GPU-Accelerated Wavelet Transforms for the Analysis of High Frequency Data

a thesis presented for the degree of
Doctor of Philosophy of Imperial College London

and the

Diploma of Imperial College

by

Julian Mark Waton

Department of Mathematics
Imperial College London
London
SW7 2AZ

February 2019

Statement of originality

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Statement of copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Supervisor: Professor Emma McCoy

GPU-Accelerated Wavelet Transforms for the Analysis of High Frequency Data

Abstract

This thesis describes new methodologies for wavelet analysis of high frequency data, and presents fast, GPU-accelerated implementations in a new R package, WaveCUDA. We address two main features of high frequency data, namely their large size and also the irregular spacing of their time values.

To deal with the size issue, we provide an R package with new parallel algorithms to perform wavelet transforms with the Lifting Scheme. We factorise the Coiflet 6 and Least Asymmetric 8 wavelet transforms into lifting steps and implement the Haar and Daubechies Extremal Phase 4 filters. We achieve considerable speedups, both in these transforms and in code that carries out thresholding schemes for nonparametric regression. We implement standard thresholding rules and provide a fast implementation of the potentially computationally expensive wavelet cross validation method of choosing the threshold. We then present a new regularised V-fold wavelet cross validation scheme for non-dyadic V.

We contribute to wavelet analysis of irregularly spaced data by using an entropy-based method for selecting an optimal grid at which to interpolate.

Acknowledgements

I would like to thank my supervisor, Prof Emma McCoy, for guiding and motivating me throughout the last four years, for introducing me to wavelets all those years ago and for some very entertaining meetings. I am very grateful to BP for financing this research and for offering me the chance to merge the academic and corporate worlds. I would like to thank my manager Philip for allowing me to give enough time to both, and for sharing his pragmatic approach to applying mathematics and statistics to the real world.

I would like to thank my examiners for their critique which allowed me to transform this thesis into a substantially better work.

Thanks to all in the statistics section who gave me guidance during my time at Imperial in statistics, programming and Emacs.

Lastly, I would like to thank my family and friends who offered proof-reading services in return for an insight into the world of wavelets, and giving me something to look forward to at the end of it all.

Contents

List of Figures **13**

List of Tables **14**

1 Introduction **17**

1.1 Structure of Thesis 18

1.2 Software Used 19

2 Wavelets **21**

2.1 The Continuous Wavelet Transform 21

2.2 The Discrete Wavelet Transform 22

2.2.1 DWT as a Matrix Transformation 24

2.2.2 The Pyramid Algorithm 26

2.2.3 DWT Filter Properties 26

2.3 Wavelet test signals 28

2.4 Wavelet filters 29

2.4.1 Haar 30

2.4.2 Daubechies 4 30

2.4.3 Coiflet 6 31

2.4.4 Least Asymmetric 8 31

2.4.5 Comparison of DWT for Different Filters 32

2.5 MODWT . . . 33
2.5.1 Comparison of MODWT for Different Filters . . . 34
2.5.2 Shift invariance of the MODWT . . . 34
2.6 DWPT . . . 35
2.7 Biorthogonal Wavelets . . . 35
2.8 Thresholding . . . 35
2.8.1 Thresholding Policies . . . 36
2.8.2 Levels for Thresholding . . . 37
2.8.3 Thresholding Schemes . . . 38
2.9 Literature Review . . . 46

3 A Review of Lifting Schemes 51

3.1 Laurent Polynomials . . . 52
3.2 The Wavelet Transform . . . 53
3.3 Lifting . . . 55
3.4 Euclidean and Lifting Algorithms . . . 56
3.5 Lifted Filters . . . 58
3.5.1 Haar Wavelet . . . 59
3.5.2 Daubechies 4 Wavelet . . . 59
3.5.3 Coiflet 6 Wavelet . . . 60
3.5.4 Least Asymmetric 8 Wavelet . . . 63

4 CUDA 67

4.1 Introduction . . . 67
4.2 CUDA Architecture . . . 68
4.3 CUDA C . . . 69
4.3.1 Optimisation in CUDA . . . 72

5 WaveCUDA **75**

5.1 Algorithmic introduction to wavelets 76

5.1.1 Parallelising the Wavelet Transform 78

5.2 CUDA-Accelerated Wavelet Transforms 78

5.2.1 Haar DWT 79

5.2.2 DWTs with Longer Filters 79

5.2.3 Haar MODWT 80

5.3 Thresholding and Smoothing 80

5.3.1 Wavelet Cross Validation 80

5.4 Using WaveCUDA 81

5.4.1 Summary of Implementation 84

5.5 Benchmarking WaveCUDA 84

6 Wavelet Cross Validation **87**

6.1 Standard Cross Validation 87

6.1.1 Introduction 87

6.1.2 Delete-d 88

6.1.3 V-fold 89

6.1.4 Computationally Efficient CV(1) 89

6.1.5 Generalised Cross Validation 90

6.2 Cross Validation in Autoregressive models 90

6.3 Wavelet Cross Validation Schemes 91

6.3.1 Nason's Two-fold Scheme 92

6.3.2 McCoy's Delete-d 93

6.3.3 Wavelet Delete-d Cross Validation 101

6.3.4 Regularised Iterative V-fold Cross Validation 106

6.4 Simulation Study 107

7 Wavelets for Irregularly-Spaced Data **109**

7.1 Review of the Literature . . . 109

7.1.1 Treating the Data as Though They Are Regularly Spaced . . . 110

7.1.2 Interpolation . . . 110

7.1.3 Exact Wavelet Coefficients . . . 111

7.1.4 Second Generation Wavelets . . . 111

7.2 Kovac's Interpolation Scheme . . . 112

7.2.1 Problem Setup . . . 113

7.2.2 Kovac's Linear Interpolation Scheme . . . 113

7.2.3 Smoothing . . . 114

7.2.4 Extensions . . . 115

7.3 Optimal Gridding . . . 116

7.3.1 Smoothing of the Irregularly-Spaced Data . . . 117

7.3.2 Simulation Studies . . . 118

8 Conclusion **123**

8.0.1 Further Work . . . 123

A Miscellaneous Background **137**

A.1 Penalised Linear Regression . . . 137

A.1.1 Standard Linear Regression . . . 138

A.1.2 Lasso . . . 138

A.1.3 Ridge Regression . . . 139

A.2 Nonparametric Regression . . . 139

A.2.1 Spline Smoothing . . . 139

A.2.2 Kernel Smoothing . . . 141

A.3 Golden Section Search . . . 142

B Exact Lifted LA8 Coefficients **145**

C WaveCUDA **149**

D Simulation Studies Supplementary Tables **161**

D.1 Wavelet Cross Validation 161

D.2 Wavelets for Irregularly-Spaced Data 164

D.2.1 Simulation Study 1: Minimising *truemse* 164

D.2.2 Simulation Study 2: Smoothing Irregular Datapoints 166

List of Figures

2.1 Discrete Wavelet Transform: Decomposition step 26

2.2 Discrete Wavelet Transform: Reconstruction step 26

2.3 Discrete Wavelet Transform: an example decomposition 27

2.4 Plots of the test signals 29

2.5 Haar filters 30

2.6 D4 filters 31

2.7 C6 filters 32

2.8 LA8 filters 33

2.9 Comparison of the DWT with Haar, D4, C6 and LA8 filters applied to the Doppler test signal 40

2.10 Comparison of the MODWT with Haar, D4, C6 and LA8 filters applied to the Doppler test signal 41

2.11 Demonstration of shift invariance 42

2.12 Comparison of the LA8 DWT applied to clean and noisy Doppler signals 43

2.13 Threshold policies 44

2.14 Comparison of different thresholding schemes 45

4.1 Comparison of GPU and CPU architectures 70

5.1 WaveCUDA plotting demonstration 83

5.2 Graph of timings for Cross Validation 86

7.1 Kovac interpolation 114

7.2 Effect of changing Kovac gridding 118

A.1 Illustration of golden section search 142

B.1 Euclidean algorithm for the LA8 wavelet filter 146

C.1 Graph of timings for the Haar DWT 158

C.2 Graph of timings for the D4 DWT 159

C.3 Graph of timings for the C6 DWT 159

C.4 Graph of timings for the LA8 DWT 160

C.5 Graph of timings for the Haar MODWT 160

List of Tables

2.1 D4 scaling coefficients 30

2.2 C6 scaling coefficients 31

2.3 LA8 scaling coefficients 32

3.1 Coiflet 6 filter lifting coefficients 62

3.2 LA8 lifting coefficients 65

4.1 A comparison of the specification of our CPUs and GPUs 70

5.1 Summary of implementation of WaveCUDA 84

6.1 Table of hat matrices for an 8×8 Haar wavelet matrix: 1 98

6.2 Table of hat matrices for an 8×8 Haar wavelet matrix: 2 99

6.3 Table of regularised hat matrices for an 8×8 Haar wavelet matrix 105

6.4 Simulation study comparing MSE of CV schemes: mean (sd) 108

7.1 Simulation study minimising *truemse*: best performing measure, no noise 119

7.2 Simulation study minimising *truemse*: mean (sd) 121

7.3 Simulation study smoothing irregular datapoints, SNR 5 122

D.1 Simulation study comparing CV schemes: lowest MSE, SNR 5 162

D.2 Simulation study comparing CV schemes: beating SURE, SNR 5 162

D.3 Simulation study comparing CV schemes: beating SURE, SNR 5 163

D.4 Simulation study comparing CV schemes: making the oracle, SNR 5 163

D.5 Simulation study minimising *truemse*: best performing measure, SNR 5 164

D.6 Simulation study minimising *truemse*: mean (sd), SNR 5 165

D.7 Simulation study smoothing irregular datapoints, SNR 10 167

Chapter 1

Introduction

Wavelets have found many applications in statistics, particularly non-parametric regression. In this thesis, we explore three main areas of wavelets: the lifting scheme, cross validation and wavelets for irregularly spaced data. We use the lifted wavelets to implement CUDA-accelerated transforms for a new R package WaveCUDA.

Our main *theoretical* contributions in this thesis are:

- We derive new lifting scheme algorithms for the Coiflet 6 (C6) and Least Asymmetric 8 (LA8) wavelet filters following the approach outlined by Daubechies & Sweldens (1998). These are algorithms that compute the standard wavelet transforms for the C6 and LA8 filters without using a moving window approach. We are interested in these lifting scheme algorithms because they can be immediately parallelisable and can be implemented in-place i.e. without needing additional memory. We also derive new standard wavelet coefficients with improved accuracy for LA8.
- We extend of McCoy & Konstantinou's (2002) cross validation (CV) scheme to general delete-d or V-fold CV using an iterative formation of the Sherman Morrison formula and regularisation. This extends the wavelet cross validation literature beyond Nason's (1996) two-fold CV, which should produce threshold estimates with reduced bias.
- We extend Kovac's (1998) treatment of wavelets for irregularly spaced data using an optimal gridding scheme as determined by an entropy-based cost measure. We extend this method as it performs well but can be improved by introducing an adaptive gridding scheme, and then we can use the full arsenal of wavelet analysis on the resulting interpolated series.

We also contribute a new R library providing CUDA-accelerated wavelet analysis, with considerable speedups for the user in wavelet transforms, wavelet threshold schemes and in an implementation of Nason's (1996) two-fold CV.

We have presented WaveCUDA at the UseR! 2014 conference in Los Angeles, and also presented a paper on Lifting The LA8 Transform at the 5th Annual International Conference on Computational Mathematics, Computational Geometry and Statistics (CMCGS 2016), at which we were awarded the best student paper award. We intend to submit a paper on our R package documentation to the Journal of Computing and Graphical Statistics (JCGS), and a paper on our new cross validation schemes to Statistics and Computing.

1.1 Structure of Thesis

This thesis is laid out with mostly background material in Chapters 2 to 4 and contributions are described in Chapters 5 to 7, although the lifting filters of Chapter 3 are an exception to the rule.

The majority of the opening chapters are dedicated to background. In *Chapter 2*, we introduce wavelets, wavelet filters, wavelet signal estimation, many of the flavours of wavelet transforms and review the statistical wavelet literature. Here, we lay the foundations for most of the rest of the thesis. *Chapter 3* explores the lifting scheme, which we will use to derive parallelisable, in-place algorithms for wavelet transforms for implementation in Chapter 5. In Chapter 3 we also give new lifting factorisations. Then, in *Chapter 4*, we delve a little into computer architecture with information on CUDA, the differences between CPU and GPU architectures, the basics of CUDA C and how we might write optimised CUDA code; if the reader wishes to focus on the development of the wavelet methodology, they may move onto subsequent chapters.

Our main contributions section starts with *Chapter 5* which covers our WaveCUDA package, an algorithmic introduction to wavelet transforms, the current implementations in WaveCUDA and some of the optimisations. Then we present timing graphs and the speedups that we achieve. We turn to cross validation in *Chapter 6*, starting with an introduction to CV for parametric regression. We then advance to wavelet CV schemes, from Nason's (1996) scheme to an extension of McCoy & Konstantinou's (2002). We conclude this chapter with a simulation study of wavelet CV schemes. In *Chapter 7* we consider wavelets for irregularly spaced data. After a thorough treatment of the literature, we then explore Kovac's (1998) scheme in more detail, and extend it to use an optimised regular grid. We show this gridding to be useful in simulation studies.

Finally, we conclude in *Chapter 8* and offer suggestions for further research. We also supply some supplementary material in the appendices, including *Appendix A*, in which we introduce some diverse background topics that we use in other chapters, from penalised linear regression to the golden section search.

1.2 Software Used

We use R (R Core Team 2015) throughout this thesis for statistical analysis and plotting. We make use of three different wavelet packages: *wmtsa* (Constantine & Percival 2013), *wavethresh* (Nason 2013*b*) and *waveslim* (Whitcher 2013). Other invaluable packages for the simulation studies are *Matrix* (Bates & Maechler 2015), *foreach* (Revolution Analytics & Weston 2014) and *doParallel* (Revolution Analytics & Weston 2015).

In WaveCUDA, we use *devtools* (Wickham, Hester & Chang 2018) for building, *roxygen2* (Wickham, Danenberg & Eugster 2018) for documenting, *ggplot2* (Wickham 2016) for plotting, *data.table* (Dowle & Srinivasan 2018) for building a coefficient table efficiently and *wavethresh* (Nason 2013*b*) for thresholding.

Our C code was compiled with *gcc* 4.8.4 (Stallman & the GCC Developer Community 2015) and the CUDA code with *nvcc* 6.5.12 (NVIDIA Corporation 2014*b*).

Chapter 2

Wavelets

In this chapter, we introduce wavelets, starting with the Continuous Wavelet Transform which is the continuous form of the wavelet transform as applied to functions. Then we define the Discrete Wavelet Transform in a signal processing framework, as this then transitions neatly to the later chapters. We introduce some of the most common wavelet filters, and this is the selection of filters that we implement in our R package. Next, we briefly explain some generalisations to the Discrete Wavelet Transform which we will refer to in later chapters. Lastly, we present a literature review of wavelets.

2.1 The Continuous Wavelet Transform

First, let us define the *Continuous Wavelet Transform* (CWT), which is applied to functions. It is analogous to the *Fourier Transform* except that it has advantages, such as having resolution in both of the time and scale domains. The *mother wavelet*, ψ, is used to segment a function; the *father wavelet*, ϕ, on the other hand is a scaling function. A function is transformed by taking its inner product with wavelet functions. In the case of the *Haar* wavelet transform, a basic wavelet transform, these are the following:

$$\psi(x) = \begin{cases} 1 & x \in [0, \frac{1}{2}), \\ -1 & x \in (\frac{1}{2}, 1], \\ 0 & \text{otherwise.} \end{cases} \quad \phi(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The detail wavelets are defined from the mother wavelet by translations and dilations:

$$\psi_{a,b}(x) = \frac{1}{\sqrt{a}} \psi \left(\frac{x - b}{a} \right),$$

for $a > 0,\ b \in \mathbb{R}$. We can choose discrete values for a and b that maintain an invertible transform.

A common choice is *critical sampling* which is the coarsest such discretisation (Vidakovic 1999). It uses dyadic translations and dilations $a = 2^{-j}, b = k2^{-j},\ j, k, \in \mathbb{Z}$,

$$\psi_{j,k}(x) = 2^{\frac{j}{2}}\psi(2^j x - k). \tag{2.2}$$

These wavelets are orthonormal, i.e. $< \psi_{j,k}, \psi_{l,m} >= \delta_{j,l}\ \delta_{k,m}$, hence they form an orthonormal basis of $L^2(\mathbb{R})$. And so, every $f \in L^2(\mathbb{R})$ can be decomposed by wavelets (Nason 2008):

$$f(x) = \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} d_{j,k}\psi_{j,k}(x), \tag{2.3}$$

where the wavelet coefficients of f, the $d_{j,k}$, are defined by

$$d_{j,k} = \int_{-\infty}^{\infty} f(x)\psi_{j,k}(x)\mathrm{d}x =< f, \psi_{j,k} > . \tag{2.4}$$

The scaling coefficients are calculated by taking the inner product with the scaling wavelets. The size of detail and scaling coefficients represents the strength of signal features at corresponding times and scales.

However, in this thesis we focus on the discrete wavelet transform which uses filters h_l and g_l instead of functions $\psi(x)$ and $\phi(x)$. These are related by the wavelet and scaling equations of Mallat (1989*b*):

$$\psi(x) = \sum_{k=-\infty}^{\infty} h_k\sqrt{2}\ \phi(2x - k),$$
$$\phi(x) = \sum_{k=-\infty}^{\infty} g_k\sqrt{2}\ \phi(2x - k).$$

Moreover, the filters h_l and g_l will have the same 'shape' as the functions $\psi(x)$ and $\phi(x)$: see for example Percival & Walden (2000) for details on the connections between wavelet filters and wavelet functions.

2.2 The Discrete Wavelet Transform

In the context of time series analysis sampled at discrete times we are interested in the *Discrete Wavelet Transform* (DWT). The cascade (now, *pyramid*) algorithm was first proposed in the context of wavelets by Mallat (1989*a*), whilst the first published reference to the name of the DWT was by Daubechies (1988). Instead of taking inner products, discrete transforms are taken using convolutions of filters with time series which we store as vectors. First, however, we need some definitions of filters. These definitions are adapted from Strang & Nguyen (1997).

Definition 2.2.1. *A* **filter** *is a linear time-invariant operator which is convolved with an input*

vector to give an output vector. For example, in (2.5) below, h is our filter, x is the input vector and y the output vector:

$$y_l = \sum_k h_k x_{l-k}, \tag{2.5}$$

where subscripts $k, l \in \mathbb{Z}$ indicate vector components.

A *filter bank* is a set of filters. In the context of wavelet transforms, we have two important filter banks: *analysis* and *synthesis* filter banks, each composed of two filters, *high-pass* h_l and *low-pass* g_l filters. A *high-pass* filter h_l is one that attenuates low frequencies and *passes* the higher frequencies, whilst a *low-pass* filter g_l picks out the low frequencies and reduces the high frequencies. In the *Haar* transform, the low-pass filter is a (scaled) moving average whilst the high-pass filter is a (scaled) moving difference. Let us now define the discrete Haar wavelet filter bank as our first example of a wavelet.

Definition 2.2.2. *The discrete Haar wavelet is defined by its filters*

$$h_l = \begin{cases} 2^{-1/2} & l = 0, \\ -2^{-1/2} & l = 1, \\ 0 & \textit{otherwise.} \end{cases} \tag{2.6}$$

$$g_l = \begin{cases} 2^{-1/2} & l = 0, \\ 2^{-1/2} & l = 1, \\ 0 & \textit{otherwise.} \end{cases} \tag{2.7}$$

Definition 2.2.3. *The discrete wavelet transform (DWT) filters a data series $\boldsymbol{y} = (y_0, \ldots, y_{n-1})$ into detail coefficients (sometimes called wavelet coefficients), $\boldsymbol{d}_j = (d_{j,0}, \ldots, d_{j,2^j-1})$ and scaling coefficients $\boldsymbol{s}_j = (s_{j,0}, \ldots, s_{j,2^j-1})$ for resolution level j. The first layer of filtering, at level $J-1$, is calculated using the following equations:*

$$\left.\begin{array}{l} s_{J-1,k} = \sum_{l=0}^{L-1} g_l y_{2k-l+1 \bmod n} \\ d_{J-1,k} = \sum_{l=0}^{L-1} h_l y_{2k-l+1 \bmod n} \end{array}\right\} k = 0, \ldots, 2^{J-1} - 1. \tag{2.8}$$

where the functions or **filters** *g_l and h_l are specific to a particular wavelet transform. g_l is known as the* **low-pass filter** *whilst h_l is the* **high-pass filter**. *Our vector $\boldsymbol{y}$ has length $n = 2^J$ for an integer J. Indeed, if n is not a power of 2, then we can make it so by appending zeros. L is the length of our filters.*

The length of a filter is analogous to the support of a function; i.e. it is the difference between the largest and smallest indices of the filter whose coefficients are non-zero; equation (2.8) is sometimes written as an infinite sum, where the filters have zero values outside their support. The filter coefficients can be termed as *responses* to *impulses* in our input vector. We hence define a *Finite Impulse Response* (FIR) filter to be a filter with finite length. Note we use

modular arithmetic in our y index: this is because we implement *periodic boundary conditions* in this thesis which assumes that the input vector is periodic. Boundary conditions are necessary for the DWT with $L > 2$, and the most common solutions are periodic or symmetric extensions to the data vector $\boldsymbol{y}$. See the work of Cohen, Daubechies & Vial (1993) for a discussion about this these and further solutions for dealing with boundary issues.

