does not apply.

Now that we have defined two estimators of the wavelet variance based upon the MODWT wavelet coefficients, we are able to compute a confidence interval for the wavelet variance. Percival (1995) approached this by claiming that the ratio between $\tilde{\nu}_X^2$ and ν_X^2 is approximately chi-square distributed

$$\frac{\eta\tilde{\nu}_X^2(\tau_j)}{\nu_X^2(\tau_j)} \approx \chi_\eta^2 \tag{5.16}$$

where η represents the equivalent degrees of freedom (EDOF), roughly speaking the number of wavelet coefficients in $\tilde{\nu}_X^2$. He argues that

$$\eta = \max\{\frac{M_j}{2^j}, 1\} \tag{5.17}$$

is a good approximation to the EDOF for $\tilde{\nu}_X^2$. An approximate $100(1-2p)\,\%$ would then be given as

$$\left[\frac{\eta\tilde{\nu}_X^2(\tau_j)}{Q_\eta(1-p)}, \frac{\eta\tilde{\nu}_X^2(\tau_j)}{Q_\eta(p)}\right] \tag{5.18}$$

where $Q_\eta(p)$ is the $p \times 100\%$ percentage point distribution for the χ_η^2. This approach ensures that ν_X^2 is "trapped" inside the interval given above, as well as the the lower confidence limit is non-negative. A useful numerical formula for computing $Q_\eta(p)$ is given by Chambers et al (1983)

$$Q_\eta(p) \approx \eta\left(1 - \frac{2}{9\eta} + \mathbf{\Phi}^{-1}(p)\left(\frac{2}{9\eta}\right)^{1/2}\right)^3 \tag{5.19}$$

where $\mathbf{\Phi}^{-1}(p)$ is the $p \times 100\%$ percentage point distribution for the standard Gaussian distribution.

Equations (5.16)-(5.18) summarize how to compute a confidence interval for the wavelet variance using the unbiased MODWT wavelet variance estimator. We can also construct a confidence interval for the wavelet variance using the biased MODWT estimator in the same way by changing the M_j in (5.17) to N.

5.3.2 DWT based estimators

In the previous subsection we looked at how to construct an estimator for the wavelet variance using MODWT wavelet coefficients. We can of course construct similar estimators using DWT

wavelet coefficients instead. As the DWT is naturally defined for sample sizes N on the form 2^J, let us start out by restricting ourselves to signals constrained by this. Now using DWT wavelet coefficients $\{W_{j,t} : t = 0, ..., N_j - 1\}$ based upon filtering the signal with $\{h_{j,l}\}$ using the pyramid algorithm, the DWT counterpart to equation (5.13) is

$$\breve{\nu}_X^2(\tau_j) = \frac{1}{\left(N_j - L'_j\right) 2^j} \sum_{t=L'_j}^{N_j-1} W_{j,t}^2 \tag{5.20}$$

yielding the unbiased DWT wavelet variance estimator. In order to arrive at this result, we have to adjust the limits in the sum so that only the DWT wavelet coefficients unaffected by the circularity assumption is included in the sum, as well as multiply the total number of such coefficients by a factor 2^j in the denominator to normalize the DWT wavelet coefficients according to the relationship between $\{h_{j,l}\}$ and $\{\tilde{h}_{j,l}\}$. From this expression it follows naturally that computing $\breve{\nu}_X^2(\tau_j)$ requires

$$N_j - L'_j > 0 \tag{5.21}$$

As with the MODWT based estimator, we can define a biased estimator. The biased DWT based estimator is

$$\breve{\breve{\nu}}_X^2(\tau_j) = \frac{1}{N_j 2^j} \sum_{t=0}^{N_j-1} W_{j,t}^2 \tag{5.22}$$

Let us now generalize these estimators to be defined for any sample size N. If we let M be the smallest integer number on the form 2^J greater than N, we can construct a new time series $\{X'_t\}$ by padding with $M - N$ zeros at the end of $\{X_t\}$. We are then able to use the DWT pyramid algorithm as usual, yielding wavelet coefficients $W'_{j,t}$. By using a similar approach as we did when we picked the $\{\widetilde{W}_{j,t}\}$ from $\{\overline{W}_{j,t}\}$ to form the unbiased MODWT based estimator of the wavelet variance for $\{X_t\}$ we can pick the coefficients $\{\widetilde{W}'_{j,t}\}$ from $\{X'_t\}$ to form another unbiased MODWT estimator. The relationship between $W'_{j,t}$ and $\widetilde{W}'_{j,t}$ is then

$$W'_{j,t} = 2^{j/2}\widetilde{W}'_{j,2^j(t+1)-1} \quad , t = 0, ..., \frac{M}{2^j} - 1$$

To form the generalized unbiased DWT estimator, we need to pick the DWT coefficients constrained by $L_j - 1 \leq 2^j(t+1) - 1 \leq N - 1$. By doing so, we end up with the subsequence of the DWT wavelet coefficients yielding an unbiased DWT based wavelet variance estimator

$$\breve{\nu}_X^2(\tau_j) = \frac{1}{M'_j 2^j} \sum_{t=L'_j}^{\lfloor N_j-1 \rfloor} W_{j,t}'^2 \tag{5.23}$$

where $M'_j = \lfloor \frac{N}{2^j} - 1 \rfloor - L'_j + 1$. Because this estimator reduces to (5.20) when N is of length 2^j, it can be regarded as a generalized version of (5.20). In order to be able to compute (5.23), we need to have

$$\lfloor \frac{N}{2^j} - 1 \rfloor - L'_j \geq 0 \tag{5.24}$$

We can also generalize the biased estimator (5.22). This is given as

$$\breve{\nu}_X^2(\tau_j) = \frac{1}{N_j' 2^j} \sum_{t=0}^{\lfloor N_j - 1 \rfloor} W_{j,t}'^2 \tag{5.25}$$

where $N_j' = \lfloor \frac{N}{2^j} - 1 \rfloor + 1$.

As with the MODWT estimators, the DWT based estimators can also be used to compute a confidence interval for the wavelet variance. It follows the same procedure as for the MODWT based estimator, except that the EDOFs for the generalized unbiased DWT based estimator is approximated as

$$\eta = \max\{M_j', 1\} \tag{5.26}$$

Substituting M_j' with N_j' gives the appropriate approximation for the generalized biased DWT estimator.

As an endnote to this subsection, it's worth mentioning that Percival (1995) compares the relative merits of the MODWT and DWT based estimators. He shows that a DWT based estimator cannot be more accurate than a MODWT based estimator, but indicates that the DWT based estimator will asymptotically converge towards the performance of the MODWT estimator as L grows. So by choosing a short wavelet filter, one should rely on a MODWT based estimator. If a relatively long filter is chosen, for example the $D20$ wavelet, then the choice of estimator is no longer obvious.

5.4 Least squares estimation of the Hurst exponent

5.4.1 Estimation through the wavelet variance

We have so far reviewed how the wavelet transform decompose the sample variance in a time series across scales, defined the concept of wavelet variance and looked at how this is related to the SDF. Skipping the details of equation (5.9), we can write this relation approximately as

$$\nu_X^2(\tau_j) \approx 2 \int_{1/2^{j+1}}^{1/2^j} S_X(f)\, df \tag{5.27}$$

If we let $S_X(f)$ represent the SDF for a PPL, we have $S_X(f) \propto |f|^\alpha$. By solving the integral and use the fact that scale and frequency are inverse quantities, we have

$$\nu_X^2(\tau_j) \propto \tau_j^{-\alpha-1}$$

As we remember from chapter 3, LMPs tend to have similar appearances of their SDFs in a log/log plot as the PPL, the highest frequencies being an exception. Translating this to our notion of scales and wavelet variance, a log/log plot of the wavelet variance over scales should yield a straight line if we skip the lowest scale(s). Specializing to our fBm-model, we have

$S_X(f) \propto |f|^{-2H_B-1}$ and

$$\nu_X^2(\tau_j) \propto \tau_j^{2H_B} \quad , j > j' \tag{5.28}$$

where j' is the largest scale where the approximation of linearity is poor. We can now construct a linear model based on the wavelet variance

$$\log\left(\nu_X^2(\tau_j)\right) \approx \xi + \beta \log(\tau_j) \quad , j > j' \tag{5.29}$$

where $\beta = 2H_B$. Unfortunately, we are not able to calculate $\nu_X^2(\tau_j)$ exact, so we have to make use of one of the estimators derived in the previous section. We will use the notation of an unbiased MODWT estimator, $\tilde{\nu}_X^2(\tau_j)$, throughout for simplicity, but what follows are valid for all the wavelet variance estimators derived in the last section. We will also use the notation $J_1 = j' + 1$ later on.