Definition 2.2.4. *The general layer of filtering is then calculated using the latest level of scaling coefficients*

$$\left.\begin{array}{l} s_{j,k} = \sum_{l=0}^{L-1} g_l s_{j+1,2k-l+1 \bmod n} \\ d_{j,k} = \sum_{l=0}^{L-1} h_l s_{j+1,2k-l+1 \bmod n} \end{array}\right\} k = 0, \ldots, 2^j - 1. \tag{2.9}$$

Remark 2.2.5. *The wavelet literature appears to be divided over the letters 'g' and 'h'. We have opted to denote the scaling (low-pass) filter by 'g' and the detail (high-pass) filter by 'h'. This is, for example, used by Percival & Walden (2000) but not by Daubechies & Sweldens (1998).*

Remark 2.2.6. *In the above definitions, we consider indices j and k. Index j represents resolution level, or scale, and lies in $0, \ldots, log_2(n) - 1$ or $0, \ldots, J-1$, whereas k is our vector index, which goes from 0 to $2^j - 1$ for resolution level j. This is consistent with C-style programming. We can alternatively perform a partial wavelet transform up to a* **primary resolution** *$J_0 \geq 0$, leaving us with 2^{J_0} finest level detail and scaling coefficients.*

Definition 2.2.7. *The full transform detail coefficient vector $\boldsymbol{d}$ is a concatenation of each level of level of detail coefficients $\boldsymbol{d}_j$. The full discrete wavelet transform, $\boldsymbol{w}$, is then a concatenation of $\boldsymbol{d}$ and $s_{0,0}$, the last scaling coefficient. So we have*

$$\begin{aligned} \boldsymbol{d} &= (d_{J-1,0}, \ldots, d_{J-1,n/2-1}, d_{J-2,0}, \ldots, d_{J-2,n/4-1}, \ldots, d_{1,0}, d_{1,1}, d_{0,0}), \qquad (2.10) \\ &= (\boldsymbol{d}_{J-1}, \boldsymbol{d}_{J-2}, \ldots, \boldsymbol{d}_1, \boldsymbol{d}_0), \\ \boldsymbol{w} &= (\boldsymbol{d}, s_{0,0}). \qquad (2.11) \end{aligned}$$

2.2.1 DWT as a Matrix Transformation

The DWT can be performed using a matrix transformation. For details of how to form the matrix $\mathcal{W}$, refer to Percival & Walden (2000). The rows of this matrix correspond to filtering operations for increasing levels of transform and shift values using filters of the form $\{h_{j,l}\}$ and $\{g_{j,l}\}$ giving

$$d_{j,k} = \sum_{l=0}^{L_j-1} h_{j,l}\, y_{2^{J-j}(k+1)-l-1 \bmod n}, \tag{2.12}$$

$$s_{j,k} = \sum_{l=0}^{L_j-1} g_{j,l}\, y_{2^{J-j}(k+1)-l-1 \bmod n}, \tag{2.13}$$

where $L_j = (2^j - 1)(L-1) + 1$ is the length of filters $\{h_{j,l}\}$ and $\{g_{j,l}\}$ (note $L_{J-1} = L$ holds). For a full wavelet transform, the first $(n-1)$ rows correspond to the detail filters $\{h_{j,l}\}$, whilst the final row produces the final solitary scaling coefficient $s_{0,0}$ with filter $\{g_{0,l}\}$. Thus, the DWT may be calculated as a matrix multiplication,

$$\boldsymbol{w} = \mathcal{W}\boldsymbol{y}. \tag{2.14}$$

Remark 2.2.8. *Whilst the DWT can be calculated by multiplication with a wavelet matrix, in practice this is never used as we have the pyramid algorithm (see Section 2.2.2) which has only $\mathcal{O}(n)$ operations rather than $\mathcal{O}(n^2)$ (Nason 2008).*

This matrix $\mathcal{W}$ is an orthogonal matrix of dimension $n \times n$, i.e. $\mathcal{W}\mathcal{W}^T = I$. Non-orthogonal wavelet transforms cannot in general be performed by a single matrix transformation. For example, the Haar wavelet matrix for transforming a vector of length 8 is the 8×8 matrix

$$\mathcal{W} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ 1/2 & 1/2 & -1/2 & -1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & -1/2 & -1/2 \\ 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}. \tag{2.15}$$

Remark 2.2.9. *The energy of an orthogonal wavelet transform is conserved by Parseval's relation.*

$$||\boldsymbol{w}||^2 = ||\boldsymbol{y}||^2. \tag{2.16}$$

This follows from orthogonality of the wavelet transformation matrix, $\mathcal{W}$:

$$||\boldsymbol{w}||^2 = \boldsymbol{w}^T\boldsymbol{w} = (\mathcal{W}\boldsymbol{y})^T(\mathcal{W}\boldsymbol{y}) = \boldsymbol{y}^T(\mathcal{W}^T\mathcal{W})\boldsymbol{y} = \boldsymbol{y}^T\boldsymbol{y} = ||\boldsymbol{y}||^2.$$

Remark 2.2.10. *In practice, we are might only compute a partial transform up to primary resolution J_0; the corresponding wavelet matrix would then have the first $n - 2^{J_0}$ rows containing sequential shifts of detail filters $\{h_{J-1,l}\}$ to $\{h_{J_0,l}\}$ and the final 2^{J_0} rows with shifts of $\{g_{J_0,l}\}$. This matrix would still be orthogonal.*

Given an orthogonal wavelet matrix, it is straightforward to define an *inverse* wavelet transform or reconstruction:

$$\boldsymbol{w} = \mathcal{W}\boldsymbol{y} \Leftrightarrow \boldsymbol{y} = \mathcal{W}^T\boldsymbol{w} \quad \text{for an orthogonal wavelet matrix } \mathcal{W}.$$

In fact, as each separate layer of filtering given by (2.8) and (2.9) is just linear filtering, it can

be performed with a matrix (say $\mathcal{W}_j$) that combines all the detail and scaling coefficients in one vector of length 2^j. Each of these matrices $\mathcal{W}_j$ is also orthogonal (Percival & Walden 2000), and so we can recover the coefficients of level j from those of level $j-1$ using filters $\tilde{h}_l$ and $\tilde{g}_l$, which are simply reordered versions of h_l and g_l.

2.2.2 The Pyramid Algorithm

The DWT is generally implemented using a *pyramid algorithm* (Mallat 1989*a*) which derives its name from the dyadic decimation of coefficients during construction. Even DWTs constructed using the lifting scheme are still computed using a pyramid algorithm. The DWT can be carried out in-place because we need only keep the final scaling coefficient; the others can be re-written with updates to the detail and scaling coefficients. We illustrate the pyramid algorithm for an unnormalised Haar DWT in Figures 2.1 - 2.3: Figure 2.1 shows the downsampling or decimation of wavelet decomposition steps; Figure 2.2 shows the upsampling in reconstruction; Figure 2.3 illustrates the unnormalised decomposition of a vector $\boldsymbol{y} = (0, 0, 5, 4, 8, 6, 7, 3)$.

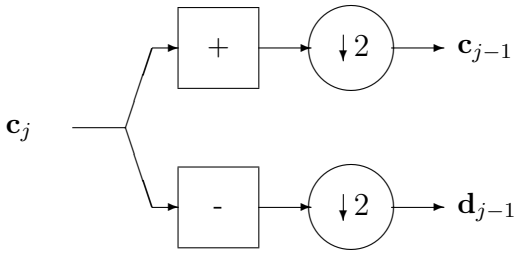


Figure 2.1: Discrete Wavelet Transform (unnormalised Haar): Decomposition step

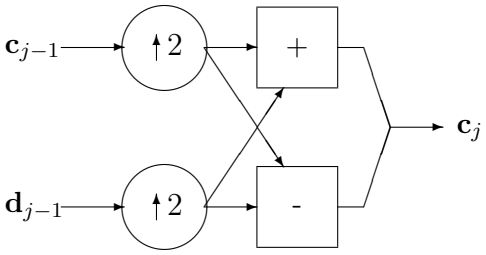


Figure 2.2: Discrete Wavelet Transform (unnormalised Haar): Reconstruction step

2.2.3 DWT Filter Properties

A wavelet filter h_l is defined $\forall\, l \in \mathbb{Z}$ but 0 outside of the range $0, \ldots, L-1$. This L (an even integer) is the width of the filter. The filter h_l must satisfy three basic properties (Daubechies 1988):

$$\sum_{l=0}^{L-1} h_l = 0, \tag{2.17a}$$

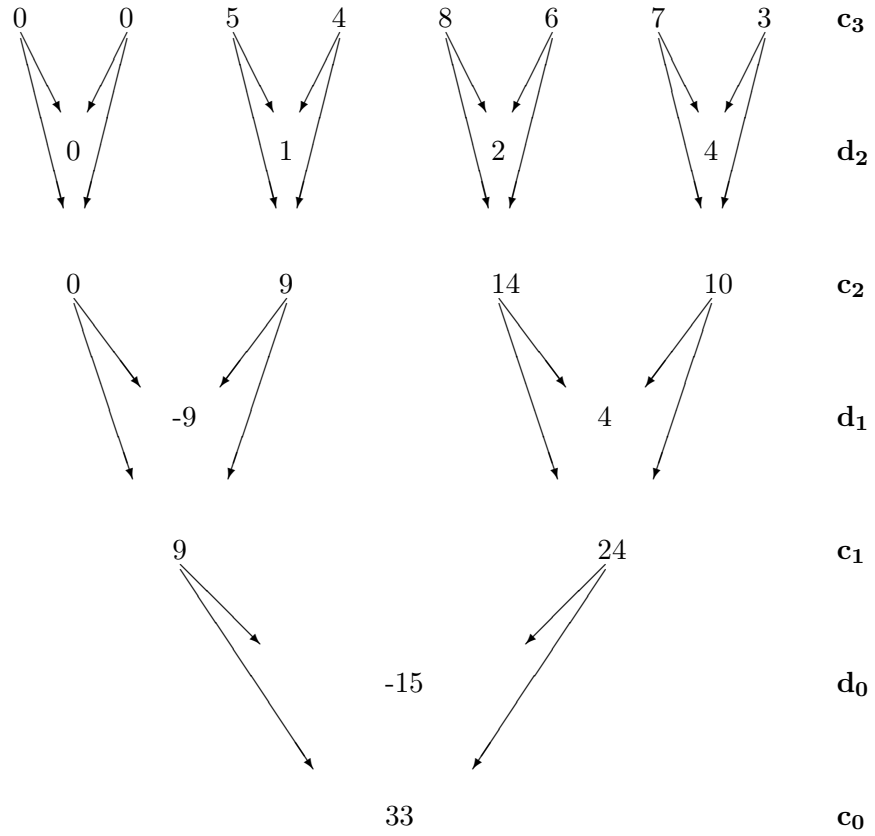


Figure 2.3: Discrete Wavelet Transform: an example decomposition using an unnormalised Haar transform

$$\sum_{l=0}^{L-1} h_l^2 = 1, \tag{2.17b}$$

$$\sum_{l=0}^{L-1} h_l h_{l+2m} = \sum_{l=-\infty}^{\infty} h_l h_{l+2m} = 0, \tag{2.17c}$$

for all non-zero integers m. The latter two equations are known together to be the *orthogonality property* of wavelet filters. Whilst Daubechies (1988) was the first to formalise these properties, they followed from the work on multiresolution analysis by Meyer (1986), Mallat (1989*a*) and Mallat (1989*b*).

Remark 2.2.11. *We do not focus on the multiresolution analysis (MRA) in this thesis. It is a sequence of closed subspaces of which our scaling functions from each resolution level form bases. The MRA is also used to denote a multiresolution approximation of a discrete signal. For good introductions to MRA, see Vidakovic (1998) for the continuous form and Percival & Walden (2000) for the discrete form.*

Once we have our DWT wavelet filter h_l, we can define the corresponding scaling filter g_l (Percival & Walden 2000):

$$g_l \equiv (-1)^{l+1} h_{L-1-l}, \tag{2.18}$$

One of the key properties of wavelet filters is the number of *vanishing moments*, which is defined below.

Definition 2.2.12. *A function $\psi(x) \in L^2(\mathbb{R})$ has m vanishing moments if, for $l = 0, \ldots, m-1$,*

$$\int x^l \psi(x) \mathrm{d}x = 0$$

Possessing vanishing moments of wavelet filters is an important property that creates zeros in a wavelet decomposition. This ensures we have sparsity in the wavelet coefficients.

Remark 2.2.13. *We introduce the notion of* **phase** *of a filter which corresponds to the time alignment of filtered and non-filtered values. Ideally, we would like to have a* **zero phase** *filter where the time alignment matches exactly, but a* **linear phase** *is also acceptable as the alignment can be corrected with a linear shift.*

2.3 Wavelet test signals

We will re-visit four wavelet test signals throughout this thesis, of which the first three were introduced to the wavelet signal estimation literature in Donoho & Johnstone's (1994) seminal paper, and the fourth is a simple smooth test function. They are all defined on $[0, 1]$.

(i) Doppler: The doppler signal starts with very fast frequency that decays over the signal, making it difficult to model outside of a time-scale transform. It has the functional form

$$f(t) = [t(1-t)]^{1/2} \sin\left(2\pi \frac{1+\epsilon}{t+\epsilon}\right), \quad \epsilon = 0.05.$$

(ii) Heavisine: The heavisine signal is made from a sine wave with two added discontinuities.

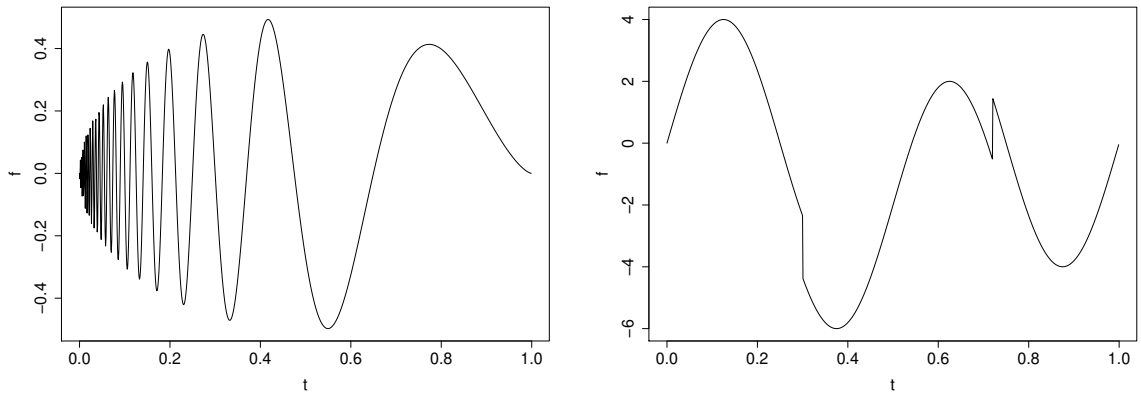
$$f(t) = 4\sin(4\pi t) - \mathrm{sign}(t - 0.3) - \mathrm{sign}(0.72 - t).$$

(iii) Blocks: The blocks signal is a step function with steps at locations that are mostly non-dyadic.

$$f(t) = \sum_{j=0}^{10} h_j K(t - t_j), \quad K(t) = (1 + \mathrm{sign}(t)/2),$$

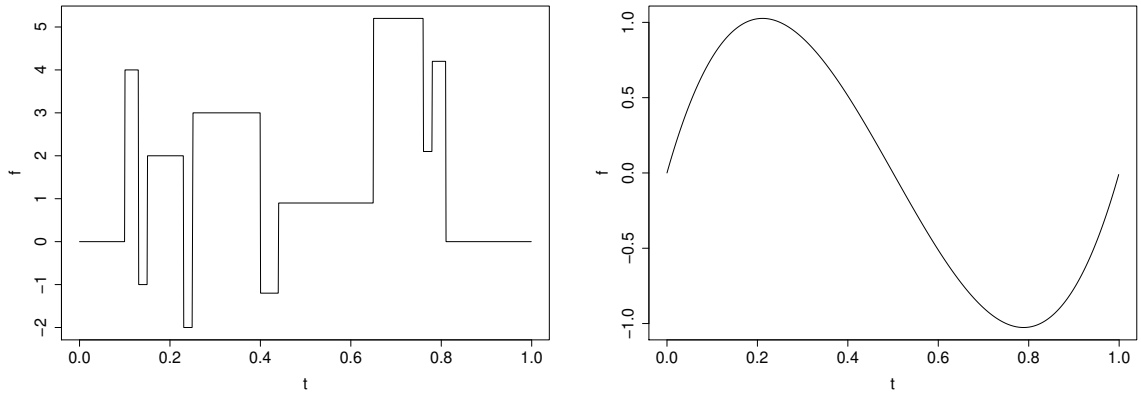
where

$$(t_j) = (0.1, 0.13, 0.15, 0.23, 0.25, 0.40, 0.44, 0.65, 0.76, 0.78, 0.81),$$
$$(h_j) = (4, -5, 3, -4, 5, -4.2, 2.1, 4.3, -3.1, 2.1, -4.2).$$



(a) Doppler

(b) Heavisine



(c) Blocks

(d) Cubic

Figure 2.4: Plots of the test signals Doppler, Heavisine, Blocks and Cubic

(iv) Cubic: This is a simple cubic function.

$$f(t) = 64t(t-1)(t-0.5)/3.$$

We plot these signals in Figures 2.4a to 2.4d.

2.4 Wavelet filters

Different wavelet filters are used for their different properties such as vanishing moments, symmetry and smoothness. We present four different wavelet filters here, the Haar, D4, C6 and LA 8 wavelets which are somewhat ubiquitous in the wavelet literature. The shorter filters, Haar and D4, give wavelet coefficients with good time resolution whilst the longer filters, C6 and LA8, have improved frequency resolution and additional smoothness properties. This is a small subset of the possible wavelet filters we might have chosen.

2.4.1 Haar

We have already given the scaling and detail filters for the Haar transform in (2.7) and (2.6) respectively. As the Haar filter only has length two, it has good time resolution but poor frequency resolution. Useful consequences of this are that the Haar transform is quickest to compute and it is immediately parallelisable (see Section 5.1.1). Nason (2013*a*) uses Haar wavelets for his stationarity test. However, because the Haar wavelet filter is not smooth, it is not generally used for reconstruction. The filters are plotted in Figure 2.5.

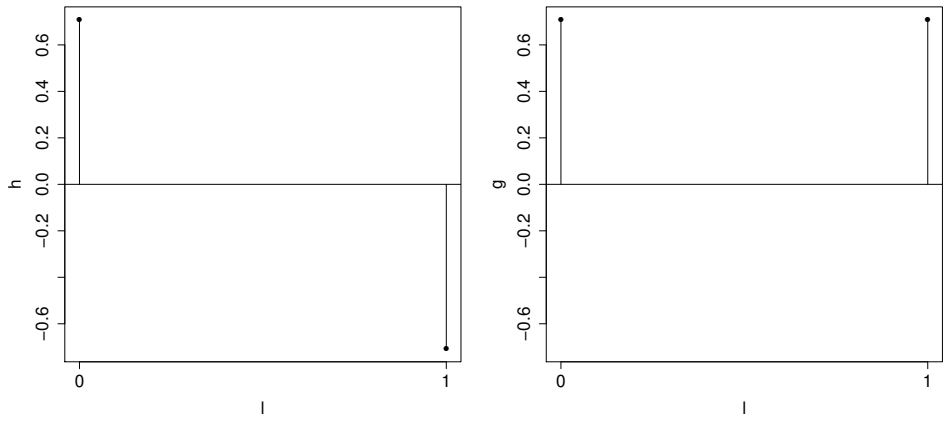


Figure 2.5: Haar detail (left) and scaling filters (right)

2.4.2 Daubechies 4

The Daubechies "Extremal Phase" 4 (D4) wavelet of length four was constructed by Daubechies to be orthonormal and with two vanishing moments (Daubechies 1988). However, it is not ideal for reconstruction as it often produces a "shark fin" artefact (Percival & Walden 2000). The D4 scaling coefficients are given in Table 2.1 and the filters are plotted in Figure 2.6. Daubechies (1988) derived extremal phase wavelet filters for $N = 2, \ldots, 10$, where $N = L/2$ (half the filter length) and represents the number of vanishing moments. Note that whilst we identify this filter using L for consistency with our other filters, often N is used in the literature.

Coefficient	Exact value	Approximate value
c_0	$(1+\sqrt{3})/(4\sqrt{2})$	0.4829629131445341
c_1	$(3+\sqrt{3})/(4\sqrt{2})$	0.8365163037378079
c_2	$(3-\sqrt{3})/(4\sqrt{2})$	0.2241438680420134
c_3	$(1-\sqrt{3})/(4\sqrt{2})$	-0.1294095225512604

Table 2.1: Daubechies 4 scaling coefficients, i.e. $g_l = c_l, l = 0, \ldots, 3$

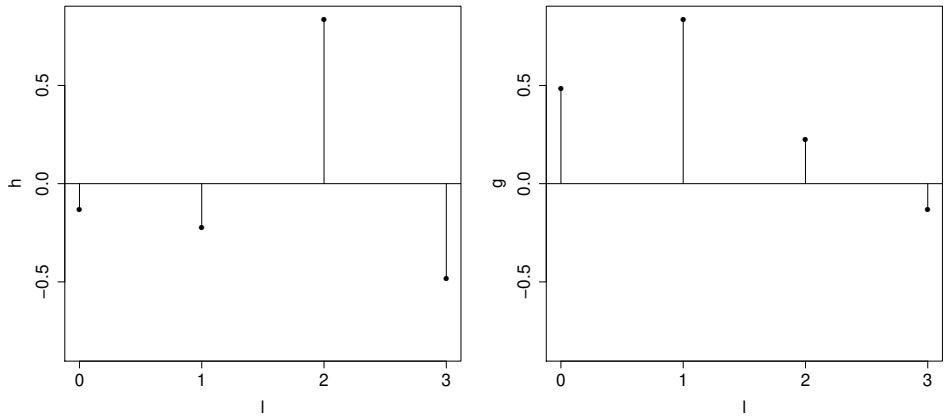


Figure 2.6: Daubechies 4 detail (left) and scaling (right) filters

2.4.3 Coiflet 6

The Coiflet wavelets were constructed by Daubechies after a suggestion from Coifman to have a filter with vanishing moments in the scaling as well as the detail filter (Daubechies 1992). The resulting filters also happen to have very good phase properties as they give good approximations to zero phase filters. The C6 wavelet has filters of length six with two vanishing moments. It is considered to be a good compromise between time and frequency resolution, as well as having the property of being approximately a zero phase filter. However, much like the D4, it can produce artefacts from its triangular shape (Percival & Walden 2000). Černá, Finěk & Najzar (2008) derived exact coefficients of the C6 wavelet, which we present in Table 2.2. We plot the wavelet filters in Figure 2.7.

Coefficient	Exact value	Approximate value
c_0	$\sqrt{2}(1-\sqrt{7})/32$	-7.273261951252645e-2
c_1	$\sqrt{2}(5+\sqrt{7})/32$	3.378976624574818e-1
c_2	$\sqrt{2}(7+\sqrt{7})/16$	8.525720202116004e-1
c_3	$\sqrt{2}(7-\sqrt{7})/16$	3.848648468648577e-1
c_4	$\sqrt{2}(1-\sqrt{7})/32$	-7.273261951252645e-2
c_5	$\sqrt{2}(-3+\sqrt{7})/32$	-1.565572813579199e-2

Table 2.2: Coiflet 6 scaling coefficients, i.e. $g_l = c_l, l = 0, \ldots, 5$

2.4.4 Least Asymmetric 8

The Least Asymmetric wavelets (also called symmlets) were constructed by Daubechies (1992, 1993) to have near linear phase whilst retaining orthonormality and compact support, which maximises their symmetry given the constraints (Percival & Walden 2000). The LA8 filter has four vanishing moments. It is frequently used because it strikes a good balance between time

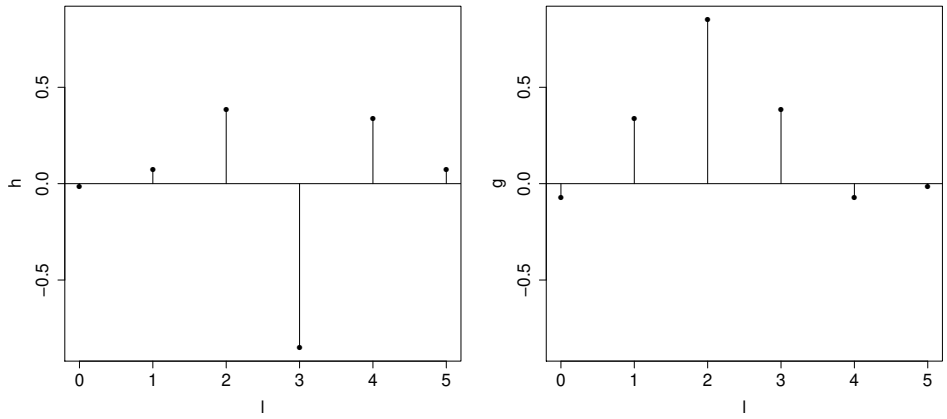


Figure 2.7: Coiflet 6 detail (left) and scaling (right) filters

and frequency resolution, and has a smooth form - as well as the other given properties. Whilst exact forms of the LA8 coefficients are not available, we have re-derived them using variable precision using the procedures outlined by Percival & Walden (2000). Our coefficients differ from theirs in the final decimal places due to the increased precision that we use.

Coefficient	Approximate value
c_0	−7.576571478950221e−2
c_1	−2.963552764600249e−2
c_2	4.976186676327750e−1
c_3	8.037387518051321e−1
c_4	2.978577956053061e−1
c_5	−9.921954357663353e−2
c_6	−1.260396726203130e−2
c_7	3.222310060405147e−2

Table 2.3: Least Asymmetric 8 scaling coefficients, i.e. $g_l = c_l, l = 0, \ldots, 7$

2.4.5 Comparison of DWT for Different Filters

In this section, we briefly compare the outputs of applying the DWT to the Doppler signal using each of the filters Haar, D4, C6 and LA8 in turn, plotted in Figure 2.9 on page 40. In these plots we see *Translate* on the x axis plotted against (detail) wavelet coefficient magnitude on the y axis for each *Level*. Here, the translate represents time and provides time alignment for the wavelet coefficients. The original signal was a vector of length 1024: thus the first level of transform, $\mathbf{d}_9$, has length 512, $\mathbf{d}_8$ has length 256 etc up to $\mathbf{d}_0$ which is a single value. These have been plotted with WaveThresh (Nason 2013*b*), which gives the option of rescaling each level of detail coefficients separately to enable us to see the transformation throughout all layers: in practice, the scaling is not the same across all layers.

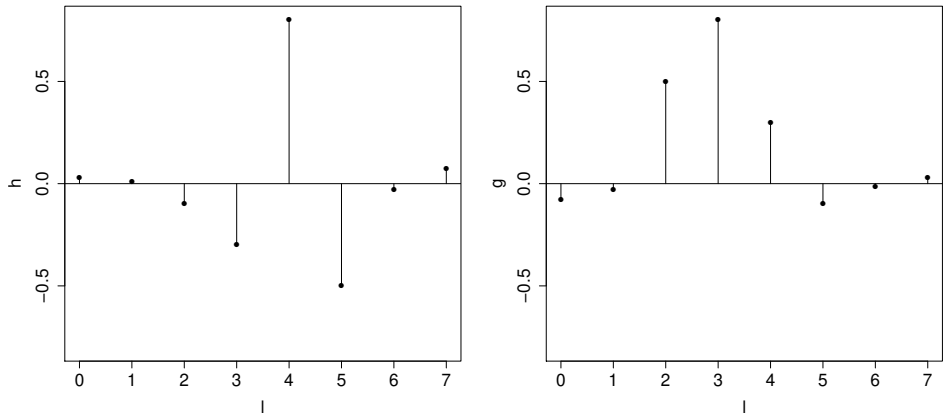


Figure 2.8: Least Asymmetric 8 detail (left) and scaling (right) filters

One can clearly see the effect of increasing vanishing moments between the different plots. As the Haar transform has only one vanishing moment, it has the most non-zero coefficients. LA8 with four vanishing moments has the fewest non-zero coefficients. D4 and C6 both have two vanishing moments. Furthermore, we can see that the Haar transform captures the most high-frequency behaviour at the start of the Doppler signal.

2.5 MODWT

The Maximum Overlap Discrete Wavelet Transform (MODWT) is a highly redundant and non-orthogonal wavelet transform that differs from the DWT in that it retains both odd and even decimations. For a good overview of the MODWT, see Nason (2008), whereas for a more thorough treatment refer to Percival & Walden (2000). The MODWT uses circularly shifted filters at each level so that each level of the transform has n wavelet coefficients, and hence the i^{th} wavelet coefficient of any level corresponds to the i^{th} element of our input vector. The main advantage of the MODWT is that is retains time resolution in all wavelet scales, as can be seen in Figure 2.10.

The MODWT can be either packet-ordered or time-ordered, both of which we have implemented in WaveCUDA. In the packet-ordered transform, each shift corresponds to a separate packet - which is like a single stage of a DWT transform. There are 2 packets in the first level of coefficients, 4 in the second, 8 in the third etc. . On the other hand, the time-ordered transform stores the detail and scaling filters in time order; this is implemented with a moving window filter.

Percival & Walden (2000) define the jth level time-ordered MODWT wavelet detail and scaling

coefficients using filters $\tilde{h}_{j,l} = h_{j,l}/2^{(J-j)/2}$ and $\tilde{g}_{j,l} = g_{j,l}/2^{(J-j)/2}$,

$$\tilde{d}_{j,k} = \sum_{l=0}^{L_j-1} \tilde{h}_{j,l}\, y_{k-l \bmod n}, \tag{2.19}$$

$$\tilde{s}_{j,k} = \sum_{l=0}^{L_j-1} \tilde{g}_{j,l}\, y_{k-l \bmod n}, \tag{2.20}$$

where $L_j = (2^j - 1)(L-1) + 1$ as before. In our software we do not apply this scaling, but this is critical for decomposing the energy of the signal:

$$||\boldsymbol{y}||^2 = \sum_{j=J_0}^{J-1} ||\tilde{\boldsymbol{d}}_j||^2 + ||\tilde{\boldsymbol{s}}_{J_0}||^2.$$

Because of the redundancy of the MODWT, it has no unique inverse: in fact there are infinitely many possible reconstructions (Percival & Walden 2000). If we consider just the first layer of coefficients in the transform, this is created from two packets: so we could reconstruct the original data from either packet or any linear combination of the two whose weights sum to one. We calculate our reconstructions in the standard way as Coifman & Donoho (1995) coined "cycle-spinning", which is to average the reconstructions from each packet.

2.5.1 Comparison of MODWT for Different Filters

We plot the MODWT for the Doppler signal now using our four chosen wavelet filters in Figure 2.10 on page 41. If we compare this to the DWT in Figure 2.9, our observations relating to vanishing moments are still relevant. However, we now have increased resolution in all levels of the transformation. Notice that the *Translate* range on the x axis is now up to 1024, because we retain both odd and even decimations. Another observation we can make with the MODWT transforms is that we can clearly see the effect of periodic boundary conditions in the lowest levels of the transform, especially for the longer filters.

2.5.2 Shift invariance of the MODWT

One important property of the MODWT compared to the DWT is the concept of shift invariance: that is to say we can cyclically shift an input vector, and this will simply shift the transform at each level. On the other hand, the DWT is not shift invariant, so shifting the input vector will give us an entirely different transform. We illustrate this using the Heavisine signal, as the discontinuities make it clear to spot shifts in the wavelet coefficients. See Figure 2.11 on page 42 where we demonstrate this with both the D4 filter, as that shows clear changes of sign for the DWT coefficients of the shifted signal. The shift is imperceptible in the

MODWT transform in these plots. If we were to use the Haar filter, it would not even pick up the discontinuities for one of the shifts.

2.6 DWPT

The Discrete Wavelet Packet Transform (DWPT) successively filters both the detail and scaling coefficients to create a *wavelet packet library* unlike the standard DWT which successively filters just the scaling coefficients. See Nason (2008) for a brief discussion or Percival & Walden (2000) for a more thorough treatment. Coifman & Wickerhauser (1992) used this library to create a *best basis* which is a partition of the library to a particular orthonormal transform that is *optimal* according to some specific cost function. One particular choice of cost function is the *Shannon entropy*, which is defined as follows on a given set of basis coefficients $\{v_i\}$ (Coifman & Wickerhauser 1992):

$$-\sum |v_i|^2 \log |v_i|^2 \tag{2.21}$$

2.7 Biorthogonal Wavelets

Initially, the wavelet filters that were developed were *orthogonal*, which means that, in the case of the discrete transform, the same pairs of filters performed the forward and backward transforms. All of the wavelet transforms defined already have been examples of orthogonal wavelet transforms. Cohen, Daubechies & Feauveau (1992) introduced a generalisation; they came up with the idea of the *biorthogonal* wavelet transform, where different pairs of filters perform the forward and backward transforms. Each pair of filters is mutually orthogonal but not orthogonal to the filters of the other pair. These transforms offer more flexibility, but lose the feature of conserving Parseval's identity (2.16) (Vidakovic 1999). In contrast to the DWT, this means that the energy of a transformed vector is no longer necessarily the same as that of the original vector.

2.8 Thresholding

Wavelet thresholding has been shown to be an effective smoothing technique due to the orthogonality of the wavelet transform and the sparsity of the detail coefficients (Nason 2008). Let us assume that we observe a noisy version of the true signal f with additive noise component $\boldsymbol{\xi}$. We consider here this noise to be Gaussian i.i.d. white noise, as is common in the wavelet literature. Note that wavelet thresholding has been extended to correlated noise, for example, by Johnstone & Silverman (1997). Thus, we have the following setup:

$$y_i = f(t_i) + \xi_i, \quad \xi_i \sim N(0, \sigma^2), \quad i = 0, \ldots, n-1, \tag{2.22}$$

where $\boldsymbol{y} = (y_0, \ldots, y_{n-1})$ is our noisy signal, $\boldsymbol{t} = (t_0, \ldots, t_{n-1})$ is a vector with our time observations and $\boldsymbol{\xi} = (\xi_0, \ldots, \xi_{n-1})$ is Gaussian i.i.d. white noise.

After taking the wavelet transform, we can write a similar equation for the wavelet coefficients using orthogonality of the transform:

$$\boldsymbol{w}^* = \boldsymbol{w} + \boldsymbol{\epsilon},$$

where $\boldsymbol{w}^*$ and $\boldsymbol{w}$ are wavelet coefficient vectors (as defined in Equation (2.11) on page 24) of the noisy signal and true signal respectively and $\boldsymbol{\epsilon}$ is our wavelet transformed noise vector, i.e. $\boldsymbol{w}^* = \mathcal{W}\boldsymbol{y}$, $\boldsymbol{w} = \mathcal{W}f(\boldsymbol{t})$ and $\boldsymbol{\epsilon} = \mathcal{W}\boldsymbol{\xi}$. Seeing as the original noise vector $\boldsymbol{\xi}$ was zero mean, the last component of $\boldsymbol{\epsilon}$ corresponding to the wavelet scaling coefficient is assumed to be zero. Thus, we can write

$$\boldsymbol{d}^* = \boldsymbol{d} + \boldsymbol{\epsilon}_d,$$

for detail vectors $\boldsymbol{d}^*$ and $\boldsymbol{d}$ defined as in Equation (2.10) on page 24 and $\boldsymbol{\epsilon}_d$ being the subvector of $\boldsymbol{\epsilon}$ corresponding to the wavelet detail coefficients.

Now we can revisit the statement above stating that wavelets are effective for smoothing. If we have a smooth signal to begin with, then the detail coefficients will be sparse; then due to the orthogonality of the transform, the energy of the signal will be conserved in few coefficients. Moreover, the orthogonality also ensures that the transformed noise is still Gaussian i.i.d white noise with variance $\mathcal{W}\sigma^2\mathcal{W}^T = \sigma^2 I$.

$$\operatorname{cov}[\boldsymbol{\epsilon}] = \mathcal{W}\sigma^2\mathcal{W}^T = \sigma^2 I = \operatorname{cov}[\boldsymbol{\xi}], \tag{2.23}$$

Hence, small values in the detail coefficients are likely to be noise, whilst larger values are likely to represent the true signal plus noise.

We illustrate this in Figure 2.12 which shows a clean Doppler signal alongside one with added normal noise. Then below, we see the corresponding wavelet transforms of both signals. We can see here that the additive noisy mainly affects the wavelet coefficients in the higher levels of the transformation: the lower levels appear unaffected. We can clearly see that if we were to threshold the coefficients in the lowest levels then we can remove all the noise and only lose the highest frequency information.

2.8.1 Thresholding Policies

Donoho & Johnstone (1994) proposed thresholding these detail coefficients to approximate the true signal. They suggested *hard* and *soft* thresholding policies, which are, respectively:

$$\hat{d} = \eta_H(d^*, \lambda) = d^* \mathbb{I}_{|d^*|>\lambda}, \tag{2.24}$$

$$\hat{d} = \eta_S(d^*, \lambda) = sgn(d^*)(|d^*| - \lambda)\mathbb{I}_{|d^*|>\lambda}. \tag{2.25}$$

Here, $\hat{d}$ is the estimate of the noisy detail coefficient d^* and λ is our threshold.

Remark 2.8.1. *Soft thresholding (which we saw in Section A.1.2) is* **shrinkage** *which shifts all detail coefficients towards zero by a constant amount λ. On the other hand, hard thresholding kills all detail coefficients below this level λ. Selection of this λ is a bias/variance trade-off: pick too high a value and we will* **oversmooth***, reducing variance but introducing bias; pick too low a value and we* **undersmooth***, increasing variance but with less bias. Furthermore, hard thresholding creates more variance than soft thresholding, which has increased bias.*

Gao & Bruce (1997) came up with the *firm* or *semi-soft* threshold which is designed to strike a compromise between the hard and soft thresholds by acting like the soft threshold for small values of $|d^*|$ and like the hard threshold for large values. It is defined to be

$$\eta_F(d^*, \lambda_1, \lambda_2) = \begin{cases} 0 & |d^*| \leq \lambda_1, \\ sgn(d^*)\frac{\lambda_2(|d^*|-\lambda_1)}{\lambda_2-\lambda_1} & \lambda_1 < |d^*| < \lambda_2, \\ d^* & |d^*| \geq \lambda_2. \end{cases}$$

This is a generalisation of "mid" thresholding, which has $\lambda_2 = 2\lambda_1$ and hence only has one free parameter (Percival & Walden 2000).

The policies of hard, soft and mid thresholding are plotted in Figure 2.13.

We measure the error of our estimate by applying the inverse transformation to get $\hat{\boldsymbol{y}}$ and considering the *Integrated Square Error* (ISE):

$$\hat{M} = n^{-1} \sum_{i=1}^{n} [\hat{y}_i - f(t_i)]^2 , \tag{2.26}$$

Since we are unlikely to know the true f, we often consider the Mean ISE, or MISE, $\mathrm{E}\,\hat{M}$. By Parseval's relation (2.16), it is clear that minimising the ISE is equivalent to minimising the error in the detail coefficients. The contribution to the error (i.e. the risk) from each detail coefficient is given by

$$M(d, \hat{d}) = E[(d - \hat{d})^2] = \begin{cases} \sigma^2 & \text{if } |d^*| > \lambda, \\ d^2 & \text{if } |d^*| \leq \lambda, \end{cases} \tag{2.27}$$

in the case of hard thresholding. Then, to minimise the ISE, we want to choose the minimum out of σ^2 and d^2 for each detail coefficient.

2.8.2 Levels for Thresholding

We use the term *primary resolution* to denote the coarsest resolution level at which we apply thresholding. Successful thresholding can be very sensitive to the choice of primary resolution

(Nason 2008). Johnstone & Silverman (1997) found that level-dependent thresholding was an effective way of denoising a signal with a correlated noise process. In WaveCUDA, we allow the user to select which levels they would like to threshold, which gives them control of primary resolution and enables them to perform level-dependent thresholding.

2.8.3 Thresholding Schemes

We now outline a few common thresholding schemes.

- *Universal Threshold*: this was proposed by Donoho & Johnstone (1994) in their *VisuShrink* scheme. It is designed to be visually advantageous in that it removes all of the noise asymptotically[1]. Moreover, it is asymptotically optimal: the threshold converges to the *oracle* threshold. The universal threshold is $\lambda^{univ} = \sigma(2\log n)^{1/2}$, where this σ is usually estimated robustly using the Median Absolute Deviation (MAD) of the finest-scale detail coefficients. The universal threshold is useful in that it is very quick to compute and does effectively remove noise from a signal, but it oversmooths and hence does not give a low ISE in reconstruction compared to these other methods. Nevertheless, it is used ubiquitously as a benchmark for any new thresholding schemes.

- *SURE*: this is *Stein's Unbiased Risk Estimation*, also developed by Donoho & Johnstone (1995). It uses the SURE technique to calculate unbiased level-dependent thresholds. Whilst *VisuShrink* aims to remove all noise and hence often oversmooths, *SureShrink* is concerned with minimising the error of smoothing. Let

$$\mathrm{SURE}(\boldsymbol{d^*}, \lambda) = K - 2\sum_{i=1}^{K} \mathbb{I}_{(|d_i^*| \leq \lambda)} + \sum_{i=1}^{K} \left(\min(|d_i^*|, \lambda)\right)^2,$$

  with K denoting the number of detail coefficients, then

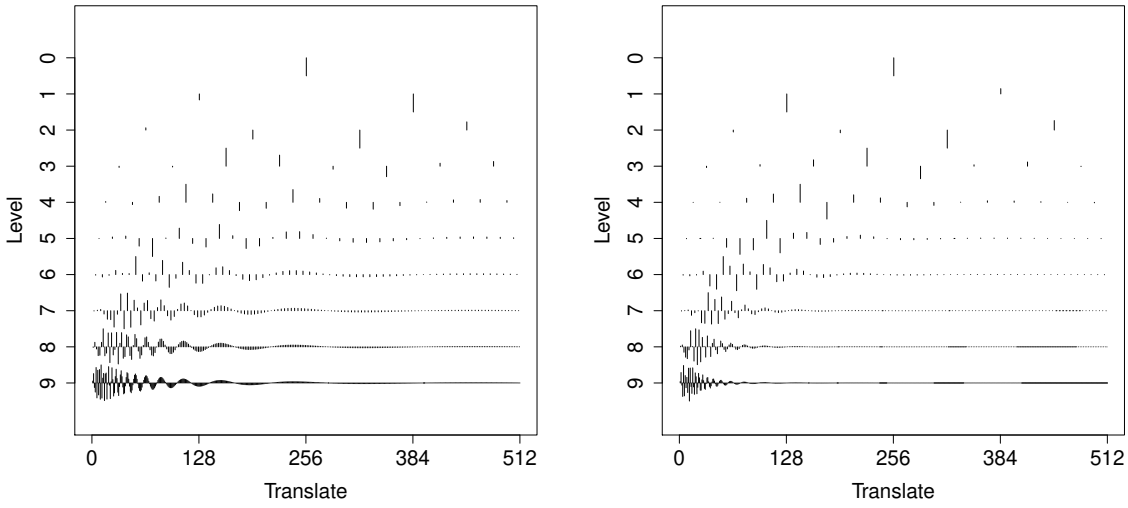
$$\lambda^{sure} = \underset{0 \leq \lambda \leq \lambda^{univ}}{\arg\min}\ \mathrm{SURE}(\boldsymbol{d^*}, \lambda).$$

  The SURE threshold is more computationally intensive to calculate than the universal threshold, but as it is designed to minimise error, its ISE is generally lower. We may still perceive some noise in the reconstruction.

- *Cross-validation*: the techniques of cross validation have been extended to wavelet thresholding. This thresholding scheme often gives the lowest errors, however at the largest computational cost. Indeed, with existing implementations this cost may make this infeasible, however our fast implementation may change this.

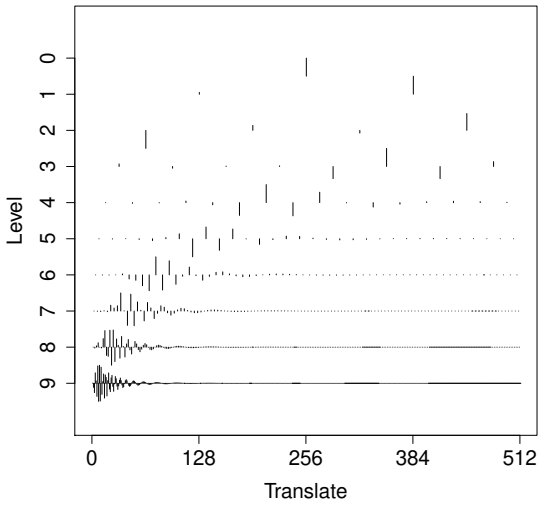
[1] In fact, for Gaussian white noise it removes all the noise with the probability $\mathbb{P}\left(\max_i |z_i| > (2\log n)^{1/2}\right)$ where z_i are realisations of a $N(0,1)$ distribution

We compare these different thresholding schemes in Figure 2.14. In each of the plots we use a noisy Doppler signal with SNR (Signal to Noise Ratio) of 10 (this is the noisy Doppler signal in Figure 2.12). We take the DWT of the signal using the LA8 filter and apply the different thresholding schemes in turn. In the left hand plots, we have annotated the DWT with the thresholds using red segments: anything within the red segment is thresholded using a soft threshold. Note that in Figure 2.14 we have altered the scaling of the wavelet coefficients from Figure 2.12: here, it is clearer to see the wavelet coefficients and thresholds without rescaling, whereas before it was clearer to see the features in the transformation by rescaling by level. In the right hand plots, we see the reconstruction of the noisy Doppler signal after thresholding. Notice how the universal threshold gives a reconstruction with no perceptible noise, however if we wanted to minimise the error in the reconstructed signal then this threshold might be considered too heavy handed. The SURE and CV thresholding schemes retain some of the noise but with much lower error.