The relationship between $\nu_X^2(\tau_j)$ and $\tilde{\nu}_X^2(\tau_j)$ are given in equation (5.16). Taking the logarithm of the equation, we can rewrite it as

$$\log\left(\nu_X^2(\tau_j)\right) \approx \log\left(\tilde{\nu}_X^2(\tau_j)\right) - \log\left(\chi^2_{\eta_j}\right) + \log(\eta_j) \tag{5.30}$$

and construct a linear regression model based on

$$Y(\tau_j) = E\{\log\left(\tilde{\nu}_X^2(\tau_j)\right) - \log\left(\chi^2_{\eta_j}\right) + \log(\eta_j)\} = \log\left(\tilde{\nu}_X^2(\tau_j)\right) - \psi\left(\frac{\eta_j}{2}\right) + \log\left(\frac{\eta_j}{2}\right) \tag{5.31}$$

The linear regression model is then

$$Y(\tau_j) = \xi + \beta \log(\tau_j) + e_j \tag{5.32}$$

where the error term is as follows

$$e_j = \log\left(\frac{\hat{\nu}_X^2(\tau_j)}{\nu_X^2(\tau_j)}\right) - \psi\left(\frac{\eta_j}{2}\right) + \log\left(\frac{\eta_j}{2}\right) \tag{5.33}$$

We have here made use of a result from the literature, claiming that the expectation value of the logarithm of the chi-square distribution is

$$E\{\log\left(\chi^2_{\eta_j}\right)\} = \psi\left(\frac{\eta_j}{2}\right) + \log(2)$$

with variance

$$var\{\log\left(\chi^2_{\eta_j}\right)\} = \psi'\left(\frac{\eta_j}{2}\right)$$

where ψ and ψ' are the digamma and trigamma functions, the first and second derivatives of the gamma function, respectively.

Written in vector notation, equation (5.32) takes the form

$$\mathbf{Y} = A\mathbf{b} + \mathbf{e} \tag{5.34}$$

where $\mathbf{Y} \equiv [Y(\tau_{J_1}), ..., Y(\tau_{J_0})]^T$, A being a $(J_0 - J_1 + 1) \times 2$ matrix with column vectors $[1, ..., 1]^T$ and $[\log(\tau_{J_1}), ..., \log(\tau_{J_0})]^T$, and $\mathbf{b} \equiv [\xi, \beta]^T$. The random vector $\mathbf{e} \equiv [e_{J_1}, ..., e_{J_0}]^T$ has mean $\mathbf{0}$ and a diagonal covariance matrix $\sum_{\mathbf{e}}$ with nonzero elements $\psi'\left(\frac{\eta_{J_1}}{2}\right), ..., \psi'\left(\frac{\eta_{J_0}}{2}\right)$.

We can now apply the theory of least squares estimation to determine the value of β . For this purpose Percival & Walden (2006) recommend using a weighted least squares estimator (WLSE). The WLSE of $\mathbf{b}$ is

$$\mathbf{b} = \left(A^T \sum_{\mathbf{e}}^{-1} A\right)^{-1} A^T \sum_{\mathbf{e}}^{-1} \mathbf{Y}$$

with covariance matrix

$$\left(A^T \sum_{\mathbf{e}}^{-1} A\right)^{-1}$$

Inserting our linear model into the expressions above, the WLSE yields

$$\xi = \frac{\sum w_j \log^2(\tau_j) \sum w_j Y(\tau_j) - \sum w_j \log(\tau_j) \sum w_j \log(\tau_j) Y(\tau_j)}{\sum w_j \sum w_j \log^2(\tau_j) - \left(\sum w_j \log(\tau_j)\right)^2} \tag{5.35}$$

and

$$\beta = \frac{\sum w_j \sum w_j \log(\tau_j) Y(\tau_j) - \sum w_j \log(\tau_j) \sum w_j Y(\tau_j)}{\sum w_j \sum w_j \log^2(\tau_j) - \left(\sum w_j \log(\tau_j)\right)^2} \tag{5.36}$$

and

$$var\{\beta\} = \frac{\sum w_j}{\sum w_j \sum w_j \log^2(\tau_j) - \left(\sum w_j \log(\tau_j)\right)^2} \tag{5.37}$$

with weights $w_j \equiv 1/\psi'\left(\frac{\eta_j}{2}\right)$. By running numerical experiments on a set of FD processes, Percival & Walden (2006) find that $\sqrt{\frac{1}{4} var\{\beta\}}$ is close to the root mean square error of the estimates of δ, and therefore a good measure of the precision of the estimate. Given the relationship between β and H_B, we can estimate the Hurst exponent as

$$H_B = \frac{\beta}{2} \tag{5.38}$$

with standard deviation

$$SD\{H_B\} = \sqrt{\frac{1}{4} var\{\beta\}} \tag{5.39}$$

5.4.2 Instantaneous estimation

Now we have reviewed how to estimate the Hurst exponent through the wavelet variance. The procedure can be interpreted as averaging the square of a set of wavelet coefficients in time, fit a regression line, and estimate the Hurst exponent by the slope of this line [1]. It is however possible to apply the WLSE on the wavelet coefficients directly without doing any averaging

[1] Simonsen (2003) makes use of an even simpler averaging procedure using DWT wavelet coefficients, the so-called averaging wavelet coefficients method to estimate the Hurst exponent.

at all. To do so, we must require that the coefficients are co-located in time. We can assure this by applying a zero phase wavelet filter, for example a LA or Coiflet filter, and make the appropriate shifts as described in chapter ???. The wavelet variance estimators are then formed by picking a single, squared MODWT wavelet coefficient from each level. At time t we have the linear regression model

$$Y_t(\tau_j) = \log\left(\widetilde{W}^2_{j,t_j}\right) - \psi\left(\frac{1}{2}\right) - \log(2) \tag{5.40}$$

Using this model, the WLSE yields

$$H_B = \frac{1}{2}\left(\frac{(J_0 - J_1 + 1)\sum \log(\tau_j)\, Y_t(\tau_j) - \sum \log(\tau_j) \sum Y_t(\tau_j)}{(J_0 - J_1 + 1)\sum \log^2(\tau_j) - \left(\sum \log(\tau_j)\right)^2}\right) \tag{5.41}$$

$$SD\{H_B\} = \sqrt{\frac{1}{4} \cdot \frac{\pi^2 (J_0 - J_1 + 1)}{8(J_0 - J_1 + 1)\sum \log^2(\tau_j) - \left(\sum \log(\tau_j)\right)^2}} \tag{5.42}$$

Chapter 6

The program

6.1 Overview

6.2 Compiled code

In this section we will present the C++ code developed. The code is divided into two files, *wla.h* being a header file and *wla.cpp* containing the functions needed to perform the relevant calculations. To keep the code as "clean" as possible, most of the comments are removed, and a description of each function are given at the top of each code-snippet, including a reference to the relevant equations presented in the previous chapters.

wla.h:
The header file defines the classes and their member functions. The structure of the *dwt* and *modwt* classes is quite similar. The following code defines the *dwt* class:

```
class dwt{
 private:
  int l, type, nu;
  double *g, *h;

  void trans(int, double *, double *, double *);
  void itrans(int, double *, double *, double *);
  void init();
  int g_shift(int);
  int h_shift(int);

 public:
  dwt(int, int);
  ~dwt();
  void ptrans(int, int, double *, double **);
  void mra(int, int, double **, double **);
  void wavevar(int, int, int, int, double *, double *, double *, double **);
  void edof(int, int, int, double *);
  void shift(int, int, double **);
  void get_boundaries(int, int, int, double *);
  friend int get_nu(int, int);
```

```
  friend double get_daub(int, int);
  friend double get_la(int, int);
  friend double get_bl(int, int);
  friend double get_coiflet(int, int);
};
```

Its "big brother", the *modwt* class, has the structure:

```
class modwt{
 private:
  int l, type, nu;
  double *g, *h;

  int g_shift(int);
  int h_shift(int);
  void trans(int, int, double *, double *, double *);
  void itrans(int, int, double *, double *, double *);
  void init();

 public:
  modwt(int, int);
  ~modwt();
  void ptrans(int, int, double *, double **);
  void mra(int, int, double **, double **);
  void wavevar(int, int, int, int, double *, double *, double *, double **);
  void edof(int, int, int, double *);
  void shift(int, int, double **);
  void get_boundaries(int, int, int, double *);
  friend int get_nu(int, int);
  friend double get_daub(int, int);
  friend double get_la(int, int);
  friend double get_bl(int, int);
  friend double get_coiflet(int, int);
};
```

The *toolbox* class is intended to be the interface to the high level code. Its public functions are the algorithms available in the GUI. Any extension to the program should be done by adding a new "tool" to the toolbox and make it available in the GUI.

```
class toolbox{
 private:
  void reflection(int, double *&);
  void padding(int, int, int&, double *&);
  void truncate(int, double *&);
  void wlse(int, int, int, double *, double *, double *, double *, double *);
  void iwlse(int, int, int, double *, double *, double **);
  double digamma(double);
```

```
  double trigamma(double);
 public:
  void ahurst(int, int, int, int, int, int, int, int, int, int, double *, double *, double *, double *,
       double *);
  void ihurst(int, int, int, int, int, int, int, double *, double *, double *);
  void mra(int, int, int, int, int, int, double *, double *, double **);
  void wa(int, int, int, int, int, int, double *, double *, double **);
};
```

wla.cpp:
The code in this file makes up the meat needed to make the *wla.h*-skeleton any useful. It has the macros:

```
#include "wla.h"
#include <cmath>
```

The following code is the constructor for the dwt class. An object of this class needs two parameters at declaration:

int width : The width, or length, of the wavelet filter
int filter : Type of waveletfilter. Determines which bank of waveletfilters to use.

The class contains three variables and two arrays:
int l : Length of the wavelet filter
int type : type of wavelet filter. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)

int nu : Shift variable which value approximate the wavelet filters as linear phase filters if possible.
double *g : Scaling filter of length l.
double *h : Wavelet filter of length l.

The values of l and type are set here. The waveletfilters and the variable nu are initialized by a call to the function dwt::init().

Dependencies:
dwt::init()

```
dwt::dwt(int width, int  filter ){
  l = width;
  type = filter ;
  g = new double[l];
  h = new double[l];
  init ();
}
```

This is the destructor for the dwt class. Its purpose is to deallocate memory used by the wavelet filters.

```
dwt::~dwt(){
  delete [] g;
  delete [] h;
}
```

The void dwt::init() function initializes the scaling and wavelet filters according to the variables l and type. It also sets the value of nu if applicable.
Dependencies:
get_daub(int, int)
get_la(int, int)
get_bl(int, int)
get_coiflet(int, int)
get_nu(int, int)

```
void dwt::init(){
  if(type == 1){
    for(int i = 0; i < l; i++){
      g[i] = get_daub(l, i);
    }
  }else if(type == 2){
    for(int i = 0; i < l; i++){
      g[i] = get_la(l, i);
    }
    nu = get_nu(l, type);
  }else if(type == 3){
    for(int i = 0; i < l; i++){
      g[i] = get_bl(l, i);
    }
    nu = get_nu(l, type);
  }else if(type == 4){
    for(int i = 0; i < l; i++){
      g[i] = get_coiflet(l, i);
    }
    nu = get_nu(l, type);
  }

  for(int i = 0; i < l; i++){
    h[i] = pow(-1,i)*g[l-1-i];
  }
}
```

The function void dwt::trans transforms an input signal from level $j-1$ to j using the DWT pyramid algorithm.

Input parameters:
int n : Length of the input signal. Must be dyadic
double *v_in : Input signal to be transformed at level $j-1$
double *v_out : The transformed, or scaled, signal at level j
double *w_out : Wavelet coefficients at level j

Dependencies:
None

```
void dwt::trans(int n, double *v_in, double *v_out, double *w_out){
  int k;
  int nh = n/2;

  for(int i = 0; i < nh; i++){
    k = 2*i+1;
    v_out[i] = g[0]*v_in[k];
    w_out[i] = h[0]*v_in[k];
    for(int j = 1; j < l; j++){
      k--;
      if(k < 0){
  k = n-1;
      }
      v_out[i] += g[j]*v_in[k];
      w_out[i] += h[j]*v_in[k];
    }
  }
}
```

The inverse transform is performed by the function void dwt::itrans. It does the inverse transform of an input signal from level j to $j-1$ using the inverse DWT pyramid algorithm.

Input parameters:
int n : Length of the input signal
double *v_in : Input signal to be inverse transformed at level j
double *w_out : Wavelet coefficients at level j
double *x_out : The inverse transformed at level $j-1$

Dependencies:
None

```
void dwt::itrans(int n, double *v_in, double *w_in, double *x_out){
  int k, m, o;
  int p = -2;
  int q = -1;
  int lh = l/2;

  for(int i = 0; i < n; i++){
    k = i;
    m = 0;
    o = 1;
    p += 2;
    q += 2;
    x_out[p] = h[o]*w_in[k]+g[o]*v_in[k];
    x_out[q] = h[m]*w_in[k]+g[m]*v_in[k];
    if(l > 2){
      for(int j = 1; j < lh; j++){
  k++;
  if(k >= n){
    k = 0;
  }
  m += 2;
  o += 2;
  x_out[p] += h[o]*w_in[k]+g[o]*v_in[k];
  x_out[q] += h[m]*w_in[k]+g[m]*v_in[k];
      }
    }
  }
}
```

The function void dwt::ptrans performs a partial transform up to level J_0 using the DWT pyramid algorithm.

Input parameters:
int n : Length of the signal to be transformed
int j_0 : Maximum level of transformation
double *x_in : Input signal to be transformed
double **wv : A matrix to store wavelet coefficients at levels $j = 1, 2, ..., J_0$ as well as the scaling function at level J_0

Dependecies:
dwt::trans(int, double *, double *, double *)

```
void dwt::ptrans(int n, int j_0, double *x_in, double **wv){
  int nh = n/2;
  double *v_in = new double[n];
  double *v_out = new double[nh];
  double *w = new double[nh];

  for(int i = 0; i < n; i++){                          //Keep input signal intact
    v_in[i] = x_in[i];
  }

  for(int i = 0; i < j_0; i++){
    dwt::trans(n/pow(2,i), v_in, v_out, w);
    for(int j = 0; j < nh; j++){
      v_in[j] = v_out[j];
      wv[i][j] = w[j];                                 //Store wavelet coefficients
    }
    nh /= 2;
  }
  nh *= 2;
  for(int i = 0; i < nh; i++){
    wv[j_0][i] = v_out[i];                             //Store scaling coefficients
  }

  delete [] v_in;
  delete [] v_out;
  delete [] w;
}
```

The function void dwt::mra performs a multiresolution analysis of a signal utilizing its wavelet decomposition.

Input parameters:
int n : Length of the signal
int j_0 : MRA level / partial transform level
double **wv : Matrix containing wavelet coefficients at levels $j = 1, 2, ..., J_0$ as well as the scaling coefficients at level J_0
double **ds : Matrix to set the detail and smooth values of the MRA

Dependencies:
dwt::itrans(int, double *, double *, double *)

```
void dwt::mra(int n, int j_0, double **wv, double **ds){
  double *x_out = new double[n];
  double *zero = new double[n];
  double *tmp = new double[n];

  for(int i = 0; i < n; i++){
    zero[i] = 0;
  }

  for(int i = 0; i < j_0; i++){
    dwt::itrans(n/pow(2,i), zero, wv[i], x_out); //Compute details D_j
    for(int j = i-1; j >= 0; j--){
      for(int k = 0; k < n; k++){
  tmp[k] = x_out[k];
      }
      dwt::itrans(n/pow(2,j), tmp, zero, x_out);
    }
    for(int j = 0; j < n; j++){
      ds[i][j] = x_out[j];
    }
  }

  dwt::itrans(n/pow(2,j_0-1), wv[j_0], zero, x_out); //Compute smooth S_j0
  for(int k = j_0-2; k >= 0; k--){
    for(int i = 0; i < n; i++){
      tmp[i] = x_out[i];
    }
```

```
    dwt::itrans(n/pow(2,k), tmp, zero, x_out);
  }
  for(int j = 0; j < n; j++){
    ds[j_0][j] = x_out[j];
  }

  delete [] x_out;
  delete [] zero;
  delete [] tmp;
}
```

The function void dwt::wavevar estimates the wavelet variance based upon a DWT wavelet decomposition of the signal. Depending on the input parameters given, it can calculate an unbiased or a biased estimator according to equations (5.23) and (5.25). The control panel in the GUI ensures that the condition (5.24) is fulfilled. A level p confidence interval for the wavelet variance is also approximated.