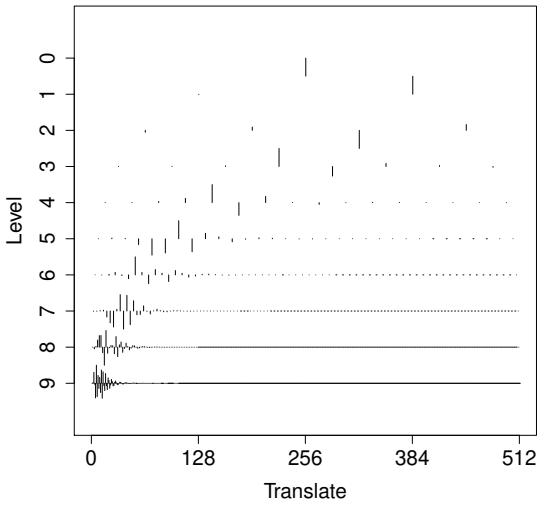


(a) Haar filter

(b) D4 filter

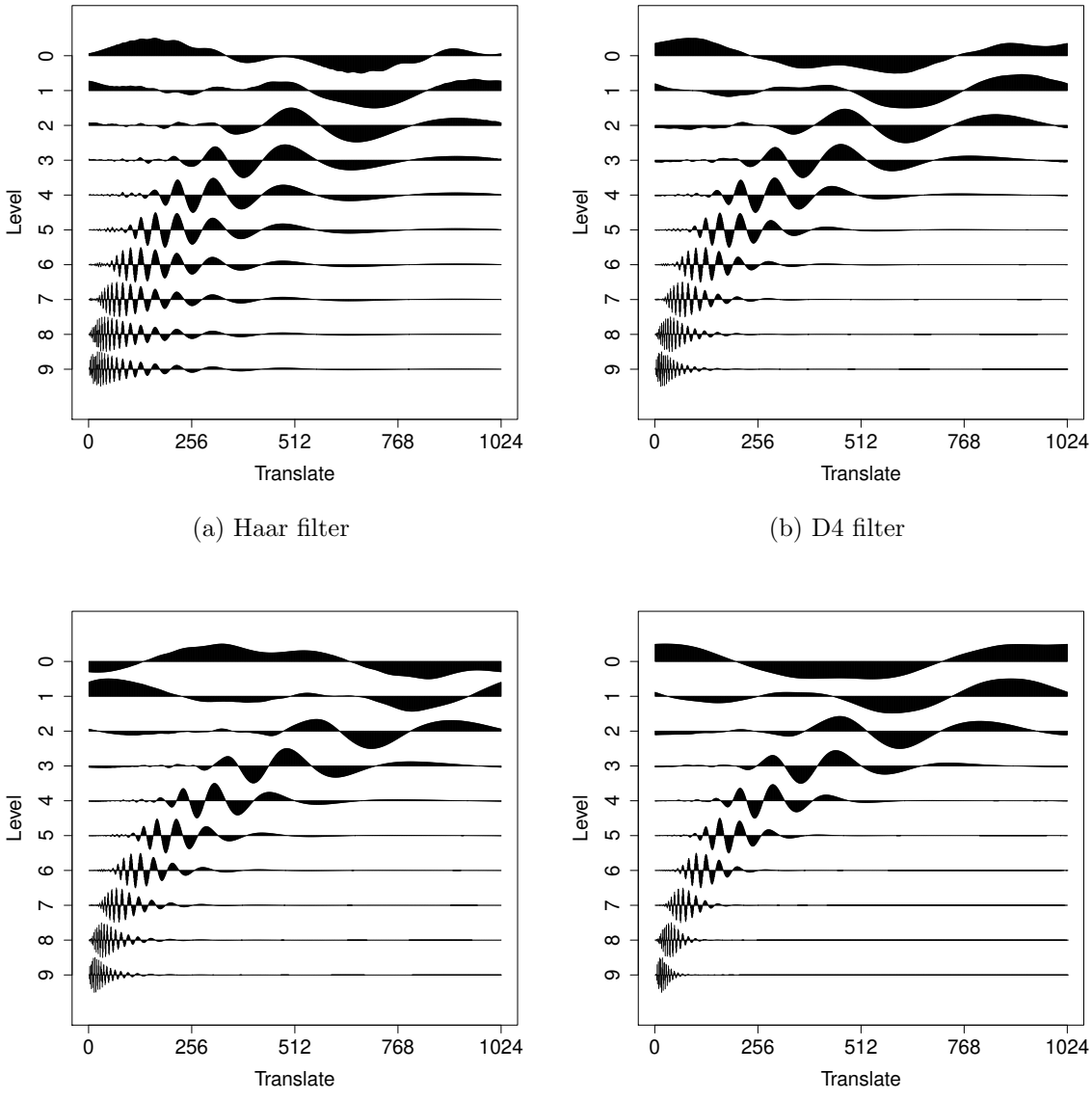


(c) C6 filter



(d) LA8 filter

Figure 2.9: Comparison of the DWT with Haar, D4, C6 and LA8 filters applied to the Doppler test signal



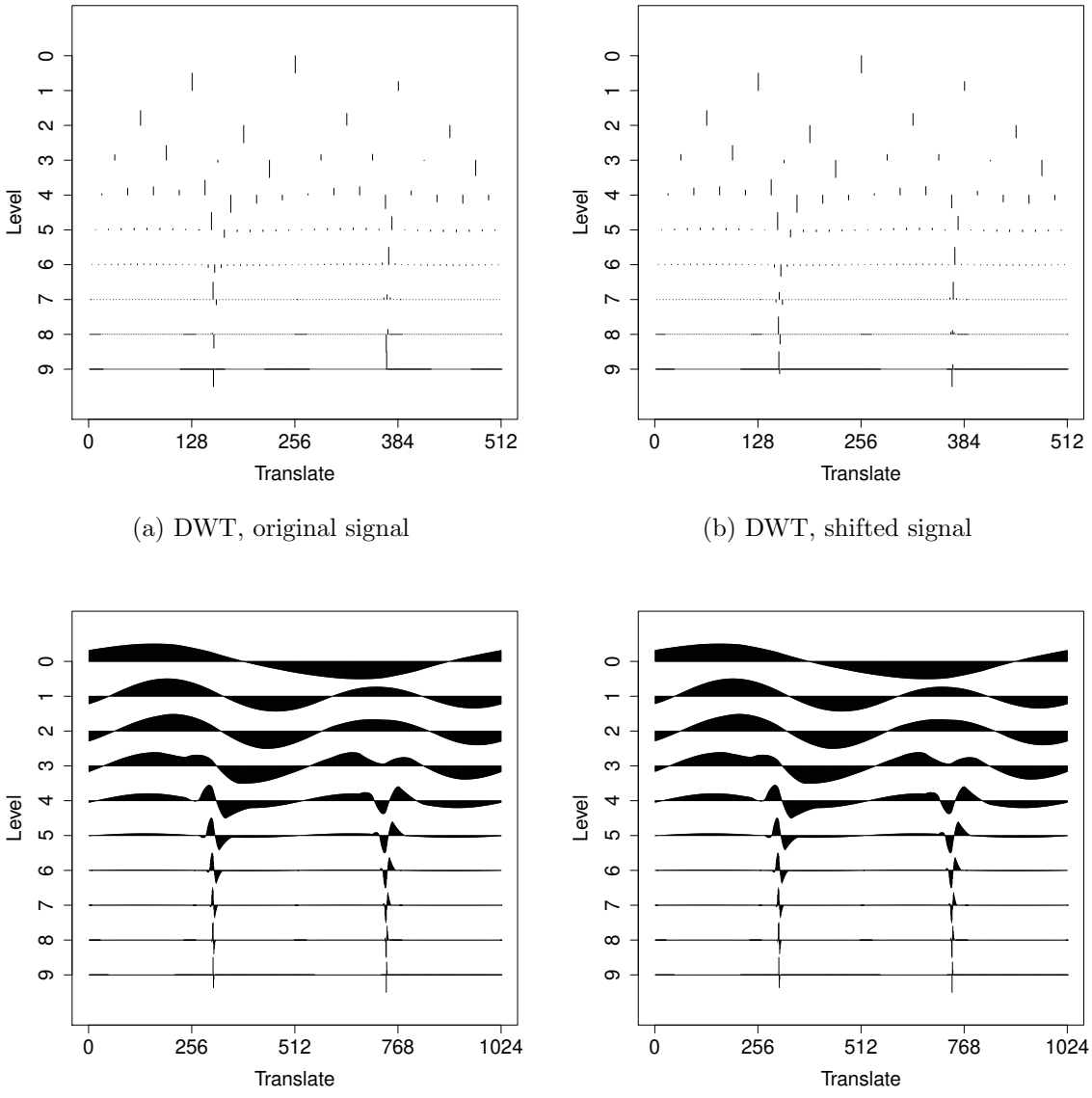
(a) Haar filter

(b) D4 filter

(c) C6 filter

(d) LA8 filter

Figure 2.10: Comparison of the MODWT with Haar, D4, C6 and LA8 filters applied to the Doppler test signal



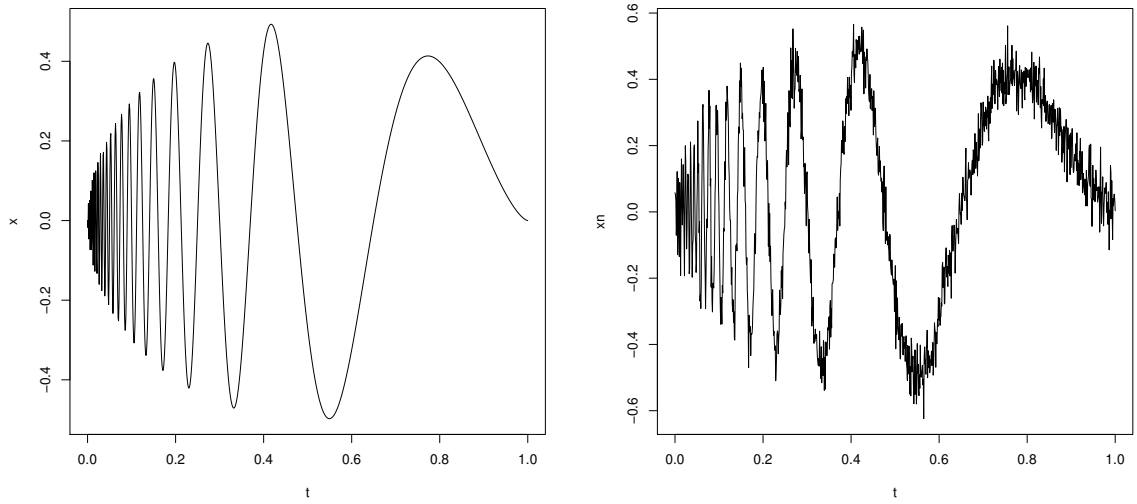
(a) DWT, original signal

(b) DWT, shifted signal

(c) MODWT, original signal

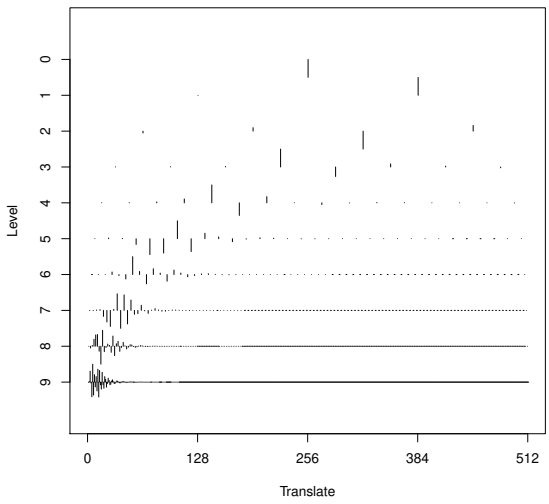
(d) MODWT, shifted signal

Figure 2.11: Demonstration of shift invariance of the MODWT compared to the DWT using the D4 filter applied to the Heavisine test signal, both unshifted and shifted.

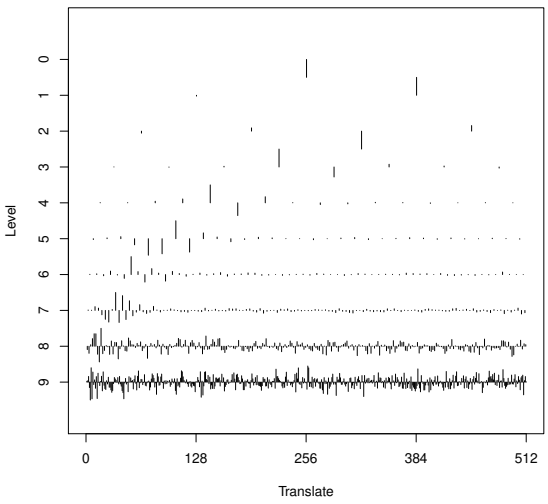


(a) Clean Doppler signal

(b) Noisy Doppler signal



(c) DWT of clean Doppler signal



(d) DWT of noisy Doppler signal

Figure 2.12: Comparison of the LA8 DWT applied to clean and noisy Doppler signals

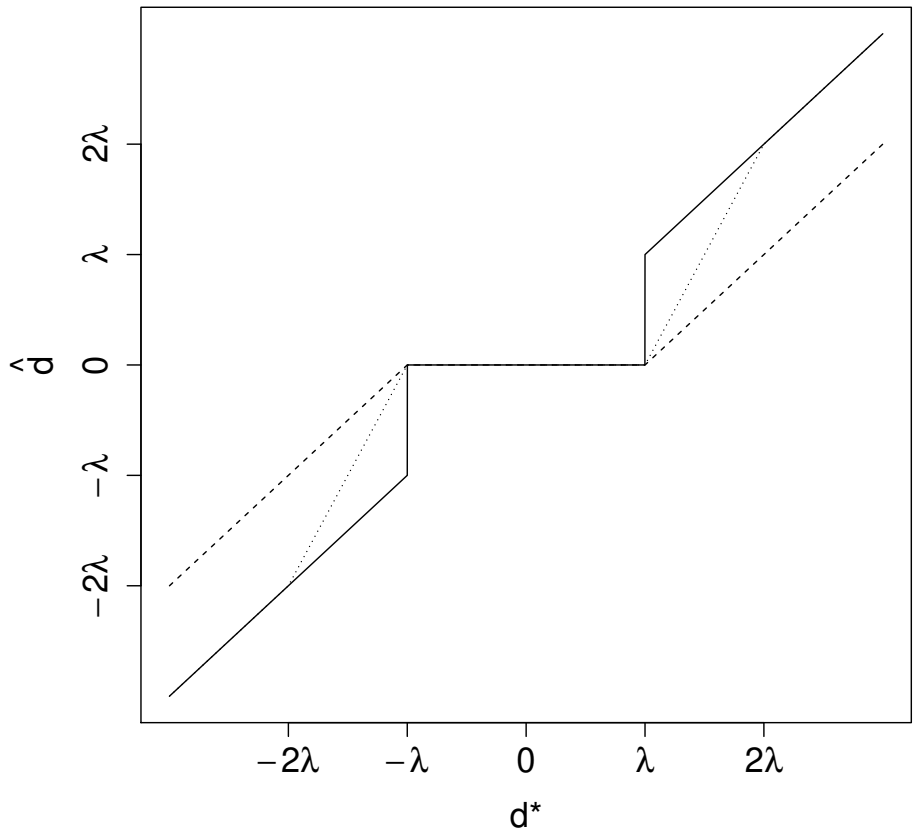
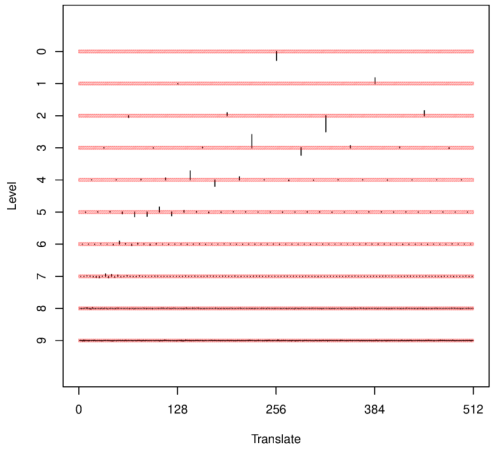
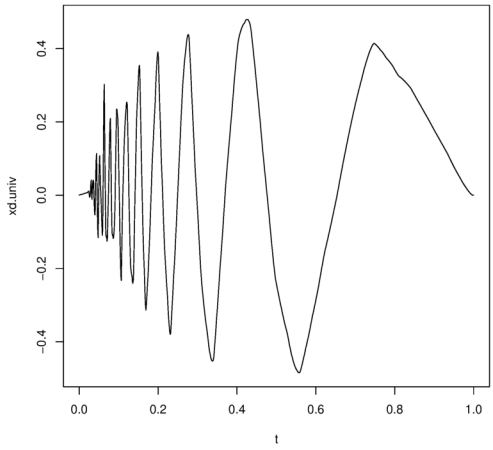


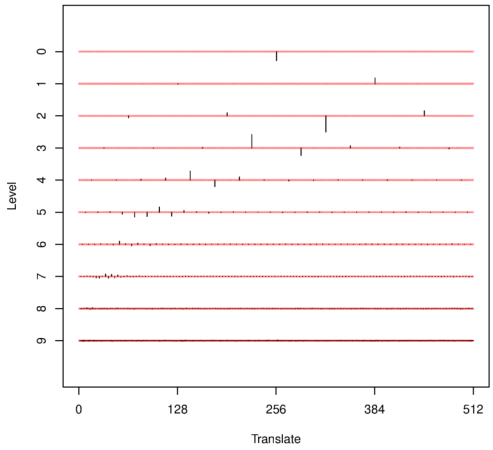
Figure 2.13: Hard (solid line), soft (dashed line) and mid (dotted line) thresholding policies plotted for a noisy detail d^* thresholded to $\hat{d}$ at a threshold level of λ.



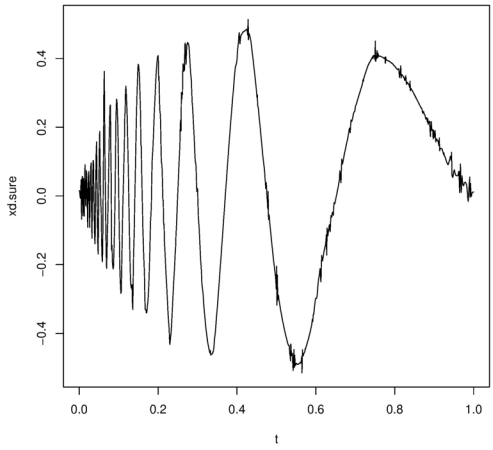
(a) DWT, annotated universal threshold



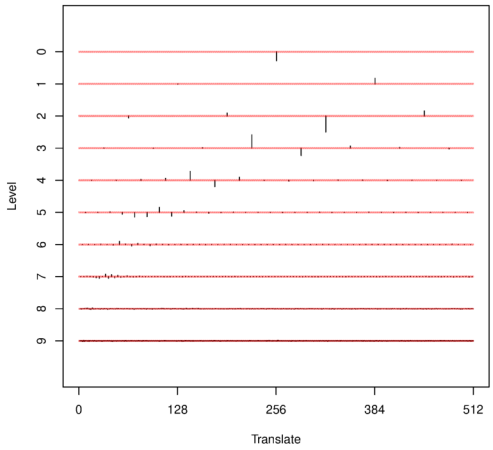
(b) Denoised signal, universal threshold



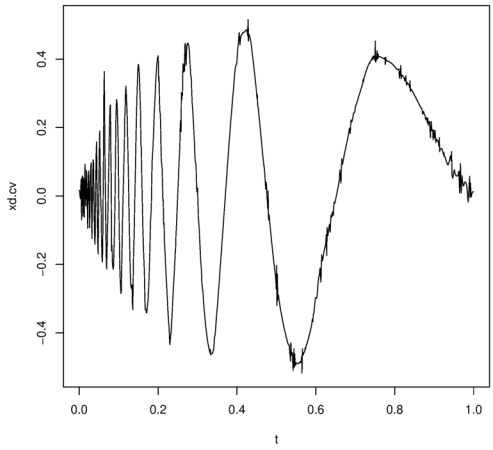
(c) DWT, annotated SURE threshold



(d) Denoised signal, Sure threshold



(e) DWT, annotated CV threshold



(f) Denoised signal, CV threshold

Figure 2.14: Comparison of different thresholding schemes applied to a noisy Doppler signal. We annotate the thresholds with red segments on the left hand plots.

2.9 Literature Review

We now present a review of the statistical literature on wavelets, focussing on the recent work on time series and changepoint detection.

Wavelets were first defined in the 1980s by Morlet, Arens, Fourgeau & Glard (1982), although a wavelet basis had already been discovered as simply a functional basis by Haar at the start of the century (Haar 1910). Since the 1990s, wavelets have enjoyed a "revolution" of sorts in the areas of mathematical analysis and statistics.

The wavelet transform is often described in relation to the Fourier transform; whilst the Fourier transform gives us only frequency information, the wavelet transform retains both time and frequency information (although in the context of wavelets this frequency component is technically *scale*). Another advantageous feature of wavelets is sparsity: given a stationary signal, its wavelet transformation is likely to be sparse, i.e. it will contain lots of zeros. This has advantages for data storage, compression and parsimony. One other notable feature of the wavelet transformation is that it can be performed in $\mathcal{O}(n)$ operations, in the case of the standard, dyadic, transform using what is called the *pyramid algorithm*. This is fast for an algorithm, and, notably, faster than the FFT (Fast Fourier Transform) which is $\mathcal{O}(n \log n)$.

There is now a sizeable literature on wavelets. The standard references include books by Percival & Walden (2000), Daubechies (1992) and Vidakovic (1999). The book by Strang & Nguyen (1997) provides a good applied approach to wavelets and filter banks from the perspective of signal processing. Nason (2008) has also written a practical book on wavelets for statistics using R. The wavelet background material in the following sections is adapted from a combination of all of these.

The majority of recent papers on wavelets have been mainly in the engineering literature looking at applications in hydrology, (see for example a review paper by Sang (2013)) watermarking (e.g. Lei, Soon, Zhou, Li & Lei (2012)), image smoothing (e.g. Ho & Hwang (2013)) and fault detection (e.g. a review paper by Feng, Liang & Chu (2013)), to name a few. Furthermore, wavelet analysis is now implicit in papers that explore time/frequency analysis. There has also been a lot of work in wavelets with neural networks (e.g. Amina, Kodogiannis, Petrounias & Tomtsis (2012)). There have been relatively fewer papers on statistical time series modelling following the seminal papers of Donoho & Johnstone (1994, 1995), however, the recent wavelet research interests have been mainly around Locally Stationary Wavelets (LSWs), complex wavelets, unbalanced Haar wavelets and wavelet variance, as well as CUDA wavelet implementations, wavelet cross validation and wavelets for irregularly-spaced data.

Nason, von Sachs & Kroisandt (2002) introduced a new model of LSWs for studying classes of non-stationary time series via constraints on the model coefficients. This allowed them to define an Evolutionary Wavelet Spectrum (EWS), measuring the local power in LSW processes at arbitrary (rescaled) time and scale. This provided for non-stationary processes an analogue of

the spectrum which is only defined for stationary processes. Fryzlewicz & Nason (2006) extended this by forcing a piecewise smoothness constraint on the EWS and smoothing the wavelet periodogram with a Fisz transform. Eckley, Nason & Treloar (2010) extend LSW processes to 2D where it is used to characterised different textures. Then Nason (2013*a*) developed a test for second order stationarity and approximate confidence intervals for localised autocovariances for locally stationary time series. Killick, Eckley & Jonathan (2013) used LSW processes to derive a hypothesis test on second order structural changes in a non-stationary signal using its EWS. Eckley & Nason (2014) manage to derive a spectral correction for locally stationary Shannon wavelet processes that reverses aliasing induced by dyadic sampling; this allows us to ascertain that the sampling rate of a time series does not need to be increased. Fryzlewicz & Ombao (2009) uses the EWS of LSW processes to discriminate between time series with unknown group memberships, which is demonstrated on seismic recordings from earthquakes and nuclear explosions. Lastly, recently Norwood & Killick (2018) uses the EWS to distinguish between long memory and changepoint models. The LSW model is often compared to the SLEX (Smoothed Local Complex EXponential) model of Ombao, Raz, von Sachs & Malow (2001), which fits local Fourier-like (not wavelet) basis vectors. Ombao, Raz, Von Sachs & Guo (2002) then adds an evolutionary wavelet spectrum to this using a Haar wavelet expansion of the spectrum over time. Huang, Ombao & Stoffer (2004) use the SLEX spectrum to classify non-stationary time series, which is also applied to earthquake and explosion seismic data.

Complex wavelets have been about since Morlets (Grossmann & Morlet 1984), but recently they have seen a resurgence with dual-tree complex wavelets (Kingsbury 2001) due to their properties of approximate shift invariance and directionally selective filters without the amount of redundancy and associated computation costs of the MODWT (Maximal Overlap Discrete Wavelet Transform). Selesnick, Baraniuk & Kingsbury (2005) give a more comprehensive account of dual-tree complex wavelets along with additional methods for deriving filters in a tutorial paper for the IEEE Signal Processing Magazine. Nelson, Gibberd, Nafornita & Kingsbury (2017) combine the theory on dual-tree complex wavelets with the work on LSWs, and the combination provides an effective model for non-stationary two-dimensional processes where directionality is necessary. They derive a (biased) periodogram (which can be used to give an unbiased estimate of the spectrum if the biasing matrix is invertible) and introduce a new stationarity test. Barber & Nason (2004) use complex-valued Daubechies wavelets and derive thresholding procedures for real-valued signals using multiwavelets and empirical Bayes methods. Although they discard the complex part of the estimated signal, their reconstructions achieve low errors. Olhede & Walden (2004*b*), on the other hand, use a (complex-valued) Hilbert transform to calculate the "analytic signal" from real-valued wavelet detail coefficients and thus a Hilbert spectrum. This method is shown to be more effective than EMD (Empirical mode Decomposition) at identifying multiple frequency subbands, and can also be used to derive thresholding procedures based on the amplitude of the analytic signal. They demonstrate further that this thresholding works well when wavelet coefficients oscillate about 0 (Olhede & Walden 2004*a*). Olhede & Metikas (2009) extend the 1D analytic wavelet transform to a 2D "monogenic" transform using quaternionic

wavelet functions. Lilly & Olhede (2010) consider modulation induced bias of analytic wavelet transforms, and they consider a generalised Morse wavelet with parameters that minimise this bias; later, they further refine these generalised Morse wavelets to so-called "Airy wavelets" that are most Gaussian and most symmetric (Lilly & Olhede 2012).