Input parameters:
int bias : Determine if a biased or an unbiased estimator are to be calculated. (0:Unbiased, 1:Biased)
int n : Length of the signal. If the signal has been padded prior to the wavelet transform, this is the length of the unpadded signal
int j_0 : Maximum level to estimate the wavelet variance
int p : Determine percentage confidence interval to be approximated. Accepted values: 90, 95, 99
double *edof : Array of length J_0 containing equivalent degrees of freedom at each level
double *wvar : Array of length J_0 to set the estimated wavelet variance values
double *ci : Array of length $2 * J_0$. The lower confidence interval limits are set in the first half of the array, the upper confidence interval limits are set in the second half of the array
double **w : Matrix containing the wavelet coefficients. Must have a minimum first dimension $J_0 + 1$

Dependencies:
None

```
void dwt::wavevar(int bias, int n, int j_0, int p, double *edof, double *wvar, double *ci, double **w){
  int l_j, m_j, n_j, level, ii;
  double nusq, phi, eta, q_lower, q_upper;

  if(bias == 0){
    for(int i = 0; i < j_0; i++){
      nusq = .0;
      level = i+1;
      l_j = ceil((1-2)*(1-(1./pow(2,level))));
      n_j = floor(double(n)/pow(2,level)-1);
      m_j = n_j-l_j+1;
      for(int j = l_j; j <= n_j; j++){
  nusq += w[i][j]*w[i][j];
      }
      wvar[i] = nusq/(m_j*pow(2,level));
    }
  }else if(bias == 1){
    for(int i = 0; i < j_0; i++){
      nusq = .0;
      level = i+1;
      n_j = floor(double(n)/pow(2,level)-1)+1;
      for(int j = 0; j < n_j; j++){
  nusq += w[i][j]*w[i][j];
      }
      wvar[i] = nusq/(n_j*pow(2,level));
    }
  }

  if(p == 90){                                    //Calculate confidence interval
    phi = 1.6449;
  }else if(p == 95){
    phi = 1.96;
  }else if(p == 99){
    phi = 2.5758;
  }else{
    phi = 0;
  }

  for(int i = 0; i < j_0; i++){
    eta = edof[i];
    q_lower = eta*pow((1-2./(9*eta)+phi*sqrt(2./(9*eta))),3);
    phi *= -1;
    q_upper = fabs(eta*pow((1-2./(9*eta)+phi*sqrt(2./(9*eta))),3));
    phi *= -1;
    ii = j_0+i;
    ci[i] = log(eta*wvar[i]/q_lower);
    ci[ii] = log(eta*wvar[i]/q_upper);
  }
}
```

The function void dwt::edof calculates the equivalent degrees of freedom (EDOF) according to equation (5.26) for an unbiased estimator. For a biased estimator, the equation is modified according to the theory reviewed in chapter 5.

Input parameters:
int bias : Determine if the EDOFs are to be used by a biased or an unbiased estimator. (0:Unbiased, 1:Biased)
int n : Length of the signal. If the signal has been padded prior to the wavelet transform, this is the length of the unpadded signal
int j_0 : Number of EDOFs to be calculated
double *eta : Array to store the calculated EDOFs

Dependencies:
None

```
void dwt::edof(int bias, int n, int j_0, double *eta){
  int j, l_j;
  double m_j, n_j;

  if(bias == 0){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      l_j = ceil((l-2)*(1-(1./pow(2,j))));
      n_j = floor(double(n)/pow(2,j)-1);
      m_j = n_j-l_j+1;
      eta[i] = m_j;
      if(eta[i] < 1){
  eta[i] = 1;
      }
    }
  }else if(bias == 1){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      n_j = floor(double(n)/pow(2,j)-1)+1;
      eta[i] = n_j;
      if(eta[i] < 1){
  eta[i] = 1;
      }
    }
  }
}
```

The function int dwt::g_shift calculates the shift needed to allign the DWT scaling coefficients at level j with physical time according to equation ???.

Input parameters:
int j : This is assumed to be level $j-1$

Dependencies:
None

```
int dwt::g_shift(int j){
  j++;
  int l_j = ((pow(2,j)-1)*(l-1)+1);
  int nu_j = nu*(l_j-1)/(l-1);
  int gamma_j = ceil(((fabs(nu_j)+1)/pow(2,j))-1);

  return gamma_j;
}
```

The function int dwt::h_shift calculates the shift needed to allign the DWT wavelet coefficients at level j with physical time according to equation ???.

Input parameters:
int j : This is assumed to be level $j-1$

Dependencies:
None

```
int dwt::h_shift(int j){
  j++;
  int l_j = ((pow(2,j)-1)*(l-1)+1);
  int nu_j = -(l_j/2+l/2+nu-1);
  int gamma_j = ceil(((fabs(nu_j)+1)/pow(2,j))-1);

  return gamma_j;
}
```

The function void dwt::shift shifts the wavelet and scaling coefficients in the wv matrix to be alligned in physical time.

Input parameters:
int j_0 : Level of the transformation
int n : Length of the transformed signal
double **wv : Matrix containing wavelet and scaling coefficients. First dimension must be of length $J_0 + 1$

Dependencies:
dwt::g_shift(int)
dwt::h_shift(int)

```
void dwt::shift(int j_0, int n, double **wv){
  int shift, n_j;
  double *tmp = new double[n/2];

  for(int i = 0; i < j_0; i++){
    n_j = n/pow(2,i+1);
    shift = dwt::h_shift(i);
    for(int j = shift; j < n_j; j++){
      tmp[j-shift] = wv[i][j];
    }
    for(int j = 0; j < shift; j++){
      tmp[n_j-shift+j] = wv[i][j];
    }
    for(int j = 0; j < n_j; j++){
      wv[i][j] = tmp[j];
    }
  }
  shift = dwt::g_shift(j_0-1);
  for(int j = shift; j < n_j; j++){
    tmp[j-shift] = wv[j_0][j];
  }
  for(int j = 0; j < shift; j++){
    tmp[n_j-shift+j] = wv[j_0][j];
  }
  for(int j = 0; j < n_j; j++){
    wv[j_0][j] = tmp[j];
  }

  delete[] tmp;
}
```

The function void dwt::get_boundaries determines which wavelet, scaling or mra coefficients that are influenced by boundary conditions. The boundaries are initialized to a negative value, ie a negative value means that no coefficients are influenced by boundary conditions.

Input parameters:
int param : Assumes values 0,1,2. (0:Unshifted dwt coefficients,1:Shifted dwt coefficients, 2:Mra coefficients)
int j_0 : Level of transformation
int n : Length of transformed signal
double *boundaries : Array of length $2 * J_0 + 2$. The value of the first J_0 elements is the index of the last wavelet coefficient affected by boundary conditions at level element+1 in the beginning of the wavelet vector. The values of the nextJ_0 elements is the index of the first wavelet coefficient that are affected by boundary conditions at level element+1 $- J_0$ in the end of the wavelet vector. The last two elements contain the similar values for the scaling coefficients

Dependencies:
dwt::g_shift(int)
dwt::h_shift(int)

```
void dwt::get_boundaries(int param, int j_0, int n, double *boundaries){
  int j, l_j, n_j, gamma, gamma_hat;

  for(int i = 0; i <= 2*j_0+1; i++){                    // initialize boundaries
    boundaries[i] = -1;
  }

  if(param == 0){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      l_j = ceil((l-2)*(1-(1./pow(2,j))));
      boundaries[i] = l_j-1;
    }
    boundaries[2*j_0] = l_j-1;
  }else if(param == 1){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      l_j = ceil((l-2)*(1-(1./pow(2,j))));
      n_j = n/pow(2,j);
      gamma = dwt::h_shift(i);
      gamma_hat = l_j-gamma;
      boundaries[i] = gamma_hat-1;
      boundaries[i+j_0] = n_j-gamma;
    }
    gamma = dwt::g_shift(j_0-1);
    gamma_hat = l_j-gamma;
    boundaries[2*j_0] = gamma_hat-1;
    boundaries[2*j_0+1] = n_j-gamma;
  }else if(param == 2){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      l_j = ceil((l-2)*(1-(1./pow(2,j))));
      boundaries[i] = pow(2,j)*l_j-1;
      l_j = ((pow(2,j)-1)*(l-1)+1);
      boundaries[i+j_0] = n-(l_j-pow(2,j));
    }
    l_j = ceil((l-2)*(1-(1./pow(2,j))));
    boundaries[2*j_0] = pow(2,j)*l_j-1;
    l_j = ((pow(2,j)-1)*(l-1)+1);
    boundaries[2*j_0+1] = n-(l_j-pow(2,j));
  }
}
```

Constructor for the *modwt* class. An object of this class needs two parameters at declaration:

int width : The width, or length, of the wavelet filter
int filter : Type of waveletfilter. Determines which bank of waveletfilters to use.

The class contains three variables and two arrays:
int l : Length of the wavelet filter
int type : type of wavelet filter. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)

int nu : Shift variable which value approximate the wavelet filters as linear phase filters if possible.
double *g : Scaling filter of length l.
double *h : Wavelet filter of length l.

The values of l and type are set here. The waveletfilters and the variable nu are initialized by a call to the function modwt::init().

Dependencies:
modwt::init()

```
modwt::modwt(int width, int filter){
  l = width;
  type = filter;
  g = new double[l];
  h = new double[l];
  init();
}
```

Destructor for the *modwt* class. Deallocate memory for the wavelet filters.

```
modwt::~modwt(){
  delete [] g;
  delete [] h;
}
```

The void modwt::init() function initializes the scaling and wavelet filters according to the variables l and type. It also sets a value of nu if applicable.

Dependencies:
get_daub(int, int)
get_la(int, int)
get_bl(int, int)
get_coiflet(int, int)
get_nu(int, int)

```
void modwt::init(){
  if(type == 1){
    for(int i = 0; i < l; i++){
      g[i] = get_daub(l, i)/sqrt(2);
    }
  }else if(type == 2){
    for(int i = 0; i < l; i++){
      g[i] = get_la(l, i)/sqrt(2);
    }
    nu = get_nu(l, type);
  }else if(type == 3){
    for(int i = 0; i < l; i++){
      g[i] = get_bl(l, i)/sqrt(2);
    }
    nu = get_nu(l, type);
  }else if(type == 4){
    for(int i = 0; i < l; i++){
      g[i] = get_coiflet(l, i)/sqrt(2);
    }
    nu = get_nu(l, type);
  }

  for(int i = 0; i < l; i++){
    h[i] = pow(-1,i)*g[l-1-i];
  }
}
```

The function void modwt::trans transforms an input signal from level $j-1$ to j using the MODWT pyramid algorithm.

Input parameters:
int n : Length of the input signal. In contrast to the DWT, the MODWT is well defined for any sample size
int it : Iteration number. Equals level $j-1$
double *v_in : Input signal to be transformed at level $j-1$
double *v_out : The transformed, or scaled, signal at level j
double *w_out : Wavelet coefficients at level j

Dependencies:
None

```
void modwt::trans(int n, int it, double *v_in, double *v_out, double *w_out){
  int k;

  for(int i = 0; i < n; i++){
    k = i;
    v_out[i] = g[0]*v_in[k];
    w_out[i] = h[0]*v_in[k];
    for(int j = 1; j < l; j++){
      k -= pow(2,it);
      if(k < 0){
  k += n;
      }
      v_out[i] += g[j]*v_in[k];
      w_out[i] += h[j]*v_in[k];
    }
  }
}
```

The function void modwt::itrans does an inverse transform of an input signal from level j to $j-1$ using the inverse MODWT pyramid algorithm.

Input parameters:
int n : Length of the input signal
int it : Iteration number. Equals level $j-1$
double *v_in : Input signal to be inverse transformed at level j
double *w_out : Wavelet coefficients at level j
double *x_out : The inverse transformed at level $j-1$

Dependencies:
None

```
void modwt::itrans(int n, int it, double *v_in, double *w_in, double *x_out){
  int k;

  for(int i = 0; i < n; i++){
    k = i;
    x_out[i] = h[0]*w_in[k]+g[0]*v_in[k];
    for(int j = 1; j < l; j++){
      k += pow(2, it);
      if(k >= n){
  k -= n;
      }
      x_out[i] += h[j]*w_in[k]+g[j]*v_in[k];
    }
  }
}
```

The function void modwt::ptrans transforms a signal to level J_0.

Input parameters:
int n : Length of the signal to be transformed
int j_0 : Maximum level of transformation
double *x_in : Signal to be transformed
double **wv : Matrix to store wavelet and scaling coefficients. First dimension must be J_0+1, second dimension must be n.

Dependencies:
modwt::trans(int, int, double *, double *, double *)

```
void modwt::ptrans(int n, int j_0, double *x_in, double **wv){
  double *v_in = new double[n];
  double *v_out = new double[n];
  double *w = new double[n];

  for(int i = 0; i < n; i++){
    v_in[i] = x_in[i];
  }
  for(int i = 0; i < j_0; i++){
    modwt::trans(n, i, v_in, v_out, w);
    for(int j = 0; j < n; j++){
      v_in[j] = v_out[j];
      wv[i][j] = w[j];
    }
  }
  for(int j = 0; j < n; j++){
    wv[j_0][j] = v_out[j];
  }

  delete [] v_in;
  delete [] v_out;
  delete [] w;
}
```

The function void modwt::mra performs a multiresolution analysis of a signal utilizing its wavelet decomposition.

Input parameters:
int n : Length of the signal
int j_0 : MRA level / partial transform level
double **wv : Matrix containing wavelet coefficients at levels $j = 1, 2, ..., J_0$ as well as the scaling coefficients at level J_0
double **ds : Matrix to set the detail and smooth values of the MRA

Dependencies:
modwt::itrans(int, int, double *, double *, double *)

```
void modwt::mra(int n, int j_0, double **wv, double **ds){
  double *x_out = new double[n];
  double *zero = new double[n];
  double *tmp = new double[n];

  for(int i = 0; i < n; i++){
    zero[i] = 0;
  }

  for(int i = 0; i < j_0; i++){                    //Compute details D_j
    modwt::itrans(n, i, zero, wv[i], x_out);
    for(int j = i-1; j >= 0; j--){
      for(int k = 0; k < n; k++){
  tmp[k] = x_out[k];
      }
      modwt::itrans(n, j, tmp, zero, x_out);
    }
    for(int j = 0; j < n; j++){
      ds[i][j] = x_out[j];
    }
  }

  modwt::itrans(n, j_0-1, wv[j_0], zero, x_out);  //Compute smooth S_j0
  for(int k = j_0-2; k >= 0; k--){
    for(int i = 0; i < n; i++){
      tmp[i] = x_out[i];
    }
    modwt::itrans(n, k, tmp, zero, x_out);
  }
  for(int j = 0; j < n; j++){
    ds[j_0][j] = x_out[j];
  }

  delete [] x_out;
  delete [] zero;
  delete [] tmp;
}
```

The function void modwt::wavevar estimates the wavelet variance based upon a MODWT wavelet decomposition of the signal. Depending on the input parameters given, it can calculate

an unbiased or a biased estimator according to equations (5.13) and (5.15). The function assumes that equation (5.14) is fulfilled. This is ensured by the Python code. A level p confidence interval for the wavelet variance is also approximated.

Input parameters:
int bias : Determine if a biased or an unbiased estimator are to be calculated. (0:Unbiased, 1:Biased)
int n : Length of the signal
int j_0 : Maximum level to estimate the wavelet variance
int p : Determine percentage confidence interval to be approximated. Accepted values: 90, 95, 99
double *edof : Array of length J_0 containing equivalent degrees of freedom at each level
double *wvar : Array of length J_0 to set the estimated wavelet variance values
double *ci : Array of length $2 * J_0$. The lower confidence interval limits are set in the first half of the array, the upper confidence interval limits are set in the second half of the array
double **w : Matrix containing the wavelet coefficients. Must have a minimum first dimension $J_0 + 1$

Dependencies:
None

```
void modwt::wavevar(int bias, int n, int j_0, int p, double *edof, double *wvar, double *ci, double **w){
  int l_j, m_j, ii;
  double nusq, phi, eta, q_lower, q_upper;

  if(bias == 0){
    for(int i = 0; i < j_0; i++){
      nusq = .0;
      l_j = (pow(2,i+1)-1)*(l-1)+1;
      m_j = n-l_j+1;
      l_j--;
      for(int j = l_j; j < n; j++){
  nusq += w[i][j]*w[i][j];
      }
      wvar[i] = nusq/m_j;
    }
  }else if(bias == 1){
    for(int i = 0; i < j_0; i++){
      nusq = .0;
      for(int j = 0; j < n; j++){
  nusq += w[i][j]*w[i][j];
      }
      wvar[i] = nusq/n;
    }
  }

  if(p == 90){                                          //Calculate confidence interval
    phi = 1.6449;
  }else if(p == 95){
    phi = 1.96;
  }else if(p == 99){
    phi = 2.5758;
  }
  for(int i = 0; i < j_0; i++){
    eta = edof[i];
    q_lower = eta*pow((1-2./(9*eta)+phi*sqrt(2./(9*eta))),3);
    phi *= -1;
    q_upper = fabs(eta*pow((1-2./(9*eta)+phi*sqrt(2./(9*eta))),3));
    phi *= -1;
    ii = j_0+i;
    ci[i] = log(eta*wvar[i]/q_lower);
    ci[ii] = log(eta*wvar[i]/q_upper);
  }
}
```

The function void modwt::edof calculates the equivalent degrees of freedom (EDOF) according to equation (5.17). For a biased estimator, M_j in the equation are replaced by N, the length of the input signal.

Input parameters:
int bias : Determine if the EDOFs are to be used by a biased or an unbiased estimator. 0:Unbiased, 1:Biased
int n : Length of the signal
int j_0 : Number of EDOFs to be calculated
double *eta : Array to store the calculated EDOFs

Dependencies:
None

```
void modwt::edof(int bias, int n, int j_0, double *eta){
  int j, l_j, m_j;

  if(bias == 0){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      l_j = (pow(2,j)-1)*(l-1)+1;
      m_j = n-l_j+1;
      eta[i] = m_j/pow(2,j);
      if(eta[i] < 1){
  eta[i] = 1;
      }
    }
  }else if(bias == 1){
    for(int i = 0; i < j_0; i++){
      j = i+1;
      eta[i] = n/pow(2,j);
      if(eta[i] < 1){
  eta[i] = 1;
      }
    }
  }
}
```

The function int modwt::g_shift calculates the shift needed to allign the MODWT scaling coefficients with physical time according to equation ???.
Input parameters:
int j : This is assumed to be level $j-1$

Dependencies:
None

```
int modwt::g_shift(int j){
  j++;
  int l_j = (pow(2,j)-1)*(l-1)+1;
  int nu_j = nu*(l_j-1)/(l-1);

  return nu_j;
}
```

The function int modwt::h_shift calculates the shift needed to allign the MODWT wavelet coefficients with physical time according to equation ???.