Girardi & Sweldens (1997) introduced "unbalanced Haar" (UH) wavelet transforms, which are tree-based wavelet transforms related to Haar which differ in that unlike the Haar wavelets they do not necessarily have jumps in the middle of their basis functions. Fryzlewicz (2007) uses the UH transform for signal estimation, citing key advantages that a dyadic length signal is not required and it also identifies jumps at non-dyadic locations. The author discusses function estimation using modified wavelet thresholding and changepoint detection. He later extends the changepoint detection methodology with UH to a method called Wild Binary Segmentation, which involves sampling random sequences from a signal and computing a CUSUM statistic on each; once a maximum statistic is found and it is determined to be significant, the search resumes either side of the new changepoint (Fryzlewicz 2013). Fryzlewicz & Timmermans (2015) extend the concept of UH to two dimensions with SHAH (SHape Adaptive Haar) wavelets that allow for extremely sparse transformations of piecewise-constant images. Cho & Fryzlewicz (2015) propose a new method SBS (Sparsified Binary Segmentation) which is used to detect multiple changepoints in high-dimensional time series by considering the second-order structure. Thus, it combines a multivariate LSW model with changepoint detection using binary segmentation on its second-order structure; the changepoints are sparsified by only keeping those that pass a threshold and consolidating those found on different scales.

A precursor to the EWS is wavelet variance, initially considered for processes whose dth order difference is second-order stationary. Percival (1995) derives an unbiased estimator of wavelet variance using the MODWT and computes confidence intervals using a normal approximation. Serroukh, Walden & Percival (2000) identify that a time-independent wavelet variance can be found in time series that can be nonstationary provided the backwards differenced process is locally stationary, and they prove that wavelet variance is asymptotic normally distributed. Related to wavelet variance is the concept of wavelet coherence, which measures the squared cross-correlation of two signals as a function of scale (see Torrence & Compo (1998) for a popular, practical reference); although the coherence is identically one at all times and scales, this can be circumvented by smoothing (Liu 1994). Plett (2007) uses wavelet coherence (and cross wavelet spectra) to detect transient oscillations in two signals. Cohen & Walden (2010) derive the wavelet coherence analytically for time series that are complex-valued, jointly stationary and Gaussian, and show that this coherence is asymptotically distributed with what they call "Goodman's distribution". Park, Eckley & Ombao (2014) extend wavelet coherence with multivariate LSWs to partial coherence that removes linear effects from other signals. This multivariate cross-dependence structure for non-stationary signals also allows them to simulate processes with a given time-scale partial coherence form.

Wavelets have been successfully used in the context of signal estimation since the seminal work

of Donoho & Johnstone (1994). This original work leads to very simple and fast implementations of wavelet smoothing. Nason (1996) extended the concept of cross validation to wavelet signal estimation: this becomes more computationally demanding and motivates faster implementations which we provide in our software package WaveCUDA. Abramovich & Benjamini (1996) introduced an adaptive thresholding methodology that controls the false discovery rate. Abramovich, Sapatinas & Silverman (1998) and Vidakovic (1998) then extended the methodology to provide Bayesian approaches to wavelet thresholding.

Chapter 3

A Review of Lifting Schemes

We move on from the standard DWT to the lifting scheme because the lifting scheme allows us to generalise the DWT with added benefits. Not only does the lifting scheme give us new implementations of existing DWTs with fewer operations, but it also gives added benefits of parallelisation with in-place transformations. With the except of the Haar transform, most DWTs are not immediately parallelisable writing in-place in memory. This chapter outlines the lifting methodology and introduces a new lifting factorisation of the LA8 transform following the algorithm of (Daubechies & Sweldens 1998) which we have published (Waton & McCoy 2016). We also develop further factorisations which allow us to implement fast algorithms.

The wavelet lifting scheme (Sweldens 1996, Sweldens 1998, Daubechies & Sweldens 1998) is a method, developed by Wim Sweldens at the end of the twentieth century, for constructing custom-designed wavelets and for speeding up further the wavelet transform. Sweldens named these *second-generation wavelets*, as they are a generalisation of 'traditional' wavelets that are not necessarily translation/dilation invariant, whilst maintaining other useful properties of wavelets. The wavelets are biorthogonal, which means that we require two different pairs of filters; one to perform the analysis, and the other to reconstruct our original data.

In recent years, the lifting scheme has been found to be exploitable with GPGPU (General-Purpose computing on Graphical Processor Units) to speed up computation of wavelet transforms, because of the scope for parallelisation and the reduced computational complexity compared to the standard wavelet transform (Van Der Laan, Jalba & Roerdink 2011).

The lifting scheme is so-called because it comprises a prediction step that is updated or *lifted* to produce the next layer of the transform.

We present the lifting scheme here as given in Daubechies & Sweldens (1998), as there it is presented in a neat, self-contained formation using *Laurent Polynomials*. This chapter follows the schematics of the paper, with some added explanations where they are deemed necessary. A more thorough introduction to the wavelets via lifting can be found in Jensen & la Cour-Harbo

(2001).

Daubechies and Sweldens factorised a number of standard wavelet transforms into lifting steps (Daubechies 1993), namely Haar, D4 (Daubechies 4-tap orthonormal filter with two vanishing moments), D6 and the (9-7) filter. These lifted transforms have been used in parallel implementations (Hopf & Ertl 1999, Tenllado & Setoain 2008, Van Der Laan et al. 2011), but we have not found further wavelet transforms factorised into lifting steps in the literature apart from a few by Jensen & la Cour-Harbo (2001) and in a relatively unknown paper by Maslen & Abbott (2000) that claims to automate the process; however, we were unable to verify their software as it is outdated. The paper only lists a couple of transforms, and the factorisations are limited by considering quotients of degree one. Instead, in the literature there are many examples of lifting schemes that move away from first to second generation wavelets where the transforms are reformulated from three stages: split, predict and update (Sweldens 1995). These do not necessarily have the feature of being translates and dilates of one wavelet function. Examples include adaptive wavelet transforms by Claypoole, Baraniuk & Nowak (1998) and a transform from Knight & Nason (2008) that lifts one coefficient at a time.

3.1 Laurent Polynomials

This section outlines the theory needed to develop the lifting transform. We will now consider our filters as Laurent polynomials. We only use FIR filters.

Definition 3.1.1. *A* **Laurent polynomial** *is a polynomial over a field $\mathbb{F}$ with positive and negative integer powers.*

Each filter h can be expressed as a Laurent polynomial using a *z-transformation*:

$$h(z) = \sum_{k=k_l}^{k_u} h_k z^{-k}, \tag{3.1}$$

where the filter h is defined by its coefficients $\{h_k \in \mathbb{R} : k \in \mathbb{Z}\}$, k_l is the first non-zero filter coefficient and k_u is the last non-zero filter coefficient; the length of the filter is hence $k_u - k_l + 1$. h can denote either the filter or the Laurent polynomial here.

Remark 3.1.2. *We are free to choose k_u or k_l to shift the impulse response for a wavelet filter h_k of given length in expressing h as a Laurent polynomial; however, in factorising wavelet transforms into lifting steps, we will have to ensure that g and h are complementary (Daubechies & Sweldens 1998) (see Definition 3.2.7), which means that they give perfect reconstruction. This can have an effect on the time alignment of our wavelet detail and scaling coefficients, but we can correct that using circular shifting after transforming.*

Definition 3.1.3. *The* **degree** *of the Laurent polynomial is defined to be:*

$$|h| = k_u - k_l. \tag{3.2}$$

The degree of the zero polynomial is defined as $-\infty$.

Laurent polynomials are defined on a commutative ring, $\mathbb{R}[z, z^{-1}]$ where addition and subtraction of Laurent polynomials gives Laurent polynomials. Multiplication and division of Laurent polynomials is also defined, producing a Laurent polynomial whose degree is the sum and difference (respectively) of the original polynomials. However, division, which gives a quotient and a remainder, is not unique; this is exploited in the lifting scheme. If the degree of a polynomial is zero (i.e. it is a monomial), then division by this polynomial is exact. A Laurent polynomial is invertible if and only if it is a monomial (Daubechies & Sweldens 1998).

Remark 3.1.4. *Monomial division is clearly unique: if we compute* a/b *then there can only be one unique quotient with no remainder. However, if we divide for example* $b_0 + b_1 z$ *into* $a_0 + a_1 z$*, there are two options:*

$$a_0 + a_1 z = \frac{a_0}{b_0}(b_0 + b_1 z) + \left(a_1 - \frac{a_0}{b_0} b_1\right) z,$$

or

$$a_0 + a_1 z = \frac{a_1}{b_1}(b_0 + b_1 z) + \left(a_0 - \frac{a_1}{b_1} b_0\right).$$

We consider 2×2 matrices of Laurent polynomials, as in (3.3):

$$M(z) = \begin{pmatrix} a(z) & b(z) \\ c(z) & d(z) \end{pmatrix}. \tag{3.3}$$

These also form a ring, $M(2; \mathbb{R}[z, z^{-1}])$. By the property stated above, it follows that if the determinant of the matrix M is a monomial then M is invertible.

Definition 3.1.5. *A matrix* $M \in M(2; \mathbb{R}[z, z^{-1}])$ *is* **unitary** *if*

$$M(z)^{-1} = M(z^{-1})^T.$$

Remark 3.1.6. *This differs from the usual definition of a unitary matrix which is one whose inverse is its conjugate transpose (see e.g. (Zhang 2011)).*

3.2 The Wavelet Transform

We now use the Laurent polynomial notation to describe the wavelet transform. In lifting, we consider biorthogonal wavelet transforms where we have two sets of filter pairs; one pair,

(h, g) for analysis: (the forward transformation), and the other, $(\tilde{h}, \tilde{g})$ for synthesis (the inverse transformation). In this thesis, we only consider orthogonal wavelet transforms, however this generalisation is an important feature of the lifting scheme.

Remark 3.2.1. *We choose to denote the forward transform as (h, g) and the inverse as $(\tilde{h}, \tilde{g})$ for convenience since it is more natural to work using the forward transformation, although Daubechies & Sweldens (1998) opt for the opposite notation.*

An extremely important property in signal processing is *perfect reconstruction*; this corresponds here to the idea of being able to perform a wavelet transformation to a signal followed by the inverse transformation and obtaining the original signal again without *distortion* or *aliasing*. Equally, these are conditions for biorthogonality (Strang & Nguyen 1997).

Definition 3.2.2. *The conditions for perfect reconstruction for biorthogonal FIR filters are:*

$$h(z)\tilde{h}(z^{-1}) + g(z)\tilde{g}(z^{-1}) = 2, \tag{3.4a}$$

$$h(z)\tilde{h}(-z^{-1}) + g(z)\tilde{g}(-z^{-1}) = 0, \tag{3.4b}$$

where (3.4a) is the condition for **no distortion** *and (3.4b) is for* **no aliasing**.

Remark 3.2.3. *Aliasing here is when two inputs to a filter give the same output. Distortion is some systematic change in the phase or amplitude of a signal due to filtering. Clearly, to perfectly reconstruct a signal, we need to avoid both in our filters.*

Definition 3.2.4. *The* **modulation matrix** *is defined to be*

$$M(z) = \begin{pmatrix} g(z) & g(-z) \\ h(z) & h(-z) \end{pmatrix}. \tag{3.5}$$

The **dual modulation matrix** $\tilde{M}(z)$ *is defined likewise.*

Hence the conditions for perfect reconstruction in Definition 3.2.2 can be rewritten as

$$\tilde{M}(z^{-1})^T M(z) = 2I, \tag{3.6}$$

where I is the 2×2 identity matrix.

Definition 3.2.5. *The* **polyphase representation** *is a way of expressing a filter h as a sum of its odd (h_o) and even (h_e) components:*

$$h(z) = h_e(z^2) + z^{-1}h_o(z^2) \tag{3.7}$$

which gives

$$h_e(z) = \sum_k h_{2k} z^{-k}$$

and

$$h_o(z) = \sum_k h_{2k+1} z^{-k},$$

where our ranges on k are finite and functions of k_l and k_u, but varying according to whether they are even or odd.

Definition 3.2.6. *The* **polyphase matrix** *is defined to be*

$$P(z) = \begin{pmatrix} g_e(z) & h_e(z) \\ g_o(z) & h_o(z) \end{pmatrix}.$$

The conditions for perfect reconstruction can also be written in terms of P:

$$P(z)\tilde{P}(z^{-1})^T = I. \tag{3.8}$$

Since the determinants of $P(z)$ and $P(z^{-1})$ must both be Laurent polynomials they must also be invertible, hence monomials. We can then assume that the determinants are 1 *wlog*, because otherwise we can divide the filter components h_o and h_e by the actual determinant of $P(z)$.

Definition 3.2.7. *A filter pair (h, g) is* **complementary** *if its polyphase matrix $P(z)$ has a determinant of 1.*

Now, all we need to do to find a pair of complementary FIR filters is to find any such matrix $P(z)$ with determinant 1. Using (3.8), we can form the following identities:

$$\tilde{g}(z) = -z^{-1}h(-z^{-1}), \tag{3.9a}$$
$$\tilde{h}(z) = z^{-1}g(-z^{-1}). \tag{3.9b}$$

3.3 Lifting

We are now in a position to state the lifting scheme, which will allow us to write our GPU-accelerated wavelet library, using the notation from the previous sections.

Theorem 3.3.1 (Lifting, Daubechies & Sweldens (1998))**.** *Let (h, g) be a complementary filter pair. Then any other filter h^{new} is complementary to g if and only if it can be written as:*

$$h^{new}(z) = h(z) + g(z)s(z^2),$$

where $s(z)$ is a Laurent polynomial. This processes of creating a new filter h^{new} from h is known as **lifting**.

The new polyphase matrix is

$$P^{new}(z) = P(z) \begin{pmatrix} 1 & s(z) \\ 0 & 1 \end{pmatrix}. \tag{3.10}$$

Then, using (3.9), we can reformulate this theorem for $\tilde{g}$:

$$\tilde{g}^{new}(z) = \tilde{g}(z) - \tilde{h}(z)s(z^{-2}). \tag{3.11}$$

Theorem 3.3.2 (Dual Lifting, Daubechies & Sweldens (1998))**.** *Let (h, g) be a complementary filter pair. Then any other filter g^{new} is complementary to h if and only if it can be written as:*

$$g^{new}(z) = g(z) + h(z)t(z^2),$$

where $t(z)$ is a Laurent polynomial. This processes is known as **dual lifting** *as it can be used to lift the dual of a wavelet filter to a new dual wavelet filter.*

The new polyphase matrix after dual lifting is calculated using

$$P^{new}(z) = P(z) \begin{pmatrix} 1 & 0 \\ t(z) & 1 \end{pmatrix}. \tag{3.12}$$

As in the lifting case, this can be reformulated for the complementary filter, here $\tilde{h}$:

$$\tilde{h}^{new}(z) = \tilde{h}(z) - \tilde{g}(z)t(z^{-2}). \tag{3.13}$$

This lifting process can be performed multiple times to create new filter pairs with different properties.

3.4 Euclidean and Lifting Algorithms

We can factor the polyphase matrix into successive lifting steps using the Euclidean Algorithm to find the greatest common divisor (gcd) of two Laurent polynomials. Two Laurent polynomials are *relatively prime* if their gcd is a monomial, or has degree 0; i.e. it is of the form z^k or z^{-k}, for $k \in \mathbb{N}$.

Theorem 3.4.1 (Euclidean Algorithm)**.** *Let $a(z)$ and $b(z) \neq 0$ be two Laurent polynomials such that $|a(z)| \geq |b(z)|$. Starting from $i = 0$ and setting $a_0(z) = a(z)$ and $b_0(z) = b(z)$, we iterate*

$$\begin{aligned} a_{i+1}(z) &:= b_i(z) \\ q_{i+1}(z) &:= \left\lceil \frac{a_i(z)}{b_i(z)} \right\rceil \end{aligned}$$

$$b_{i+1}(z) := a_i(z) - b_i(z)q_{i+1}(z)$$
$$i := i + 1$$
if $b_i(z) = 0$ *then exit and set* $n := i$.

Then $a_n(z) = gcd\,[a(z), b(z)]$.

Now, we can write

$$\begin{pmatrix} a_n(z) \\ 0 \end{pmatrix} = \prod_{i=n}^{1} \begin{pmatrix} 0 & 1 \\ 1 & -q_i(z) \end{pmatrix} \begin{pmatrix} a(z) \\ b(z) \end{pmatrix}, \tag{3.14}$$

Hence,

$$\begin{pmatrix} a(z) \\ b(z) \end{pmatrix} = \prod_{i=1}^{n} \begin{pmatrix} q_i(z) & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_n(z) \\ 0 \end{pmatrix}. \tag{3.15}$$

Likewise, complementary filters (h, g) can be factored into lifting steps. For a filter g, $g_e(z)$ and $g_o(z)$ must be relatively prime; otherwise, their gcd would divide the determinant of their polyphase matrix, which is 1. We can choose this gcd to be a constant, K, giving

$$\begin{pmatrix} g_e(z) \\ g_o(z) \end{pmatrix} = \prod_{i=1}^{n} \begin{pmatrix} q_i(z) & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} K \\ 0 \end{pmatrix}. \tag{3.16}$$

We can then find a complementary filter h^0

$$P^0(z) = \begin{pmatrix} g_e(z) & h_e^0(z) \\ g_o(z) & h_o^0(z) \end{pmatrix} = \prod_{i=1}^{n} \begin{pmatrix} q_i(z) & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & 1/K \end{pmatrix}. \tag{3.17}$$

The far right diagonal matrix is such because we can assume that n is even; if it is odd, we can multiply $g(z)$ by z and $h(z)$ by $\frac{-1}{z}$. Equation (3.17) can then be rewritten, exploiting this property further, as

$$P^0(z) = \prod_{i=1}^{n/2} \begin{pmatrix} 1 & q_{2i-1}(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ q_{2i}(z) & 1 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & 1/K \end{pmatrix}. \tag{3.18}$$

Then the filter h can be recovered by lifting the polyphase matrix, using Theorem 3.3.1:

$$P(z) = P^0(z) \begin{pmatrix} 1 & s(z) \\ 0 & 1 \end{pmatrix}. \tag{3.19}$$

Hence, we are now in a position to state a theorem saying that any FIR wavelet transform can be produced by means of starting with a trivial transform (the *Lazy Wavelet*: see the next section) followed by a series of lifting steps and scaling.

Theorem 3.4.2 (Daubechies & Sweldens (1998))**.** *For any complementary FIR filter pair* (h, g)*,*

there exist Laurent polynomials $s_i(z)$ and $t_i(z)$, as well as a (non-zero) constant K such that

$$P(z) = \prod_{i=1}^{m} \begin{pmatrix} 1 & s_i(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ t_i(z) & 1 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & 1/K \end{pmatrix}. \tag{3.20}$$

Here, we follow the steps above, then set $m = \frac{n}{2} + 1$, $s_m(z) = K^2 s(z)$ and $t_m(z) = 0$.

In the dual setting, we have

$$\tilde{P}(z) = \prod_{i=1}^{m} \begin{pmatrix} 1 & 0 \\ -s_i(z^{-1}) & 1 \end{pmatrix} \begin{pmatrix} 1 & -t_i(z^{-1}) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1/K & 0 \\ 0 & K \end{pmatrix}. \tag{3.21}$$

To summarise the above steps, we present the lifting algorithm in Algorithm (3.4.3).

Algorithm 3.4.3 (The Lifting Algorithm (Daubechies & Sweldens 1998))**.** *Take the filter pair (h, g)*

i Divide g_e by g_o using the Euclidean algorithm, giving an equation of the form (3.16).

ii Find a complementary filter h_o using (3.17).

iii Lift the complementary filter h_o to the true filter h by finding a polynomial s satisfying the expression in Theorem 3.3.1.

iv The dual filter pair $(\tilde{h}, \tilde{g})$ can be formed by comparing (3.20) with (3.21). In practice, this corresponds to reversing the sign of any steps in the lifted wavelet transform algorithm calculated for (h, g).

v The lifted wavelet transform can then be written out using factorisation (3.20) or (3.21). The lifted inverse transform is simply the forward transform in reverse order with switched signs (i.e. the additions become subtractions and vice versa).

3.5 Lifted Filters

In this section, we present the lifting coefficients for the wavelet filters introduced in Section 2.4. Our first example of lifting a transform, however, is the *Lazy wavelet*.

Example 3.5.1 (Lazy wavelet (Daubechies & Sweldens 1998))**.** *In the case that the polyphase matrix $P(z)$ is the identity matrix, we have what is called the* **Lazy wavelet**. *Here we have that $g(z) = \tilde{g}(z) = 1$ and $h(z) = \tilde{h}(z) = z^{-1}$. Here, the conditions for perfect reconstruction are trivially satisfied logically or immediately by (3.8). This wavelet subsamples a vector into even and odd samples.*

3.5.1 Haar Wavelet

The unnormalised Haar wavelet transform is factorised in (Daubechies & Sweldens 1998); here we factorise the normalised version. The Haar wavelet has the following (orthogonal) filters:

$$g(z) = \frac{1}{\sqrt{2}}(1 + z^{-1}) \quad = \tilde{g}(z), \tag{3.22}$$

$$h(z) = \frac{1}{\sqrt{2}}(-1 + z^{-1}) = \tilde{h}(z). \tag{3.23}$$

We perform the Euclidean algorithm on g and hence form P^0:

$$P^0(z) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \sqrt{2} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 2 \end{pmatrix}.$$

This gives us $K = \frac{1}{\sqrt{2}}$, and we can form our factorisation of P:

$$P(z) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \sqrt{2} \end{pmatrix}.$$

This corresponds to the forward transform given in Algorithm 1.

Algorithm 1: Haar filter lifting algorithm

$$\begin{aligned} s_l^{(0)} &= x_{2l} + x_{2l+1} \\ d_l^{(0)} &= x_{2l+1} - \frac{1}{2} s_l^{(0)} \\ s_l^{(1)} &= \frac{1}{\sqrt{2}}\, s_l^{(0)} \\ d_l^{(1)} &= \sqrt{2}\, d_l^{(0)}. \end{aligned}$$

3.5.2 Daubechies 4 Wavelet

We factorise the Daubechies 4 wavelet here using the "alternative factorisation" given by Daubechies & Sweldens (1998). Note however that their polyphase matrix contains a small typographical error which we correct here. However, in our WaveCUDA package, we use their first factorisation, as it has one fewer floating point operation per run of the algorithm (i.e. per level of transform).