Input parameters:
int j : This is assumed to be level $j-1$

Dependencies:
None

```
int modwt::h_shift(int j){
  j++;
  int l_j = (pow(2,j)-1)*(l-1)+1;
  int nu_j = -(l_j/2+l/2+nu-1);

  return nu_j;
}
```

The function void modwt::shift shifts the MODWT wavelet and scaling coefficients in the wv matrix to be alligned in physical time.

Input parameters:
int j_0 : Level of the transformation
int n : Length of the transformed signal
double **wv : Matrix containing wavelet and scaling coefficients. First dimension must be of length $J_0 + 1$

Dependencies:
modwt::g_shift(int)
modwt::h_shift(int)

```
void modwt::shift(int j_0, int n, double **wv){
  int shift;
  double *tmp = new double[n];

  for(int j = 0; j < j_0; j++){
    shift = fabs(modwt::h_shift(j));          //wavelet
    for(int i = shift; i < n; i++){
      tmp[i-shift] = wv[j][i];
    }
    for(int i = 0; i < shift; i++){
      tmp[n-shift+i] = wv[j][i];
    }
    for(int i = 0; i < n; i++){
      wv[j][i] = tmp[i];
    }
  }
  shift = fabs(modwt::g_shift(j_0-1));        //scaling
  for(int i = shift; i < n; i++){
    tmp[i-shift] = wv[j_0][i];
  }
  for(int i = 0; i < shift; i++){
    tmp[n-shift+i] = wv[j_0][i];
  }
  for(int i = 0; i < n; i++){
    wv[j_0][i] = tmp[i];
  }

  delete[] tmp;
}
```

The function void modwt::get_boundaries determines which wavelet, scaling or mra coefficients that are influenced by boundary conditions. The boundaries are initialized to a negative value, ie a negative value means that no coefficients are influenced by boundary conditions.

Input parameters:
int param : Assumes values 0,1,2. (0:Unshifted modwt coefficients, 1:Shifted modwt coefficients, 2:Mra coefficients)
int j_0 : Level of transformation
int n : Length of transformed signal
double *boundaries : Array of length $2*J_0+2$. The value of the first J_0 elements is the index of the last wavelet coefficient affected by boundary conditions at level element+1 in the beginning of the wavelet vector. The values of the next J_0 elements is the index of the first wavelet coefficient that are affected by boundary conditions at level element+1 $- J_0$ in the end of the wavelet vector. The last two elements contain the similar values for the scaling coefficients

Dependencies:
modwt::g_shift(int)
modwt::h_shift(int)

```
void modwt::get_boundaries(int param, int j_max, int n, double *boundaries){
  int l_j, nu;

  for(int i = 0; i <= 2*j_max+1; i++){
    boundaries[i] = -1;
  }

  if(param == 0){
    for(int i = 0; i < j_max; i++){
      l_j = (pow(2,i+1)-1)*(l-1)+1;
      boundaries[i] = l_j-2;
    }
    boundaries[2*j_max] = l_j-2;
  }else if(param == 1){
    for(int i = 0; i < j_max; i++){
      l_j = (pow(2,i+1)-1)*(l-1)+1;
      nu = modwt::h_shift(i);
      boundaries[i] = l_j-2-fabs(nu);
      boundaries[i+j_max] = n-fabs(nu);
    }
    nu = modwt::g_shift(j_max-1);
    boundaries[2*j_max] = l_j-2-fabs(nu);
    boundaries[2*j_max+1] = n-fabs(nu);
  }else if(param == 2){
    for(int i = 0; i < j_max; i++){
      l_j = (pow(2,i+1)-1)*(l-1)+1;
      boundaries[i] = l_j-2;
      boundaries[i+j_max] = n-l_j+1;
    }
    boundaries[2*j_max] = l_j-2;
    boundaries[2*j_max+1] = n-l_j+1;
  }
}
```

The function void toolbox::reflection modifies an array by adding a reflected copy of itself at the end.

Input parameters:
int n : Length of the original array
double *&x : Original array to be modified

Dependencies:
None

```
void toolbox:: reflection (int n, double *&x){
  double *tmp = new double[2*n];

  for(int i = 0; i < n; i++){
    tmp[i] = x[i];
    tmp[i+n] = x[n-i-1];
  }

  delete [] x;
  x = tmp;
}
```

The function void toolbox::padding checks if an input array is of dyadic length. If not, it is padded with either zeros or the sample mean, depending on the input parameters.

Input parameters:
int padding : Determines if array is to be padded with zeros or sample mean. (0:Zeros 1:Sample mean)
int n : Length of original array
int &m : Length of padded array. It is set to be the smallest power of two greater than n. If n is dyadic, m and n is equal
double *&x : Array to be padded

Dependencies:
None

```
void toolbox::padding(int padding, int n, int &m, double *&x){
  int j = 0;
  double value;
  double *tmp;

  if(padding == 0){
    value = 0;
  }else if(padding == 1){
    value = .0;
    for(int i = 0; i < n; i++){
      value += x[i];
    }
    value /= n;
  }

  while(pow(2,j) < n){
    j++;
  }
  m = pow(2,j);
  if(m > n){
    tmp = new double[m];
    for(int i = 0; i < n; i++){
      tmp[i] = x[i];
    }
    for(int i = n; i < m; i++){
      tmp[i] = value;
    }
    delete [] x;
    x = tmp;
  }
}
```

The function void toolbox::truncate truncates an array to a given length

Input parameters:
int n : Length of the truncated array
double *&x : Array to be truncated

Dependencies:
None

```
void toolbox::truncate(int n, double *&x){
  double *tmp = new double[n];

  for(int i = 0; i < n; i++){
    tmp[i] = x[i];
  }
  delete [] x;
  x = tmp;
}
```

The function void toolbox::wlse fits a weighted least squares regression line to the estimated wavelet variance. It also calculate the hurst exponent, its variance and its standard deviation.

Input parameters:
int j_0 : Number of wavelet variance estimates
int j_min : Lowest level to fit the regression line
int j_max : Highest level to fit the regression line
double *edof : Array of length J_0 containing the EDOFs of each level
double *wvar : Array of length J_0 containing the wavelet variance estimates of each level
double *y : Array of length J_0 to store the logarithm of the "true" wavelet variance according to equation (5.31)
double *y_hat : Array of length $j_{max} - j_{min} + 1$ to store the values of the the regression line. The value of y_hat is calculated at the points τ_j in the range $j_{min} \leq \tau_j < j_{max}$ according to equation (5.29).
double *output : Array of length 3 to store the hurst exponent, the variance of the β parameter and the standard deviation of the Hurst exponent estimate according to equations (5.38), (5.37) and (5.39) respectively.

Dependencies:
toolbox::digamma(double)
toolbox::trigamma(double)

```
void toolbox::wlse(int j_0, int j_min, int j_max, double *edof, double *wvar, double *y, double *y_hat,
      double *output){
  double arg, beta, eta, zeta, var;
  double *tau = new double[j_0];
  double *w = new double[j_0];

  for(int i = 0; i < j_0; i++){
    arg = edof[i]/2;
    w[i] = 1./trigamma(arg);
    y[i] = log(wvar[i])-digamma(arg)+log(arg);
    tau[i] = pow(2,i);
  }

  double a = .0;
  double b = .0;
  double c = .0;
  double d = .0;
  double e = .0;
  double f = .0;

  j_min--;
  for(int i = j_min; i < j_max; i++){
    a += w[i];
    b += w[i]*log(tau[i])*y[i];
    c += w[i]*log(tau[i]);
    d += w[i]*y[i];
    e += w[i]*pow(log(tau[i]),2);
    f += w[i]*log(tau[i]);
  }

  beta = (a*b-c*d)/(a*e-f*f);
  zeta = (e*d-c*b)/(a*e-f*f);
  var = a/(a*e-f*f);
  output[0] = beta/2;
  output[1] = var;
  output[2] = sqrt(.25*var);

  for(int i = j_min; i < j_max; i++){
    y_hat[i] = zeta+beta*log(tau[i]);
  }

  delete [] tau;
  delete [] w;
}
```

The function void toolbox::iwlse estimate Hurst exponents based on the instantaneous estimator of equation (5.41). The final Hurst exponent is calculated as the average of all the estimated Hurst exponents, including the coefficients influenced by boundary conditions. The standard deviation of the estimate, who is constant at each point in time, is calculated according to equation (5.42).

Input parameters:
int j_min : Lowest level to fit the regression line
int j_max : Highest level to fit the regression line
int n : Length of the transformed signal
double *hurst : Array of length n to store the estimated Hurst exponent at each point in time
double *output : Array of length 3 to store the hurst exponent, the variance of the β parameter and the standard deviation of the Hurst estimate.
double **w : Matrix containg the wavelet coefficients. The coefficients must have been calculated using a linear phase filter and shifted so that the coefficients are alligned with physsical time

Dependencies:
None

```
void toolbox::iwlse(int j_min, int j_max, int n, double *hurst, double *output, double **w){
  int jj = j_max-j_min+1;
  double eul = 0.5772156649015328606; //Euler-Mascheroni constant
  double pi = 3.14159265358979323846; //The well known PI
  double a, b, c, d, var;
  double avgh = .0;
  double *tau = new double[j_max];
  double *y = new double[j_max];
  double test;
  j_min--;

  for(int i = 0; i < n; i++){//time
    for(int j = j_min; j < j_max; j++){//scale
      tau[j] = pow(2,j);
      y[j] = log(pow(w[j][i],2))+log(2)+eul;
    }
    a = .0;
    b = .0;
    c = .0;
    d = .0;
    for(int j = j_min; j < j_max; j++){
      a += log(tau[j])*y[j];
      b += log(tau[j]);
      c += y[j];
      d += pow(log(tau[j]),2);
    }

    hurst[i] = .5*(jj*a-b*c)/(jj*d-b*b);
    avgh += hurst[i];
  }
  var = (jj*pow(pi,2))/(8*(jj*d-b*b));
  output[0] = avgh/n;
  output[1] = var;
  output[2] = sqrt(.25*var);

  delete [] tau;
  delete [] y;
}
```

The function double toolbox::digamma returns the digamma value of an input. It is based on a code found at: http://people.sc.fsu.edu/ burkardt/cpp_src/prob/prob.

Input parameters:
double x : Input value

Dependencies:
None

```
double toolbox::digamma(double x){
```

```
  double s3 = 0.08333333333;
  double s4 = 0.0083333333333;
  double s5 = 0.003968253968;
  double value = .0;
  double y, z;

  z = x;
  while(z < 10){
    value -= 1./z;
    z += 1.;
  }
  y = 1./pow(z,2);
  value += log(z)-.5/z - y*(s3-y*(s4-y*s5));

  return value;
}
```

The function double toolbox::trigamma returns the trigamma value of an input. It is based on an asymptotic expansion in terms of Bernoulli numbers.

Input parameters:
double x : Input value

Dependencies:
None

```
double toolbox::trigamma(double x){
  double b2 = 1/6;
  double b4 = -1/30;
  double b6 = 1/42;
  double b8 = -1/30;
  double b10 = 5/66;
  double value = .0;
  double y, z;

  z = x;
  while(z < 10000){
    value += 1./pow(z,2);
    z += 1.;
  }
  y = 1./pow(z,2);
  value += .5*y+(1.+y*(b2+y*(b4+y*(b6+y*(b8+y*b10)))))/z;

  return value;
}
```

The function void toolbox::ahurst is one of the four "tools" developed for this project. It estimates the Hurst exponent by averaging a set of wavelet coefficients and fitting a regression

line to these. The "machinery" depends on the input parameters.

Input parameters:
int transtype : Determines if the averaged Hurst estimate is to be based on DWT or MODWT wavelet coefficients. (0:DWT, 1:MODWT)
int l : Length of wavelet filter
int type : Type of waveletfilter. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)

int bias : Determine if the wavelet variance estimate is biased or not. (0:Unbiased, 1:Biased)
int refl : Determine boundary conditions. (0:Circular, 1:Reflection)
int n : Length of input signal
int j_0 : Level of transformation
int j_min : Lowest level to fit regression line
int j_max : Highest level to fit the regression line
int p : Determine width of wavelet variance confidence interval. Accepted values:90,95,99
double *x_in : Input signal
double *y : Array of length J_0 to store the true wavelet variances
double *y_hat : Array of length $j_{max} - j_{min} + 1$ to store the points of the regression line
double *ci : Array of length $2 * J_0$ to store the minimum and maximum values of the confidence intervals
double *output : Array of length 3 to store the Hurst exponent, the variance of β and the standard deviation of the Hurst estimate

Dependencies:
dwt::ptrans(int, int, double *, double **)
dwt::edof(int, int, int, double *)
dwt::wavevar(int, int, int, int, double *, double *, double *, double **)
modwt::ptrans(int, int, double *, double **)
modwt::edof(int, int, int, double *) modwt::wavevar(int, int, int, int, double *, double *, double *, double **)
toolbox::reflection(int, double *&)
toolbox::padding(int, int, int &, double *)
toolbox::truncate(int, double *&)
toolbox::wlse(int, int, int, double *, double *, double *, double *, double *)

```
void toolbox::ahurst(int transtype, int l, int type, int bias, int refl, int n, int j_0, int j_min, int
    j_max, int p,
        double *x_in, double *y, double *y_hat, double *ci, double *output){
  int pad = 0;
  double *edof = new double[j_0];
  double *wvar = new double[j_0];
  double **wv = new double *[j_0+1];

  if(transtype == 0){
    int m;
    dwt object(l, type);
    if(refl == 0){
      toolbox::padding(pad, n, m, x_in);
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[m/2];
      }
      object.ptrans(m, j_0, x_in, wv);
      object.edof(bias, n, j_0, edof);
      object.wavevar(bias, n, j_0, p, edof, wvar, ci, wv);
      toolbox::wlse(j_0, j_min, j_max, edof, wvar, y, y_hat, output);
      toolbox::truncate(n, x_in);
    }else if(refl == 1){
      toolbox::padding(pad, n, m, x_in);
      toolbox:: reflection (m, x_in);
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[2*m];
      }
      object.ptrans(2*m, j_0, x_in, wv);
      object.edof(bias, n, j_0, edof);
      object.wavevar(bias, 2*n, j_0, p, edof, wvar, ci, wv);
      toolbox::wlse(j_0, j_min, j_max, edof, wvar, y, y_hat, output);
      toolbox::truncate(n, x_in);
    }
  }else if(transtype == 1){
    modwt object(l, type);
    if(refl == 0){
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[n];
      }
      object.ptrans(n, j_0, x_in, wv);
      object.edof(bias, n, j_0, edof);
      object.wavevar(bias, n, j_0, p, edof, wvar, ci, wv);
      toolbox::wlse(j_0, j_min, j_max, edof, wvar, y, y_hat, output);
    }else if(refl == 1){
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[2*n];
      }
      toolbox:: reflection (n, x_in);
      object.ptrans(2*n, j_0, x_in, wv);
      object.edof(bias, n, j_0, edof);
      object.wavevar(bias, 2*n, j_0, p, edof, wvar, ci, wv);
      toolbox::wlse(j_0, j_min, j_max, edof, wvar, y, y_hat, output);
      toolbox::truncate(n, x_in);
```

```
    }
  }

  for(int j = 0; j <= j_0; j++){
    delete [] wv[j];
  }
  delete [] wv;
  delete [] wvar;
  delete [] edof;
}
```

The function void toolbox::ihurst is one of the four "tools" developed for this project. It performs a wavelet transform based on the MODWT pyramid algorithm, shifts the coefficients tho be alligned with physical time and fits a regression line at each point in time and estimates the Hurst exponent as an average of these.

Input parameters:
int l : Length of wavelet filter
int type : Type of wavlet filter. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)

int refl : Determine boundary conditions. (0:Circular, 1:Reflection)
int j_0 : Level of transformation
int j_min : Lowest level to fit regression line
int j_max : Highest level to fit the regression line
int n : Length of input signal
double *x_in : Input signal
double *hurst : Array of length n to store the estimated values of the Hurst exponent at each point in time
double *output : Array of length 3 to store the Hurst exponent, its variance and its standard deviation

Dependencies:
modwt::ptrans(int, int, double *, double **)
modwt::shift(int, int, double **)
toolbox::reflection(int, double *&)
toolbox::truncate(int, double *&)
toolbox::iwlse(int, int, int, double *, double *, double **)

```
void toolbox::ihurst(int l, int type, int refl, int j_0, int j_min, int j_max, int n,
          double *x_in, double *hurst, double *output){
  modwt object(l, type);
  double **ds = new double *[j_0+1];
  double **wv = new double *[j_0+1];

  if(refl == 0){
    for(int j = 0; j <= j_0; j++){
      wv[j] = new double[n];
    }
    object.ptrans(n, j_0, x_in, wv);
    object.shift(j_0, n, wv);
    toolbox::iwlse(j_min, j_max, n, hurst, output, wv);
  }else if(refl == 1){
    for(int j = 0; j <= j_0; j++){
      wv[j] = new double[2*n];
    }
    toolbox::reflection(n, x_in);
    object.ptrans(2*n, j_0, x_in, wv);
    for(int i = 0; i <= j_0; i++){
      toolbox::truncate(n, wv[i]);
    }
    object.shift(j_0, n, wv);
    toolbox::iwlse(j_min, j_max, n, hurst, output, wv);
  }

  for(int j = 0; j <= j_max; j++){
    delete[] wv[j];
    delete[] ds[j];
  }
  delete[] ds;
  delete[] wv;
}
```

The function void toolbox::mra performs an multiresolution analysis of a given signal based on either a DWT or MODWT pyramid algorithm and calculates which coefficients are influenced by boundary conditions.