The Daubechies 4 wavelet filter can be expressed as the following Laurent polynomials:

$$g(z) = \left(\frac{1+\sqrt{3}}{4\sqrt{2}}\right) z + \left(\frac{3+\sqrt{3}}{4\sqrt{2}}\right) + \left(\frac{3-\sqrt{3}}{4\sqrt{2}}\right) z^{-1} + \left(\frac{1-\sqrt{3}}{4\sqrt{2}}\right) z^{-2} = \tilde{g}(z),$$

$$h(z) = \left(\frac{1-\sqrt{3}}{4\sqrt{2}}\right) z - \left(\frac{3-\sqrt{3}}{4\sqrt{2}}\right) + \left(\frac{3+\sqrt{3}}{4\sqrt{2}}\right) z^{-1} - \left(\frac{1+\sqrt{3}}{4\sqrt{2}}\right) z^{-2} = \tilde{g}(z).$$

The polyphase matrix $P(z)$ is thus given by

$$P(z) = \begin{pmatrix} g_e(z) & h_e(z) \\ g_o(z) & h_o(z) \end{pmatrix} = \begin{pmatrix} \left(\frac{3+\sqrt{3}}{4\sqrt{2}}\right) + \left(\frac{1-\sqrt{3}}{4\sqrt{2}}\right) z^{-1} & -\left(\frac{3-\sqrt{3}}{4\sqrt{2}}\right) - \left(\frac{1+\sqrt{3}}{4\sqrt{2}}\right) z^{-1} \\ \left(\frac{1+\sqrt{3}}{4\sqrt{2}}\right) z + \left(\frac{3-\sqrt{3}}{4\sqrt{2}}\right) & \left(\frac{1-\sqrt{3}}{4\sqrt{2}}\right) z + \left(\frac{3+\sqrt{3}}{4\sqrt{2}}\right) \end{pmatrix}.$$

After performing the Euclidean algorithm on g, dividing g_o into g_e, we find their (monomial) hcf and form the following factorisation:

$$\begin{pmatrix} g_e(z) \\ g_o(z) \end{pmatrix} = \begin{pmatrix} \frac{-z^{-1}}{\sqrt{3}} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{3}z}{4} + \frac{3(2-\sqrt{3})}{4} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{3+\sqrt{3}}{3\sqrt{2}} \\ 0 \end{pmatrix}.$$

Using this factorisation, we form a filter $h^o(z)$ complementary to $g(z)$

$$\begin{pmatrix} g_e(z) & h_e^o(z) \\ g_o(z) & h_o^o(z) \end{pmatrix} = \begin{pmatrix} \frac{-z^{-1}}{\sqrt{3}} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{3}z}{4} + \frac{3(2-\sqrt{3})}{4} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{3+\sqrt{3}}{3\sqrt{2}} & 0 \\ 0 & \frac{3-\sqrt{3}}{\sqrt{2}} \end{pmatrix}.$$

This gives us

$$h_e^o(z) = \frac{-\sqrt{2}z^{-1}}{1+\sqrt{3}} = \frac{4(1-\sqrt{3})z^{-1}}{4\sqrt{2}},\ h_o^o(z) = \frac{3\sqrt{2}}{3+\sqrt{3}} = \frac{4(3-\sqrt{3})}{4\sqrt{2}}.$$

From here, we calculate the polynomial $s(z)$ that can *lift* this polynomial $h^o(z)$ to the desired polynomial $h(z)$:

$$\begin{aligned} h_e(z) &= h_e^o(z) + s(z)g_e(z), \\ h_o(z) &= h_o^o(z) + s(z)g_o(z). \end{aligned}$$

This gives us $s(z) = \sqrt{3} - 2$. We can finally factorise $P(z)$, using matrix identities to switch the elements on the anti-diagonal and move the lifting matrix left:

$$\begin{pmatrix} g_e(z) & h_e(z) \\ g_o(z) & h_o(z) \end{pmatrix} = \begin{pmatrix} 1 & \frac{-z^{-1}}{\sqrt{3}} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \frac{\sqrt{3}z}{4} + \frac{3(2-\sqrt{3})}{4} & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{-1}{3} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{3+\sqrt{3}}{3\sqrt{2}} & 0 \\ 0 & \frac{3-\sqrt{3}}{\sqrt{2}} \end{pmatrix}.$$

The corresponding lifting algorithm to this factorisation is given in Algorithm 2.

3.5.3 Coiflet 6 Wavelet

We produce a new lifting factorisation for the C6 wavelet (Daubechies 1992) by implementing the work of (Daubechies & Sweldens 1998). We can express the filters in terms of the following

Algorithm 2: Daubechies 4 filter lifting algorithm

$$
\begin{aligned}
d_l^{(0)} &= x_{2l+1} - \frac{1}{\sqrt{3}}\, x_{2l+2} \\
s_l^{(0)} &= x_{2l} + \frac{(6-3\sqrt{3})}{4}\, d_l^{(0)} + \frac{\sqrt{3}}{4}\, d_{l-1}^{(0)} \\
d_l^{(1)} &= d_l^{(0)} - \frac{1}{3}\, s_l^{(0)} \\
s_l &= \frac{(3+\sqrt{3})}{(3\sqrt{2})}\, s_l^{(0)} \\
d_l &= \frac{(3-\sqrt{3})}{\sqrt{2}}\, d_l^{(1)}
\end{aligned}
$$

Laurent polynomials:

$$g(z) = c_0 z^2 + c_1 z + c_2 + c_3 z^{-1} + c_4 z^{-2} + c_5 z^{-3} \qquad = \tilde{g}(z), \tag{3.24}$$

$$h(z) = -(c_5 z^2 - c_4 z + c_3 - c_2 z^{-1} + c_1 z^{-2} - c_0 z^{-3}) = \tilde{h}(z). \tag{3.25}$$

We separate these into odd and even polynomials:

$$
\begin{aligned}
g_e(z) &= c_0 z + c_2 + c_4 z^{-1}, \\
g_0(z) &= c_1 z + c_3 + c_5 z^{-1}, \\
h_e(z) &= -(c_5 z + c_3 + c_1 z^{-1}), \\
h_o(z) &= -(-c_4 z - c_2 - c_0 z^{-1}).
\end{aligned}
$$

We have multiplied the filter h by (-1) in (3.25) to ensure that our polynomial filters are complementary. Multiplying the Laurent polynomial by a nonzero constant does not alter the filter.

$$
\begin{aligned}
|P(z)| &= \left| \begin{pmatrix} g_e(z) & h_e(z) \\ g_o(z) & h_o(z) \end{pmatrix} \right| \\
&= (z^2 + z^{-2})(c_0 c_4 + c_1 c_5) + (z + z^{-1})(c_0 c_2 + c_1 c_3 + c_2 c_4) + (c_0^2 + c_1^2 + c_2^2 + c_3^2 + c_4^2 + c_5^2) \\
&= 1.
\end{aligned}
$$

Using the Euclidean algorithm, we can factorise our filters. During the factorisation, we had to do one step of creative division (turning one division step into two steps) to ensure an even number of steps before the lifting step.

$$P(z) = \begin{pmatrix} 1 & \frac{1-\sqrt{7}}{-3+\sqrt{7}} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \frac{1}{8}\frac{(-3+\sqrt{7})^2}{z(3\sqrt{7}-7)} & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{4(3\sqrt{7}-7)(-1+\sqrt{7})}{(-3+\sqrt{7})(4\sqrt{7}-7)} \\ 0 & 1 \end{pmatrix}$$

$$\times \begin{pmatrix} 1 & 0 \\ \frac{1}{32}\frac{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)\left(z\left(4\sqrt{7}-7\right)+\left(30\sqrt{7}-75\right)\right)}{\left(3\sqrt{7}-7\right)\left(7\sqrt{7}-19\right)} & 1 \end{pmatrix} \begin{pmatrix} 1 & K^2\frac{3-\sqrt{7}}{1-\sqrt{7}} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & 1/K \end{pmatrix}.$$

where K is the gcd of g_e and g_o, which is determined to be...

$$K = \frac{2\left(-19+7\sqrt{7}\right)\sqrt{2}}{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)}.$$

We calculate our lifting polynomial as

$$s(z) = \frac{3-\sqrt{7}}{1-\sqrt{7}}.$$

This factorisation corresponds to the lifting algorithm in Algorithm 3, for which the coefficients are given by Table 3.1.

Algorithm 3: Coiflet 6 filter lifting algorithm

$$\begin{aligned}
d_l^{(0)} &= x_{2l+1} + q_{11}x_{2l} \\
s_l^{(0)} &= x_{2l} + q_{21}d_{l+1}^{(0)} \\
d_l^{(1)} &= d_l^{(0)} + q_{31}s_l^{(0)} \\
s_l^{(1)} &= s_l^{(0)} + q_{41}d_{l-1}^{(1)} + q_{42}d_l^{(1)} \\
d_l^{(2)} &= d_l^{(1)} + K^2 q_{51}s_l^{(1)} \\
s_l^{(2)} &= K s_l^{(1)} \\
d_l^{(3)} &= K^{-1}d_l^{(2)}
\end{aligned}$$

Coefficient	Exact value	Approximate value
q_{11}	$\frac{1-\sqrt{7}}{-3+\sqrt{7}}$	4.645751311064591e+0
q_{21}	$\frac{\left(-3+\sqrt{7}\right)^2}{z\left(3\sqrt{7}-7\right)}$	1.673667737846021e−2
q_{31}	$\frac{4\left(3\sqrt{7}-7\right)\left(-1+\sqrt{7}\right)}{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)}$	−4.861001748086121e+0
q_{41}	$\frac{1}{32}\frac{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)\left(4\sqrt{7}-7\right)}{\left(3\sqrt{7}-7\right)\left(7\sqrt{7}-19\right)}$	3.160743211769211e−1
q_{42}	$\frac{1}{32}\frac{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)\left(30\sqrt{7}-75\right)}{\left(3\sqrt{7}-7\right)\left(7\sqrt{7}-19\right)}$	3.857229635307634e−1
K^2q_{51}	$\frac{16\left(133\sqrt{(7)}-352\right)}{\left(3\sqrt{7}-7\right)\left(4\sqrt{7}-7\right)^2\left(-1+\sqrt{7}\right)}$	−2.460004994531774e−1
K	$\frac{2\left(-19+7\sqrt{7}\right)\sqrt{2}}{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)}$	1.069044967649698e+0
K^{-1}	$\frac{\left(-3+\sqrt{7}\right)\left(4\sqrt{7}-7\right)}{2\left(-19+7\sqrt{7}\right)\sqrt{2}}$	9.354143466934853e−1

Table 3.1: Coiflet 6 filter lifting coefficients

3.5.4 Least Asymmetric 8 Wavelet

We now produce a lifting factorisation for the LA8 wavelet using (Daubechies & Sweldens 1998). This section has been published as part of (Waton & McCoy 2016). The LA8 filter has 8 filter coefficients in each of its analysis and synthesis filters. We first express each filter as a Laurent polynomial:

$$g(z) = c_0 z + c_1 + c_2 z^{-1} + c_3 z^{-2} + c_4 z^{-3} + c_5 z^{-4} + c_6 z^{-5} + c_7 z^{-6} = \tilde{g}(z), \tag{3.26}$$

$$h(z) = c_7 z^5 - c_6 z^4 + c_5 z^3 - c_4 z^2 + c_3 z - c_2 + c_1 z^{-1} - c_0 z^{-2} \qquad = \tilde{h}(z). \tag{3.27}$$

We can split (3.26) and (3.27) up into odd and even parts:

$$\begin{aligned} g_e(z) &= c_1 + c_3 z^{-1} + c_5 z^{-2} + c_7 z^{-3}, \\ g_0(z) &= c_0 z + c_2 + c_4 z^{-1} + c_6 z^{-2}, \\ h_e(z) &= -c_6 z^2 - c_4 z - c_2 - c_0 z^{-1}, \\ h_o(z) &= c_7 z^3 + c_5 z^2 + c_3 z + c_1. \end{aligned}$$

Hence, the polyphase matrix is

$$P(z) = \begin{pmatrix} g_e(z) & h_e(z) \\ g_o(z) & h_o(z) \end{pmatrix} = \begin{pmatrix} c_1 + c_3 z^{-1} + c_5 z^{-2} + c_7 z^{-3} & -c_6 z^2 - c_4 z - c_2 - c_0 z^{-1} \\ c_0 z + c_2 + c_4 z^{-1} + c_6 z^{-2} & c_7 z^3 + c_5 z^2 + c_3 z + c_1 \end{pmatrix}.$$

The determinant of this matrix is

$$\begin{aligned} |P(z)| =& z^{-3}\left[\left(\sum_{l=0}^{1} c_l c_{l+6}\right) z^6 + \left(\sum_{l=0}^{3} c_l c_{l+4}\right)(z + z^5)\right] + z^{-3}\left[\left(\sum_{l=0}^{5} c_l c_{l+2}\right)(z^2 + z^4)\right] \\ &+ z^{-3}\left[\left(\sum_{l=0}^{7} (c_l)^2\right) z^3 + \left(\sum_{l=0}^{1} c_l c_{l+6}\right)\right] \\ =& \frac{0 \times z^6 + 0 \times (z + z^5) + 0 \times (z^2 + z^4) + z^3 + 0}{z^3} \\ =& 1, \end{aligned}$$

by Equations (2.17b) and (2.17c) from page 27. Now that we know that our polyphase matrix has determinant one, we can proceed to factor our scaling filter using the Euclidean Algorithm. This can be found in the appendices. We find a factorisation in terms of new coefficients q_{jk} (from successive re-parameterisation) and K (the gcd) that are polynomial expressions of our coefficients $c_i, i = 0, \ldots, 7$. Using this factorisation, we can form a filter h^0 complementary to g:

$$
\begin{aligned}
P^0(z) &= \begin{pmatrix} g_e(z) & h_e^0(z) \\ g_o(z) & h_o^0(z) \end{pmatrix} \\
&= \begin{pmatrix} q_{11} + q_{12}z^{-1} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_{21} + q_{22}z^{-1} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_{31} + q_{32}z^{-1} & 1 \\ 1 & 0 \end{pmatrix} \\
&\quad \times \begin{pmatrix} q_{41}z + q_{42} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & K^{-1} \end{pmatrix}.
\end{aligned}
$$

Then we lift the complementary filter polynomial $h^0(z)$ to $h(z)$ by solving

$$h_e(z) = h_e^0(z) + s(z)g_e(z),$$

for $s(z)$, giving

$$s(z) = \gamma_2 z^2 + \gamma_1 z + \gamma_0. \tag{3.28}$$

We include full polynomial expansion of the γ_i in terms of the original wavelet coefficients c_i in Appendix B as they are particularly extensive.

In order to obtain the lifting step polynomial (3.28) we used Maple (Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. 2015) to factorise an un-simplified version of (3.28) in terms of the coefficients $c_i, i = 0, \ldots, 7$. This returned a fraction of the form

$$s_{frac}(z) = \frac{\alpha_5 z^5 + \alpha_4 z^4 + \alpha_3 z^3 + \alpha_2 z^2 + \alpha_1 z + \alpha_0}{\beta_3 z^3 + \beta_2 z^2 + \beta_1 z + \beta_0} = \frac{A(z)}{B(z)},$$

where each α_j and β_j are polynomial expressions of the c_i. We set $A(z)$ and $B(z)$ to be the numerator and denominator, respectively, of $s_{frac}(z)$. Numerically plotting $s_{frac}(z)$ gave something that appeared to be a quadratic and three of the numerical roots of polynomial $A(z)$ were very similar to those of $B(z)$, but the massive expansion in terms of the c_i were not easily simplified. Having picked out the quadratic polynomial at the start of the power series expansion of $s_{frac}(z)$, we were eventually able to show that the remainder was zero by simplifying, using a Gröbner basis defined using various orthogonality properties of the coefficients.

We now have our factorisation of $P(z)$:

$$
\begin{aligned}
P(z) &= \begin{pmatrix} 1 & q_{11} + q_{12}z^{-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ q_{21} + q_{22}z^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & q_{31} + q_{32}z^{-1} \\ 0 & 1 \end{pmatrix} \\
&\quad \times \begin{pmatrix} 1 & 0 \\ q_{41}z + q_{42} & 1 \end{pmatrix} \begin{pmatrix} 1 & K^2(\gamma_2 z^2 + \gamma_1 z + \gamma_0) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} K & 0 \\ 0 & 1/K \end{pmatrix}.
\end{aligned}
$$

Finally, we give our lifted LA8 algorithm in Algorithm 4 with coefficients from Table 3.2.

We note that we do not plot the lifting coefficients, as they are not used in filters; rather, they

Algorithm 4: Least Asymmetric 8 filter lifting algorithm

$$
\begin{aligned}
d_l^{(0)} &= x_{2l+1} + q_{11}x_{2l} + q_{12}x_{2l+2} \\
s_l^{(0)} &= x_{2l} + q_{21}d_l^{(0)} + q_{22}d_{l+1}^{(0)} \\
d_l^{(1)} &= d_l^{(0)} + q_{31}s_l^{(0)} + q_{32}s_{l+1}^{(0)} \\
s_l^{(1)} &= s_l^{(0)} + q_{41}d_{l-1}^{(1)} + q_{42}d_l^{(1)} \\
d_l^{(2)} &= d_l^{(1)} + K^2\gamma_2 s_{l-2}^{(1)} + K^2\gamma_1 s_{l-1}^{(1)} + K^2\gamma_0 s_l^{(1)} \\
s_l^{(2)} &= K s_l^{(1)} \\
d_l^{(3)} &= K^{-1} d_l^{(2)}
\end{aligned}
$$

Coefficient	Approximate value
q_{11}	−5.254527458732594e+1
q_{12}	−2.556583965520256e+0
q_{21}	1.784231379329362e−2
q_{22}	−7.110054749942946e−4
q_{31}	8.411184074671726e+2
q_{32}	5.498294188059471e+2
q_{41}	−3.361060977406985e−3
q_{42}	2.289462248002221e−2
$\gamma_2 K^2$	−8.434130678687068e−1
$\gamma_1 K^2$	−2.942429943568852e+0
$\gamma_0 K^2$	−4.367837909852400e+1
K	1.408227631773041e+0
K^{-1}	7.101124686361548e−1

Table 3.2: LA8 lifting coefficients

allow us to obtain the DWT wavelet coefficients via different algorithms.

Chapter 4

CUDA

We have developed an R library for wavelet analysis using Graphics Processor Units (GPUs). One such programming language for working on GPUs is CUDA C, which can be run within R. Whilst the DWT already has a *fast* serial algorithm ($\mathcal{O}(n)$), the pyramid algorithm (Mallat 1989*a*), we find that we can achieve speed increases for large DWTs, for MODWTs and for computationally intensive schemes like cross validation.

In this chapter, we will give an introduction to CUDA and CUDA C. We give brief introductions to the CUDA architecture and how this maps to the code. In Chapter 5, we will give details of our implementations for the WaveCUDA software package. Most of the material in this chapter is based on NVIDIA Corporation (2014*a*), which is an excellent resource for starting to code in CUDA C. NVIDIA also provides many online resources for understanding the subtleties of CUDA programming. We base our computer architecture definitions on material from Page (2009).

The statistically-minded reader is free to skip this chapter, although it will be useful for an introduction to CUDA and for understanding the intricacies of optimisation in our software.

4.1 Introduction

CUDA is an acronym for *Compute Unified Device Architecture*. It provides capability to program on a graphics card using a markup language for C/C++, Java and Fortran, amongst other languages. CUDA enables programmers to use their graphics card as a GPGPU (*General Purpose Graphics Processor Unit*) without having to work at a low level or know its precise architecture. CUDA was introduced in 2006 by NVIDIA as GPU programming was growing. We note that CUDA code can only be run on compatible NVIDIA graphics cards, however, this has not inhibited its uptake in the academic community due to its performance. An alternative to CUDA is OpenCL, which is an open GPGPU language working on CPUs and GPUs and is not

limited to one vendor. Graphics cards have enjoyed a substantial improvement in the last few years, in keeping with *Moore's Law* (NVIDIA Corporation 2014*a*), which conjectured that the number of transistors on a microchip would double roughly every two years. This is partly due to increasing demand for powerful graphics from the gaming community and elsewhere. CUDA uses explicit parallel programming, exploiting the highly parallel nature of graphics cards. It is worth noting at this point that wavelet transforms can be performed in parallel and in place using the Lifting Scheme which we introduced in Chapter 3. We may measure performance in computing using simply time or FLOPS. We also note that in order to obtain speedups in CUDA code, we need a task which is sufficiently time consuming and resource consuming.

Remark 4.1.1. *FLOPS, or Floating Point Operations per Second are a measure of speed of processors. To calculate the number of FLOPS, we count the number of additions, subtractions, multiplications and divisions performed between floating point data types: in C, these are called* ***float*** *and* ***(long) double****, where a* ***double*** *type is stored in twice the number of bits as a* ***float****. As such, operations between doubles are slower, and typically FLOPS speeds are quoted for both types. We note also that divisions take longer than the other operations (in fact, on our Intel CPUs, for single and precision, the latency of division is roughly 3-4 times that of addition or subtraction (Intel 2016, Table C-15)), although they are all called floating point operations: as such, one tries to minimise the number of divisions in code.*

Remark 4.1.2. *Performance at processor level can be measured using* ***latency*** *(the time delay before an input can be processed on) and* ***bandwidth*** *or* ***throughput*** *(the rate at which new inputs can be offered) (Page 2009).*

Remark 4.1.3. ***Parallelism*** *is carrying out several sub-tasks synchronously to achieve a larger task. A loop is* ***parallelisable*** *if it can be executed in any order.*

4.2 CUDA Architecture

GPUs enjoy these speedups over CPUs due to their architectures, which are more suited to parallel programming, as we can see in Figure 4.1 (NVIDIA Corporation 2014*a*).

- An *ALU* (Arithmetic Logic Unit) is what performs computations. There are far more of these in the GPU, but their smaller size in the diagram represents their relative performance. More of the GPU is dedicated to ALUs, but at a sacrifice of the other components.
- The *Control* controls the behaviour of all components on a processor. On the GPU, there are fewer resources committed to flow control, so whilst the number of operations that can be carried out is very high, it cannot cope with operations of different natures at once.
- The *cache* is a small, temporary, very fast-access memory location. It is often pre-loaded with memory on the CPU to avoid latency issues of reading from global memory. With

GPUs, we can hide this latency by more calculations. Processors also have registers which are much smaller than caches and are even faster access.

- The *DRAM* (Dynamic Random Access Memory) is a larger body of memory not located on the processor which is hence slower to access. This is what is known colloquially as RAM, and forms the main part of volatile memory.

A comparison of some relevant technical specifications of computers that we program with is displayed in Table 4.1. These specifications highlight how the GPUs are geared to carrying out parallel tasks. Let us give a little more insight into the various fields:

- Cores: Each CPU core has dedicated control, cache and ALU. These GPU cores are divided between controls (called Streaming Multiprocessors, SMs) which each control 192 cores.
- DRAM: to illustrate the limitations of the smallest memory size here (2GB), this could theoretically store a maximum size of 2^{28} of a vector of type double. The CPUs and GPUs roughly have DRAM sizes of the same order.
- Cache: 256KB could theoretically store 2^{15} doubles. We note that the GPUs have significantly less cache memory.
- Clock speed: Clocks in digital circuits are signals that oscillate between 0 and 1 in a regular fashion to synchronise the events within the circuit. The clock speed or clock rate is the number of clock cycles per second. Every instruction (e.g. addition, multiplication) may require several clock cycles. Simply put, the higher the clock speed, the faster we can do computation.
- Memory bandwidth: This is the amount of memory that can be transferred per second. Note that the GPUs have higher bandwidth to make up for less fast-access memory.