Input parameters:
int transtype : Determines if a DWT or MODWT algorithm is to be used
int l : Length of wavelet filter
int type : Type of wavelet. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)
int refl : Determine boundary conditions. (0:Circular, 1:Reflection)
int j_0 : Level of transformation
int n : Length of input signal
double *boundaries : Array of length $2 * J_0 + 2$ to store placements of boundary coefficients
double *x_in : Input signal
double **ds : Matrix to store the detail and smooth coefficients. First dimension of length $J_0 + 1$, second dimension of length

Dependencies:
dwt::ptrans(int, int, double *, double **)
dwt::mra(int, int, double **, double **)
dwt::get_boundaries(int, int, int, double *)
modwt::ptrans(int, int, double *, double **)
modwt::mra(int, int, double **, double **)
modwt::get_boundaries(int, int, int, double *)
toolbox::padding(int, int, int, double *)
toolbox::reflection(int, double *&)
toolbox::truncate(int, double *&)

```
void toolbox::mra(int transtype, int l, int type, int refl, int j_0, int n,
      double *boundaries, double *x_in, double **ds){
  int pad = 1;
  int mra = 2;
  double **wv = new double *[j_0+1];

  if(transtype == 0){
    int m;
    dwt object(l, type);

    if(refl == 0){
      toolbox::padding(pad, n, m, x_in);
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[m];
      }
      object.ptrans(m, j_0, x_in, wv);
      object.mra(m, j_0, wv, ds);
      object.get_boundaries(mra, j_0, m, boundaries);
      toolbox::truncate(n, x_in);
    }else if(refl == 1){
      toolbox::padding(pad, n, m, x_in);
```

```
      toolbox:: reflection (m, x_in);
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[2*m];
      }
      object.ptrans(2*m, j_0, x_in, wv);
      object.mra(2*m, j_0, wv, ds);
      object.get_boundaries(mra, j_0, m, boundaries);
      for(int j = 0; j <= j_0; j++){
  toolbox::truncate(m, ds[j]);
      }
      toolbox::truncate(n, x_in);
    }
  }else if(transtype == 1){
    modwt object(l, type);

    if(refl == 0){
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[n];
      }
      object.ptrans(n, j_0, x_in, wv);
      object.mra(n, j_0, wv, ds);
      object.get_boundaries(mra, j_0, n, boundaries);
    }else if(refl == 1){
      for(int j = 0; j <= j_0; j++){
  wv[j] = new double[2*n];//Not sure?
      }
      toolbox:: reflection (n, x_in);
      object.ptrans(2*n, j_0, x_in, wv);
      object.mra(2*n, j_0, wv, ds);
      object.get_boundaries(mra, j_0, n, boundaries);
      for(int j = 0; j <= j_0; j++){
  toolbox::truncate(n, ds[j]);
      }
      toolbox::truncate(n, x_in);
    }
  }

  for(int j = 0; j <= j_0; j++){
    delete [] wv[j];
  }
  delete [] wv;
}
```

The function void toolbox::wa performes a wavelet analysis of a signal based on a DWT or

MODWT pyramid algorithm. It also calculates which coefficients that are influenced by boundary conditions.

Input parameters:
int transtype : Determines if a DWT or MODWT algorithm is to be used
int l : Length of wavelet filter
int type : Type of wavelet. (1:Daubechies, 2:Least Assymetrical, 3:Best Localized, 4:Coiflet)
int refl : Determine boundary conditions. (0:Circular, 1:Reflection)
int j_0 : Level of transformation
int n : Length of input signal
double *boundaries : Array of length $2 * J_0 + 2$ to store placements of boundary coefficients
double *x_in : Input signal
double **wv : Matrix containing wavelet coefficients at levels $j = 1, 2, ..., J_0$ as well as the scaling coefficients at level J_0

Dependencies:
dwt::ptrans(int, int, double *, double **)
dwt::shift(int, int, double **)
dwt::get_boundaries(int, int, int, double *)
modwt::ptrans(int, int, double *, double **)
modwt::shift(int, int, double **)
modwt::get_boundaries(int, int, int, double *)
toolbox::padding(int, int, int, double *)
toolbox::reflection(int, double *&)
toolbox::truncate(int, double *&)

```
void toolbox::wa(int transtype, int l, int type, int refl, int j_max, int n,
     double *boundaries, double *x_in, double **wv){
  int shifted = 0;
  int pad = 1;

  if(transtype == 0){
    int m;
    dwt object(l, type);
    if(refl == 0){
    toolbox::padding(pad, n, m, x_in);
    object.ptrans(m, j_max, x_in, wv);
    if(type != 1){
      shifted = 1;
      object.shift(j_max, m, wv);
    }
    object.get_boundaries(shifted, j_max, n, boundaries);
    toolbox::truncate(n, x_in);
    }else if(refl == 1){
      toolbox::padding(pad, n, m, x_in);
      toolbox::reflection(m, x_in);
      object.ptrans(2*m, j_max, x_in, wv);
      for(int i = 0; i <= j_max; i++){
  toolbox::truncate(m/pow(2,i), wv[i]);
      }
      toolbox::truncate(n, wv[j_max]);
      if(type != 1){
  object.shift(j_max, m, wv);
      }
      object.get_boundaries(shifted, j_max, n, boundaries);
      toolbox::truncate(n, x_in);
    }
  }else if(transtype == 1){
    modwt object(l, type);
    if(refl == 0){
      object.ptrans(n, j_max, x_in, wv);
      if(type != 1){
  shifted = 1;
  object.shift(j_max, n, wv);
      }
      object.get_boundaries(shifted, j_max, n, boundaries);
    }else if(refl == 1){
      toolbox::reflection(n, x_in);
      object.ptrans(2*n, j_max, x_in, wv);
      for(int i = 0; i <= j_max; i++){
  toolbox::truncate(n, wv[i]);
      }
      if(type != 1){
  object.shift(j_max, n, wv);
      }
      object.get_boundaries(shifted, j_max, n, boundaries);
      toolbox::truncate(n, x_in);
    }
  }
```

```
}
```

The function int get_nu returns the class variable nu. The value of the variable depends on the length and type of the chosen wavelet filter.

Input parameters:
int l : Length of wavelet filter
int type : Type of wavelet filter. (2:Least Assymetrical, 3:Best Localized, 4:Coiflet). Daubechies filters do not have a shift parameter.

Dependencies:
None

```
int get_nu(int l, int type){
  if(type == 2){
    if(l == 14){
      return -l/2+2;
    }else if(l == 10 || l == 18){
      return -l/2;
    }else{
      return -l/2+1;
    }
  }else if(type == 3){
    if(l == 14){
      return -5;
    }else if(l == 18){
      return -11;
    }else if(l == 20){
      return -9;
    }
  }else if(type == 4){
    return -2*l/3+1;
  }
}
```

The function double get_daub is one of the wavelet filter banks. It contains Daubechies wavelet filters of length 2-20. The wavelet coefficients of $D4-D20$ have been cut out. The other wavelet filter banks have a similar structure.

Input parameters:
int l : Length of wavelet filter
int i : Coefficient index

Dependencies:
None

```
double get_daub(int l, int i){
  const double d2[2] = {
    0.7071067811865475,
    0.7071067811865475
  };
 (...)
  if(l == 2){
    return d2[i];
  }else if(l == 4){
    return d4[i];
  }else if(l == 6){
    return d6[i];
  }else if(l == 8){
    return d8[i];
  }else if(l == 10){
    return d10[i];
  }else if(l == 12){
    return d12[i];
  }else if(l == 14){
    return d14[i];
  }else if(l == 16){
    return d16[i];
  }else if(l == 18){
    return d18[i];
  }else if(l == 20){
    return d20[i];
  }
}
```

6.3 The Python module

6.4 The GUI

Figure 6.1: A snap shot of the GUI with a file opened.

Figure 6.2: A snap shot of the GUI showing epoch3.

Figure 6.3: The regression line fitte to the biased wavelet variance estimate using the $D6$ wavelet and reflection boundary conditions. The computed values are shown to the right. The simulated time-series have only one epoch, so the values of epoch 2 and 3 are 0.

Figure 6.4: A snap shot of the GUI in full bloom. The MRA is shown in the main window. This file has only one epoch still, so the output from aHurst and iHurst in epoch2 and 3 is rubbish.

Figure 6.5: The preferences panel. Here the options for the aHurst tool is shown.