4.3 CUDA C

CUDA extends the C/C++ programming language with markups that are more explicit parallelism than OpenMP [1]. We write "C/C++", because whilst it is intended to be an extension of C, it uses some functionality of C++ not in C. In fact, CPU code can use full C++ syntax, whilst GPU code can only use a subset of this (NVIDIA Corporation 2014*a*). A program is run from the CPU (called the *host*), and specific tasks are run on the GPU (called the *device*). Any memory used on the device has to be allocated specifically on the device, which means that memory used by both host and device has to be allocated on both processors, and transferred

[1] openmp.org OpenMP uses compiler directives to specify regions of code for parallelism

	Intel CPUs		NVIDIA GPUs	
	i7-3740QM	**E5-1650 v2**	**GeForce 650M**	**Titan**
Cores	4[1]	6[2]	384	2688
RAM	16GB[3]	16GB[4]	2GB	6GB
Cache	6MB	12MB	256KB[5]	1.5MB[5]
Clock speed	2.7 GHz[6]	3.5GHz[7]	0.83GHz	0.88GHz
Memory bandwidth	25GB/s	60GB/s	80GB/s	290GB/s

[1] 8 threads, using *hyper-threading* technology, which aims to redistribute the load of these 8 threads over the 4 cores
[2] 12 threads
[3] in my configuration; this CPU supports up to 32GB of RAM
[4] this CPU supports up to 256GB of RAM
[5] this is "L2" cache, which is shared by all the multiprocessors. As well as this cache, there is also 64KB of constant memory for the device and 64KB per multiprocessor for shared memory and L1 Cache
[6] this can be overclocked to 3.7GHz
[7] this can be overclocked to 3.9GHz

Table 4.1: A comparison of the specification of our CPUs and GPUs

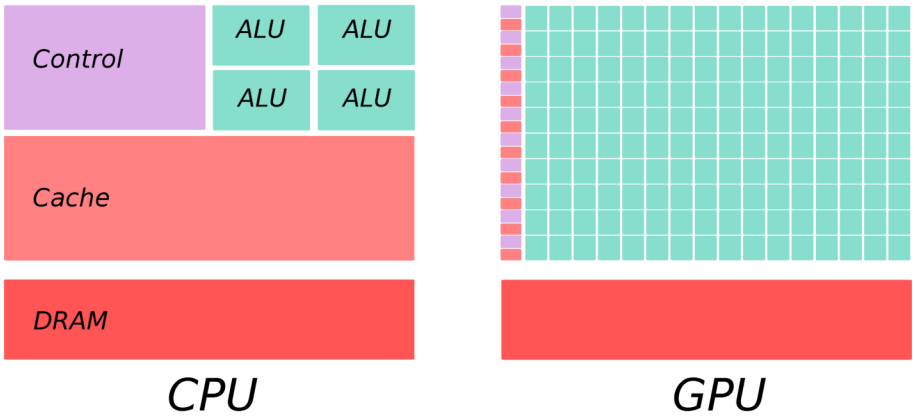


Figure 4.1: Comparison of GPU and CPU architectures. Diagram reproduced from NVIDIA Corporation (2014*a*).

between the two. This memory transfer is time consuming, so must be offset by time gains on the device; moreover, we might also transfer memory in chunks so that we might work on one part whilst transferring another part.

In CUDA C, functions to be performed in parallel by the graphics card are called *kernels*. These use the following syntax:

```
__global__ void KernelName(type variable);
```

To call a kernel, a triple chevron notation is used (the last two inputs in the chevrons are optional):

```
KernelName<<<NumBlocksPerGrid,ThreadsPerBlock,SharedMemPerBlock,Stream>>>(variables);
```

This illustrates how the explicit parallelism is programmed: the *threads* are tasks that are executed in parallel in a *SIMT* (Single Instruction, Multiple Thread) fashion. There are multiple threads in a *block*. It is also worth noting that both `NumBlocksPerGrid` and `ThreadsPerBlock` are `dim3` types, which means that they can be 3-dimensional, 2-dimensional or 1-dimensional. There are also multiple blocks in a *grid*. Let us illustrate a simple kernel that copies data between vectors of length `len` on a GPU:

```
__global__ void copyveccu(const real* from_d, real* to_d, uint len){
  // note that we define type uint to be an unsigned integer
  // and real to be a double
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
  // for(i=0;i<len;i++) to[i]=from[i];
  if(i < len) to_d[i] = from_d[i];
}
```

Each thread is identified by `threadIdx.x`, `blockIdx.x` and `blockDim.x`: `threadIdx.x` is the number of the thread in its block, from 0 to `ThreadsPerBlock - 1`, `blockIdx.x` is the number of the block in its grid, and `blockDim.x` is the size of the grid, set by `NumBlocksPerGrid`. The number of threads in a block can be up to 1024 on our GPUs, whilst the number of blocks per grid is determined by the number of processors and the size of the data being worked upon. For example, a likely scenario might be if we are working on a vector, then each thread might access one element of the vector and hence we will need a total number of threads greater than or equal to the size of the vector. The number of blocks and threads per block is subject to optimisation of dividing the data, mapping to appropriate hardware dimensions and scheduling to try to keep the processor occupied and 'hide' memory transfers. In this example, we might consider the following values:

```
#define BLOCK_SIZE 256

// ...

int ThreadsPerBlock = BLOCK_SIZE;
int NumBlocksPerGrid =(len + ThreadsPerBlock - 1) / ThreadsPerBlock;
```

We note here than in C, division between integers is integer division, so this value of `NumBlocksPerGrid` ensures that we have at least enough threads to reference each element of our vectors.

Memory transfers between CPU and GPU memory are slow, and so we want to ideally 'hide' these in scheduling or minimise them. Within-GPU memory transfers are faster, but varying at different hardware levels, i.e. the latency on the GPU of accessing data on the DRAM (global memory) is much higher than on the cache or shared memory. Shared memory in CUDA is very fast, but is only shared between threads in a block, whilst all threads have access to global memory. Shared memory also only has the lifetime of a block: after that block has finished executing a kernel, its shared memory is no longer accessible. In practice, kernels will read values from global memory, transfer them to shared memory for intermediate calculations then

load the results back into global memory. We give a simple example using one level of the Haar transform in Listing C.7.

Note that the `__syncthreads();` call (commented out) in the above code sample is a thread barrier: it prevents threads from continuing to the subsequent lines until all threads have reached the synchronisation point to prevent them from for example accessing memory that hasn't yet been updated (this is called a race condition). In this particular example, the commented thread barrier is not actually required, but they are usually used after loading values into shared memory. In fact, this would cause an unnecessary stall in execution.

The threads, blocks and grids do not map explicitly onto the architecture of a GPU; rather, they represent the parallelism of a task. A *Streaming Multiprocessor* (SM) (of which there are 14 on our Titan GPU, and each SM controls 192 cores) schedules tasks to be carried out on the cores by groups of 32 threads called *warps*. Since threads are divided into *blocks* in the code, it is often desirable to have a block size that is a multiple of 32, the warp size. This scheduling is done dynamically by the GPU with the aim of keeping the cores occupied and hiding latency. It cannot be controlled by the user, but it can be facilitated for example by optimising block size and grid size and minimising memory transfers between locations with highest latency.

4.3.1 Optimisation in CUDA

We have already discussed how GPUs are suited to *SIMT* programming, and where they can be used to give speedups. In order to significantly improve running time, we need to dedicate considerable time to optimisation. Some important optimisations that we have implemented are below.

- Block size: As mentioned above, the number of threads per block is key to mapping constraints of a particular programming problem to memory and hardware constraints. For wavelet transforms, we would like to work in powers of two that both divide up our vectors and are factors of memory sizes. In reality, however, we need to use block sizes that are a power of two plus some spaces for boundary coefficients. Another recommendation is to use enough threads to saturate the memory bandwidth.
- Shared memory: Shared memory should be used to make use of GPUs' low memory latency on-chip. In our wavelet transforms, this is the optimisation that led to the largest incremental speedup. Moreover, in the case of the Haar transform, we were able to write a single kernel that performs two levels of transform as the improvements from using shared memory less the cost of transferring memory between memory levels outweighed issues of thread redundancy.
- Thread divergence and redundancy: When threads within a warp execute different branches of conditional code, we have thread divergence. This causes the execution to be serial.

Redundancy is leaving allocated threads idle, and could be the result of stalls from thread barriers. Hence, both thread divergence and redundancy should be avoided.

- Streaming and latency hiding: Streams allow programs to fully utilise hardware resources by increasing the amount of parallelisation in *streams*. Streams can also enable certain sections of programs to be run at the same time as other sections are transferring memory, which is called latency hiding.

Chapter 5

WaveCUDA

In this chapter, we present a new R package called WaveCUDA, which provides fast, CUDA-accelerated wavelet analysis for R users with compatible NVIDIA graphics cards. Also included in the package are standard CPU implementations of wavelet transforms. The GPU implementations are particularly useful for computationally expensive methods such as cross validation in Chapter 6 and optimal gridding searches for irregularly spaced data in Chapter 7.

This chapter starts with an algorithmic introduction to wavelet transforms, followed by the fast implementations of transforms and thresholding in our package. We then introduce the user functionality of WaveCUDA and finally we demonstrate the main advantage of the package via benchmarking.

There are existing R packages for wavelet analysis, notably:

- wavethresh (Nason 2013*b*): This is a useful package focusing on wavelet thresholding which has many excellent features including good graphics.
- wmtsa (Constantine & Percival 2013): This is the package accompanying the book by Percival & Walden (2000). It provides many features for wavelet analysis, and was written with more of an emphasis on rigour than speed.
- waveslim (Whitcher 2013): This is the package accompanying the book by Gencay, Selçuk & Whitcher (2001). It is a lightweight package with more of an emphasis on speed and simplicity.

However, none of these package use CUDA-acceleration. In WaveCUDA, we provide faster functionality for some of the more computationally intensive wavelet procedures. There are existing R packages using CUDA (but not wavelets), such as:

- gputools (Buckner, Seligman & Wilson 2013): This package provides an interface to some modelling and matrix GPU code.

- HiPlarm (Montana, Nash & Szeremi 2013): This package is High Performance Linear Algebra in R for the Matrix package.

GPU implementations of wavelets pre-date CUDA with highly technical papers by Jiang & Ortega (1999) and Tenllado & Setoain (2008) (the latter paper was published after the initial release of CUDA). These pre-CUDA papers lack the functionality of CUDA and so are limited in their optimisation options. Van Der Laan et al. (2011) appear to be the first to implement lifting-based wavelet transforms using CUDA. They produce impressive speedups, and focus on 2D and 3D transforms where the parallelism can be exploited even more. However, they do not carry out their operations in-place in memory and so effectively use double the necessary memory. Moreover, this is a little outdated as NVidia graphics cards and CUDA versions have new improved features. There have also been subsequent publications on wavelets with CUDA. Zhu, Yadong, Zhang, Wang, Wang & Wang (2015) claim significant speedups without using lifting but compare their timing results with a slow benchmark of a serial CPU implementation involving FFT steps. Ikuzawa, Ino & Hagihara (2016) modify Van Der Laan et al.'s (2011) data structures to give coalesced coefficients for improved data transfers, but the preprocessing in this method make it slower than previous code if the data needs to be transferred from and then back to the CPU. Quan & Jeong (2016) implement a number of the same optimisations as we have made, in a 2D setting, and claim to have the quicker running times than Van Der Laan et al. (2011). So far, we have focused on 1D wavelet transforms, also implementing efficient functions for MODWT transforms, thresholding and functions combining these operations. However, unlike the references given above, we have written our code for the use of statisticians working in R.

WaveCUDA was built using devtools (Wickham, Hester & Chang 2018) and documented with roxygen2 (Wickham, Danenberg & Eugster 2018). The plotting in the package uses ggplot2 (Wickham 2016), the efficient table building uses data.table (Dowle & Srinivasan 2018) and we also include a function for converting to wavethresh class (Nason 2013*b*).

5.1 Algorithmic introduction to wavelets

Standard DWT Implementation

In WaveCUDA, as well as CUDA-accelerated implementations, we also provide standard DWT implementations on the CPU. The standard DWT is typically implemented using the pyramid algorithm which we introduce in Section 2.2.2. Here we implement the DWT serially using a moving window with a recursive function to perform the pyramid algorithm and store the results in-place. Now let us dissect this statement:

- Serial: Executing loops in order, one by one, i.e. not in parallel.

- Moving window: We convolve a filter with a vector starting at the beginning of the vector and continuing sequentially.
- Recursive function: A recursive function is one that calls itself. In C, this is implemented by returning the result of another function call.
- In-place: The vector input of the function is overwritten to form the output.

We illustrate this with a simple implementation of the Haar forward transform in Listing C.1, which can be found in the appendix. We initially run this function with a value of `skip` of 1. This variable is then doubled after each level's transform to provide the decimation and downsampling. The full transform has been calculated after $\log n$ levels when `skip` reaches n (where n is the length of the input vector, and we note here that in the code we use `len` instead), although we control the depth of the transform with the variable `nlevels`. `invR2` is $1/\sqrt{(2)}$. Our memory structure is illustrated in Listing C.2. Let us also consider the D4 forward transform for comparison with a longer filter, in Listing C.3. Clearly, the D4 filter is more involved because it requires special treatment at the boundaries. We implement periodic boundary conditions, i.e. we assume that the input vector is periodic so the filter is wrapped around back to the start when it reaches the end. We have standard DWT CPU implementations of the Haar, D4, C6 and LA8 filters.

These can be run in WaveCUDA with the following:

```
w1 <- CPUTransform(x1, direction = "FWD", nlevels = 0, transform.type = "DWT",
                   filter = "Haar")
w2 <- CPUTransform(x2, direction = "FWD", nlevels = 0, transform.type = "DWT", filter = "D4")
```

MODWT Implementations

We have implemented the MODWT for only the Haar filter on both CPU and GPU. We stated in Section 2.5 that the MODWT retains n detail and scaling coefficients at each level. One consequence of this is that it cannot be implemented with in-place memory transactions. Therefore we decided to implement this MODWT retaining all scaling and detail coefficients, which for a full transform requires an array of size $2n(\log_2 n)$. There are two options for computing the MODWT: packet-ordered and time-ordered. Our packet-ordered transform recycles the Haar DWT code for running single levels of packets in-place on shifted copies of the latest scaling coefficients. This is best illustrated in Listing C.5. The time-ordered transform does not rely on copying and shifting the coefficients but rather reads coefficients from the previous level and writes to the current level. We illustrate the workings of the time-ordered Haar MODWT in Listing C.6. We note that we can convert between time and packet orderings, as the transforms produce the same coefficients just in different orders.

The Haar MODWT does require periodic boundary conditions unlike the DWT because it

doesn't decimate. We handle this in the packet ordered code by circularly shifting the copies of the latest scaling coefficients. In the time ordered code, we need only consider modular arithmetic.

We have also implemented a CPU version of the D4 MODWT with lifting as a proof of concept. Both packet-ordered and time-ordered versions translate simply to the DWT lifting functions just as we adapted our standard DWT functions.

We can run the standard MODWT implementations as follows:

```
w1 <- CPUTransform(x1, direction = "FWD", nlevels = 0, transform.type = "MODWT",
                   filter = "Haar")
w2 <- CPUTransform(x2, direction = "FWD", nlevels = 0, transform.type = "MODWT",
                   filter = "D4")
```

In the remainder of this chapter we present the CUDA versions of the wavelet filters, illustrate the thresholding functions implemented in WaveCUDA then demonstrate example usage of the package. This example usage includes plotting, reconstructions, an illustration of the `WST` class, converting to data table and converting to a `wavethresh` object. Lastly, we benchmark example usage for timing.

5.1.1 Parallelising the Wavelet Transform

We note that the DWT as calculated in the standard implementation is immediately parallelisable if we change it from being in-place to using separate input and output vectors. The Haar transform is even parallelisable with in-place memory treatment since its filter length is two and so there is no overlap of the moving windows. As we typically require large vectors to see benefits of CUDA implementations, it makes sense to seek in-place algorithms. The lifting scheme as described in Chapter 3 provides an alternative method of performing DWTs which allows for independent iterations and can be performed in-place in memory: this is ripe for parallelisation. It also has the added benefit of typically reducing the number of floating point operations.

Let us now consider a lifting version of the D4 forward transform, in Listing C.4. Note that it retains the same memory structure and recursive structure. In C, we can parallelise this transform by inserting `#pragma omp parallel for private (i)` statements before each for loop. We have lifted DWT CPU implementations of the Haar, D4, C6 and LA8 filters.

5.2 CUDA-Accelerated Wavelet Transforms

The CUDA wavelet transforms in WaveCUDA are coded using optimised CUDA kernels run by wrapper functions that set up the CUDA variables and control the recursion.

5.2.1 Haar DWT

We have shown a simple example of a Haar DWT kernel using shared memory in Listing C.7. However, we can further optimise this using a kernel that calculates more than one level of transform per kernel in Listing C.8. This sacrifices some redundancy to reduce the number of memory transfers between global and shared memory. This lends itself very well to a CUDA implementation because the intermediate calculations can be done in shared memory and we do not have any thread divergence. However, as we go deeper into the wavelet transform, we run this kernel with larger values of `skip` and fewer threads, all the way down to two threads for the final level: at this point, we are using the GPU's resources very badly.

The CUDA-accelerated Haar transform can be run with a small modification to the code above:

```
w <- GPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "DWT", filter = "Haar")
```

5.2.2 DWTs with Longer Filters

Longer filters give rise to more complicated implementations, as we have to worry about boundary conditions not just at the ends of the signal but also at every point that we divide it into shared memory. This also means that the implementations are slower, as all of the boundary coefficients have to be found and written into a separate global memory vector before we can start the transform for a level. See Listing C.9 for a diagram of our shared memory structure for a kernel performing two levels of transform. It is clear from Equations (3.24) and (3.25) on page 61 that a single level of transform will require us to handle boundary conditions of two extra coefficients at each end of the signal and at each instance that we divide the code into shared memory blocks. For a kernel that performs two levels of transform, we then need to also pick out extra boundary coefficients for the second level of transform. The number of extra coefficients for a double level transform kernel has to be calculated from careful consideration of the lifting algorithm (given in Algorithm 3); in the case of C6 we need 6 extra coefficients at each end. We provide a CUDA kernel that performs two levels of the lifted C6 DWT in Listing C.10.

We note that the C6 lifting algorithm is comparatively nicer to implement than the LA8, as our LA8 lifting algorithm uses Laurent polynomials that are significantly out of phase: indeed, for a double level lifted LA8 DWT kernel we need 24 extra boundary coefficients at the start and 20 at the end! We have also implemented the lifted D4 filter in CUDA.

We can run the GPU version of the C6 DWT, for example:

```
w <- GPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "DWT", filter = "C6")
```

5.2.3 Haar MODWT

We have implemented CUDA-accelerated Haar MODWT kernels, both time-ordered and packet-ordered. The time-ordered version performs well for a couple of reasons: we do not lose half of the CUDA threads each level of the transform because it produces n detail and scaling coefficients at each level without decimation; and we do not need to copy any memory or handle boundary conditions bar with modular arithmetic. On the other hand, the packet-ordered version is incredibly slow. We run individual CUDA Haar DWT kernels for each packet. At the end of the transform, we are running $n/2$ packets each with two threads. Copying and circular shifting chunks of global memory is time consuming. Consequently, if we require packet-ordered coefficients, it is cheaper to compute the time-ordered coefficients then re-order them.

The GPU-accelerated Haar MODWT is calculated in WaveCUDA using:

```
w <- GPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "MODWT",
                  filter = "Haar")
```

5.3 Thresholding and Smoothing

Thresholding is straightforward to implement both on CPUs and GPUs. We have implemented both hard and soft thresholding. This is not a time-consuming task, so it is not worth transferring data onto the GPU for thresholding, however this is worth doing if the data are already situated in the GPU global memory. Furthermore, we properly utilise the resources of the GPU with computationally-intensive thresholding schemes such as cross validation.

As such, we only support thresholding as a stand-alone function in CPU code. The interested reader should consider Nason (2013*b*) for extensive thresholding schemes, for which we provide a utility function. Instead, we focus on "smoothing" on the GPU which incorporates a wavelet transform, threshold then reconstruction which minimises memory copies between CPU and GPU.

```
w_thresholded <- CPUThreshold(w, thresh = 0.5, hard.soft = "hard", min.level = 1,
                              max.level = 3)
## threshold a wavelet transformed vector
w_wavethresh <- WST.to.wavethresh(w)
## convert a WaveCUDA wavelet object to a Wavethresh object
```

5.3.1 Wavelet Cross Validation

We have implemented Nason's (1996) two-fold cross validation as it is one of the more computationally intensive uses of 1D wavelets. We describe the cross validation method in Section

6.3.1. Here, it suffices to say that it attempts to find an optimal threshold using the following steps:

- First, wavelet transform the data and calculate the universal threshold for an upper bound.
- Then divide the data into two subsets and repeatedly threshold, comparing thresholded data of each subset to the original values of the other.
- Repeat the above step until we converge on a threshold based on minimising the predictive squared error.

We found the first step to be time consuming on the GPU when using the usual MAD estimate of standard deviation of the wavelet detail coefficients. Instead we calculate the regular sample standard deviation, which is likely to be higher. However, this is not an issue, because this just requires a few extra iterations of the second step. Moreover, with our efficient transforms, the end result is faster. In the second step, we can use streams to fully load the GPU. All the above steps can be carried out on the GPU without having to copy memory back and forth until we return the thresholded data. Thus, we do enough work to reap the benefits of the GPU's performance despite the time taken to transfer memory across.

We note that the cross validation function can be run with hard or soft thresholding, on a specified subset of levels and using any of the implemented wavelet transforms.

```
x_smoothed <- GPUSmooth(x, nlevels = 10, transform.type = "DWT", filter = "Haar",
                        thresh.type = "cv", hard.soft = "soft", min.level = 1,
                        max.level = 10)
```

5.4 Using WaveCUDA

In this section, we demonstrate some simple usage of the package. The package is hosted on GitHub at `https://github.com/wavecuda/wavecuda`. We will take a signal, transform it with three different filters and then show some further functions.

To begin with, we need to load the library, our plotting library and generate the signal (which uses a third library).

```
> library(wavecuda)
> library(ggplot2)
>
> set.seed(10)
> x <- wmtsa::make.signal("doppler", n = 1024, snr = 5)
> x <- x@data
```

Next, we will generate three different transforms on the same signal. We will also demonstrate reconstruction. We can reconstruct with the `CPUTransform` and `GPUTransform` functions

(`direction = "BWD"`) taking a vector input, however the reconstruction functions are more convenient. Notice that we can do a forward transformation with the CPU but a backward transformation with the GPU. This is because GPU memory is allocated and freed within any function using the GPU.

```
> w1 <- CPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "DWT",
+                    filter = "D4")
>
> w2 <- GPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "DWT",
+                    filter = "C6")
>
> w3 <- GPUTransform(x, direction = "FWD", nlevels = 0, transform.type = "MODWT",
+                    filter = "Haar")
> xReconstructed <- GPUReconstruct(w3)
```

Let us first do a quick sanity check of the reconstruction.

```
> all.equal(xReconstructed, x)
[1] TRUE
```

Indeed, we find that the reconstructed vector is the same. Let us now demonstrate the plotting in WaveCUDA. We produced Figure 5.1 using the code listing below.

```
> p <- plot(w1)
> p <- p + theme_grey(base_size = 14)
>
> ggsave("wavecuda_plot_D4.pdf", p)
Saving 7 x 7 in image
```

Next, we demonstrate the structure that we return from a wavelet transform. It is a list structure containing information on the wavelet transform as well as the input vector and the transformed vector. This allows the user to easily access the level and translate information for any coefficient.

```
> w2
-------------------------------------
Wavecuda STructure object WST class
of type:                   DWT
with filter:               C6
levels of transform:       8
original vector of length: 1024
-------------------------------------
> str(w2)
List of 7
$ x      : num [1:1024] 0.00108 -0.02865 -0.0365 -0.08071 0.02553 ...
$ ttype  : chr "DWT"
$ filt   : chr "C6"
$ filtlen: int 6
$ nlevels: int 8
$ len    : int 1024
```

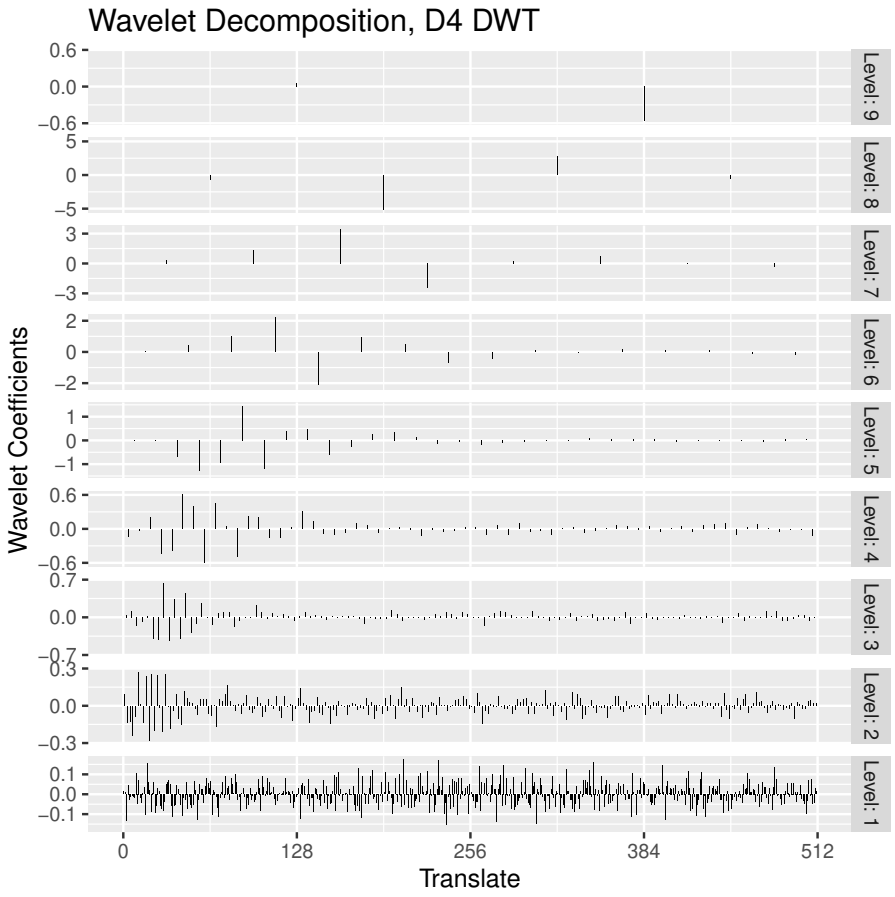


Figure 5.1: Graph demonstrating plotting in WaveCUDA. Graph illustrates the DWT decomposition of a noisy Doppler signal (SNR 5) using a D4 filter.

```
$ w      : num [1:1024] 0.69269 -0.00548 -0.06784 -0.06666 0.00173 ...
 - attr(*, "class")= chr "WST"
```

The best way to work with the wavelet transformed vector is with the thresholding and transformation functions. However, if the user wants to work with the raw coefficients, we supply a utility function to convert this into a data table.

```
> wdt3 <- WSTtoDT(w3)
> wdt3
                W CoeffType Level Translate
   1: -0.01949016         s     1         1
   2: -0.04606657         s     1         2
   3: -0.08287811         s     1         3
   4: -0.03901821         s     1         4
   5:  0.07027214         s     1         5
  ---
20476: -0.42898560         d    10      1020
20477: -0.44141062         d    10      1021
20478: -0.44702792         d    10      1022
```

```
20479: -0.46056430        d    10      1023
20480: -0.48332785        d    10      1024
```

5.4.1 Summary of Implementation

For an up to date account of the implementation of WaveCUDA, see the package repository and documentation on GitHub at `https://github.com/wavecuda/wavecuda`. The extent of the functionality at the time of writing is summarised in Table 5.1.

Functionality (filters)	**DWT**				**MODWT**	
	Haar	D4	C6	LA8	Haar	D4
CPUTransform	✓	✓	✓	✓	✓	✓
GPUTransform	✓	✓	✓	✓	✓	[1]
CPUSmooth (cv, univ, manual)	✓	✓	✓	✓	✓	
GPUSmooth (univ,[2] manual)	✓	✓	✓	✓	✓	
GPUSmooth (cv)	✓				✓	

Functionality (general, all implemented)
CPUReconstruct
GPUReconstruct
CPUThreshold
WSTtoDT
plot.WST
WSTtoWavethresh

[1] A proof of concept lifting MODWT D4 algorithm has been written but not implemented on the GPU

[2] Universal threshold uses standard deviation rather than median absolute deviation in the GPU version

Table 5.1: Summary of implementation of WaveCUDA. We note that the Haar transform does not require the lifting version, which is only subtly different, and that (with the exception of Haar) only the lifting scheme versions of the DWTs can be implemented on the GPU.

5.5 Benchmarking WaveCUDA

The main incentive for writing WaveCUDA was to achieve speedups in execution of wavelet code and computationally expensive derived code. In this section, we benchmark our C and CUDA implementations between themselves and also benchmark our WaveCUDA implementation of Nason's wavelet cross validation against wavethresh (Nason 2013*b*).

First, we compare the timings in C/CUDA for the wavelet transforms on our available machines: a laptop (ali) with Intel i7-3740QM CPU and NVIDIA GeForce 650M GPU, and a desktop (hpz) with Intel E5-1650 v2 CPU and NVIDIA Titan GPU. The hardware specifications can be found

in Table 4.1 on page 70. We compare the standard CPU versions, OpenMP CPU versions using the lifting scheme and our CUDA implementations which also use the lifting scheme. We compare the Haar, D4, C6 and LA8 DWTs and also the Haar MODWT (time-ordered). We are comparing the times directly without running from the R library and we do not include the time taken to transfer the data between the CPU and GPU for the CUDA times. This is because it constitutes a fairer test for comparing the transformation times. In practice, the R library should not be used for single transforms as the time to do memory copies may negate the benefit of computing on the GPU.

For the DWTs, we note that the fastest implementations for the largest sizes are the CUDA implementations and that the Titan GPU is by far the quickest. However, for smaller sizes of vector, the standard CPU implementations are quicker. All of the timing graphs for the Haar, D4, C6 and LA8 DWTs can be found in Appendix C in Figures C.1, C.2, C.3 and C.4. Each number in the graph represents the mean execution time from 100 transformations of a signal generated by sampling integers between 0 and `RAND_MAX`. We achieve a roughly 20× speedup on the desktop and closer to a 3× speedup on the laptop by using a CUDA-accelerated implementation of the Haar transform. The desktop GPU speedup reduces to almost 10× for our longest filter, the LA8. In Figure C.5, we plot the times taken to perform a time-ordered MODWT Haar transform on different hardware. Surprisingly, we notice that on the laptop, the OpenMP code is faster than the CUDA code. However, the desktop CUDA code is fastest, achieving a speedup of 20× over the serial CPU code and 8× over the parallel CPU code for the largest sizes of vector.

Next, we compare our CUDA-accelerated implementation of Nason's (1996) two-fold cross validation against the version in his package wavethresh (Nason 2013*b*). This is where we can really experience the advantages of the GPU. We use the Haar filter and both DWT and MODWT transform types. On this occasion, we use our WaveCUDA package for a fair test and we only run it on the desktop. Note that his package contains a bug in the interpolation part of this cross validation, so we produce different results and also converge after different numbers of iterations (even when taking into account our different upper bound). Moreover, there are various ways of running cross validation in the package wavethresh: we use the `CWCV` and `WstCV` functions, as they are fastest. We simulate noisy Doppler signals with SNR 10, and ensure the same signals are tested for each combination of transform and package. We average the timings from 5 - 100 runs, depending on the signal length. We plot the timing results in Figure 5.2.

In this case, we find our CUDA code gives some speedups of up to 100× for Haar MODWT and up to 27× for DWT.

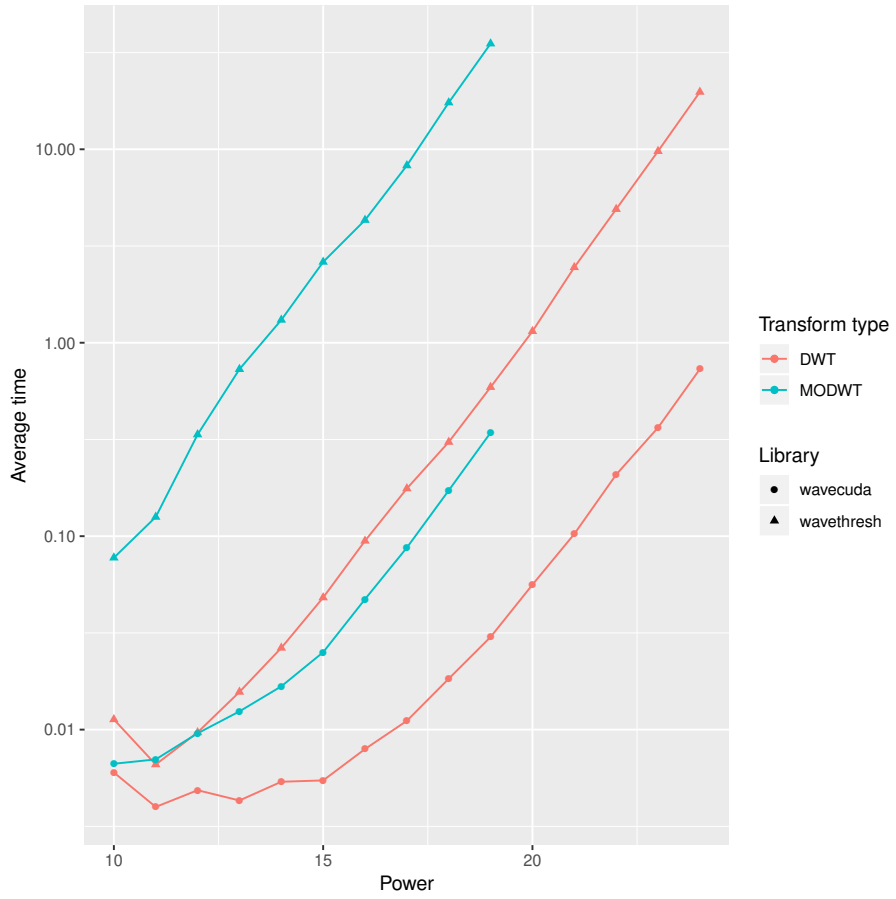


Figure 5.2: Graph of timings for Nason's cross validation using a Haar DWT and MODWT on a log scale. We compare the function in the R package wavethresh with WaveCUDA. We plot the average time for each transform from multiple runs of smoothing a Doppler signal of SNR 10.

Chapter 6

Wavelet Cross Validation

Cross validation (CV) can be used with wavelets for selection of an optimal threshold in signal estimation. CV is often a computationally-intensive task, and so it makes sense to have fast, parallel implementations of the required procedures. However, for special cases, it can be implemented efficiently.

In this chapter, we introduce cross validation in the context of linear regression. Then, after describing its application in autoregressive models, we address the subject of wavelet cross validation. Extending cross validation to wavelets is not immediate, because wavelet regression requires dyadic, equally-spaced data. Our contribution in this chapter is to introduce wavelet cross validation beyond the limited scopes of existing schemes. We introduce general delete-d and V-fold wavelet cross validation schemes, with regularisation, beyond the existing current methods which are limited to $V = 2$ and $d = 1, 2, 3$. This will allow us more granularity in our bias/variance trade-off for d or V in wavelet cross validation.

6.1 Standard Cross Validation

6.1.1 Introduction

CV is a potentially computationally-demanding method following the principle of splitting the data into training and validation sets. A model is first fitted or "trained" to the training set and then its prediction is tested on the validation set. This process of validation is then repeated over a number of splits of the data, and the model minimising the prediction error is selected. For good introductions to CV, see for example books by McQuarie & Tsai (1998), Efron (1982) or Hastie, Tibshirani & Friedman (2009), or a survey paper by Arlot & Celisse (2010). This section is based on Chapter 6 from McQuarie & Tsai (1998).

Let us formally describe CV via a linear regression model, as later on we will demonstrate how

this can lead to an efficient implementation.

$$\boldsymbol{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})^T$ is our response vector, $X = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n-1})$ is a matrix of regressors $\boldsymbol{x}_i$ each of length p, $\boldsymbol{\beta}$ is a parameter vector of length p and $\boldsymbol{\epsilon}$ is an vector of length n of i.i.d. noise with $\mathrm{E}\,\epsilon_i = 0$, $\mathrm{Var}\,[\epsilon_i] = \sigma^2$. The Least Squares (LS) estimate for $\boldsymbol{\beta}$ is

$$\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \boldsymbol{y}.$$

We would like to estimate the Mean Standard Error of Prediction (MSEP),

$$MSEP = \mathrm{E}\,(y_n - \hat{y}_n)^2, \tag{6.1}$$

for a new observation y_n which is *independent* of y_i, $i = 0, \ldots, n-1$. One possibly unbiased estimate is CV(1), the Cross-Validation criterion for one withheld observation, which is related to the PRESS (Predicted Residual Error Sum of Squares). This is also closely related to the jackknife, as noted by Efron (1982). Let

$$\hat{y}_{(i)} = \boldsymbol{x}_i^T \hat{\boldsymbol{\beta}}_{(i)}, \tag{6.2}$$

be the prediction of y_i based on all observations except i. Here, we use the standard notation of $\hat{y}_{(i)}$ and $\hat{\boldsymbol{\beta}}_{(i)}$ denoting the predicted value and estimated parameter vector, respectively, based on all but the i index observation, i.e. $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_{n-1})$. Now let $e_{(i)} = y_i - \hat{y}_{(i)}$. We can define PRESS as

$$PRESS = \sum_{i=0}^{n-1} e_{(i)}^2, \tag{6.3}$$

and CV(1) as

$$CV(1) = {}^1\!/\!_n\, PRESS. \tag{6.4}$$

CV with one withheld observation is also known as Leave One Out (LOO) CV. Clearly, there are other valid CV schemes with larger hold-out samples.

6.1.2 Delete-d

The Delete-d scheme was introduced by Shao (1993) to improve on the lack of consistency of CV(1). It leaves out d samples for validation rather than simply 1. Shao defines consistency of a CV model as being able to select the best predictive model with probability 1 as the total number of observations $n \to \infty$. Shao found instead that CV(d) is consistent if ${}^d\!/\!_n \to 1$ and

$n - d \to \infty$ as $n \to \infty$. He recommends for the choice of d

$$d = n\left(1 - \frac{1}{\log(n) - 1}\right) \quad \Longrightarrow \quad n - d = \frac{n}{\log(n) - 1}.$$

The cross validation criterion for delete-d is

$$CV(d) = \frac{1}{Md} \sum_{m=0}^{M-1} \sum_{i \notin \mathcal{M}_m} (y_i - \hat{y}_i^d)^2, \tag{6.5}$$

where $\hat{y}_i^d$ is the predicted value of y_i from training set $\mathcal{M}_m$ and M is the total number of training sets considered. The true $CV(d)$ estimate should be made from averaging over all possible choices of sets of d, however in practice we should only need to consider a subset of these for a reliable estimate.

6.1.3 V-fold

V-fold CV schemes (sometimes called K-fold) divide the date into V sets, with one for validation and the remaining $V - 1$ for training. The errors are then averaged over all V validation sets. The choice of V is a bias/variance trade-off: pick too high a value of V and the prediction error can have a high variance but low bias; or vice-versa for too low a value of V. $V = n$ gives the LOO scheme. $V = 5$ is generally accepted as a reasonable compromise, although this will clearly vary according to dataset (Hastie et al. 2009, Arlot & Lerasle 2014); indeed, in a wavelet setting we may want to pick V to be a power of two to suit our dyadic setting.

6.1.4 Computationally Efficient CV(1)

As previously mentioned, CV is not necessarily computationally expensive. Indeed, LOO CV for linear regression can be computed with rank-one updates and thus only needs one least squares estimation (e.g. (McQuarie & Tsai 1998)). The vector $\hat{\boldsymbol{\beta}}_{(i)}$ is calculated by

$$\hat{\boldsymbol{\beta}}_{(i)} = \left(X^T X - \boldsymbol{x}_i \boldsymbol{x}_i^T\right)\left(X^T \boldsymbol{y} - \boldsymbol{x}_i y_i\right).$$

The matrix inverse in this equation can be calculated using the Sherman-Morrison formula (see e.g. Petersen & Pedersen (2012)):

$$\left(A + \boldsymbol{b}\boldsymbol{c}^T\right)^{-1} = A^{-1} + \frac{A^{-1}\boldsymbol{b}\boldsymbol{c}^T A^{-1}}{1 - \boldsymbol{c}^T A^{-1}\boldsymbol{b}}, \tag{6.6}$$

for an $n \times n$ matrix A and vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ of length n. Using this result, we can write $\hat{\boldsymbol{\beta}}_{(i)}$ as

$$\hat{\boldsymbol{\beta}}_{(i)} = \left[\left(X^T X\right)^{-1} + \frac{\left(X^T X\right)^{-1} \boldsymbol{x}_i \boldsymbol{x}_i^T \left(X^T X\right)^{-1}}{1 - h_i} \right] \left(X^T \boldsymbol{y} - \boldsymbol{x}_i y_i\right), \tag{6.7}$$

where $h_i = H_{ii} = \boldsymbol{x}_i^T \left(X^T X\right)^{-1} \boldsymbol{x}_i$, a diagonal element from the hat matrix which we defined in Section A.1.1. Then, after some cancellation, we can write

$$e_{(i)} = \frac{e_i}{1 - h_i},$$

where $e_i = y_i - \boldsymbol{x}_i^T \boldsymbol{\beta}$. Thus, we can calculate CV(1) computationally efficiently:

$$CV(1) = \frac{1}{n} \sum_{i=0}^{n-1} \frac{e_i^2}{(1 - h_i)^2}. \tag{6.8}$$

This means that we only need compute the usual LS regression coefficients and the diagonals of the hat matrix $H = X(X^T X)^{-1} X^T$.

6.1.5 Generalised Cross Validation

Generalised Cross Validation (GCV) is an approximation of CV(1) introduced by Craven & Wahba (1979):

$$GCV = \frac{1}{n} \sum_{i=0}^{n-1} \frac{e_i^2}{(1 - tr(H)/n)^2}. \tag{6.9}$$

The trace of the matrix H may be easier to compute than each individual element h_i. This trace is the effective number of parameters or the effective degrees of freedom. The GCV has been found to undersmooth less than CV(1) (Hastie et al. 2009).

6.2 Cross Validation in Autoregressive models

CV has also been extended to univariate autoregression (see e.g. McQuarie & Tsai (1998)). Here, clearly, we lose our i.i.d. assumption on y_n. However, we can take steps to limit the damage of this. Let us take an $AR(p)$ model

$$y_t = \Phi_1 y_{t-1} + \cdots + \Phi_p y_{t-p} + \epsilon_t, \tag{6.10}$$

where the ϵ are i.i.d. with $\mathrm{E}\,\epsilon_t = 0$, $\mathrm{Var}\left[\epsilon_t\right] = \sigma^2$. It follows from this model that y_i and y_j are not independent. However, if they are sufficiently far apart, we may assume that they are approximately independent - for a particular $l \geq p$ such that $|i - j| > l$. Hence, we can instead

withhold blocks of size $2l+1$ centred about a validation point:

$$CV_l(1) = \frac{1}{n-p}\sum_{t=p}^{n-1}\left(y_t - \hat{y}^l_{(t)}\right)^2,$$

where

$$\hat{y}^l_{(t)} = \hat{\Phi}_1^{(t\pm l)} y_{t-1} + \cdots + \hat{\Phi}_p^{(t\pm l)} y_{t-p},$$

with the somewhat heavy notation $\hat{\Phi}_i^{(t\pm l)}$ denoting an estimate of Φ_i withholding $\pm l$ neighbouring observations about y_t.

6.3 Wavelet Cross Validation Schemes

Wavelets are often used in time series applications where we are observing y_i which are not i.i.d.

$$y_i = f(t_i) + \xi_i.$$

However, we have seen that CV can still be applied in these settings. Further issues with wavelet CV schemes are that the DWT is designed to work in an equispaced setting (we should be sampling at a constant sample rate) and we also require a sample size of the form $n = 2^m$, $m \in \mathbb{N}$. This immediately makes it non-trivial to implement most delete-d or V-fold schemes. Nason (1996) has proposed a two-fold CV scheme and also a LOO CV scheme; the two-fold scheme splits the data into odd and even samples to retain a constant sampling rate, whilst the LOO scheme removes a point then reflects the signal and extends the size to a power of two. McCoy & Konstantinou (2002) derived delete-d schemes for $d = 1, 2, 3$ by writing the DWT as a linear regression and calculating the LS estimate of the reconstructing after deleting columns of the wavelet matrix. Oh, Kim & Lee (2009) also produced LOO and delete-d wavelet CV schemes using imputation for the left out data based on hierarchical likelihood. GCV has been extended to wavelets by Jansen, Malfait & Bultheel (1997) and Weyrich & Warhola (1998); their treatment of the missing value was to assume a value that would be unaffected by thresholding. Amato & Vuza (1998) used GCV to estimate the regularisation parameter in a regularised wavelet regression setting. Likewise, Antoniadis, Gregoire & McKeague (1994) and Amato, Antoniadis & Pensky (2006) used GCV for selection of parameters in wavelet kernel settings. Hurvich & Tsai (1998) extended Nason's two-fold scheme to take account of AIC. Nason (2002) also used an interpolating LOO scheme to select thresholds and also the number of vanishing moments as well as the primary resolution of a wavelet regression. McGinnity, Varbanov & Chicken (2017) also extended Nason's CV scheme to level-dependent thresholds and block thresholding to deal with non-Gaussian errors, and they minimise the threshold using a smoothing penalty on the reconstructions. The method is effective for non-Gaussian errors but less competitive when they are Gaussian. Moreover, whilst the level-dependent thresholds are beneficial for improved performance in MSE, they add significant computation time.

6.3.1 Nason's Two-fold Scheme

We divide our original data $\boldsymbol{y}$ into even and odd sets $y_0, y_2, ..., y_{n/2-2}$ and $y_1, y_3, ..., y_{n/2-1}$. The function f is estimated separately using each set. Let $\hat{f}^E_\lambda$ (respectively $\hat{f}^O_\lambda$) be the estimate of f using the even (odd) set and thresholded using λ. They are also re-indexed from $j = 0 \to n/2-1$. Then we calculate

$$\bar{f}^E_{\lambda,j} = \frac{1}{2}\left(\hat{f}^E_{\lambda,j} + \hat{f}^E_{\lambda,j-1}\right), \quad j = 0, \ldots, n/2-1$$

$$\bar{f}^O_{\lambda,j} = \frac{1}{2}\left(\hat{f}^O_{\lambda,j+1} + \hat{f}^O_{\lambda,j}\right), \quad j = 0, \ldots, n/2-1$$

We might assume periodic boundary coefficients to define $\hat{f}^E_{\lambda,-1}$ and $\hat{f}^O_{\lambda,n/2}$, or simply set them to $\hat{f}^E_{\lambda,0}$ and $\hat{f}^O_{\lambda,n/2-1}$ respectively.

$\bar{f}^E_{\lambda,j}$ interpolates even function estimates, hence we use this interpolate as an estimate for the odd values - and visa-versa for $\bar{f}^O_{\lambda,j}$. Hence, we estimate the Mean Integrated Square Error (MISE) using

$$\hat{M}(\lambda) = \sum_{j=0}^{n/2-1} \left[\left(\bar{f}^E_{\lambda,j} - y_{2j+1}\right)^2 + \left(\bar{f}^O_{\lambda,j} - y_{2j}\right)^2\right].$$

Let λ^* denote the minimiser of $\hat{M}(\lambda)$. This is a threshold for a vector of length $n/2$, so the final two-fold threshold for the full data of length n is a rescaled version of this:

$$\lambda^N_{CV} = \left(1 - \frac{\log(2)}{\log(n)}\right)^{-1/2} \lambda^*. \tag{6.11}$$

This scaling is inspired from Donoho & Johnstone's (1994) universal threshold, which we introduced in Section 2.8.3.

In practice, the minimiser λ^* can be found using a golden section search (see Section A.3) because the function $\hat{M}$ is nearly convex (Nason 1995). We can give this search algorithm suitable bounds for the threshold of 0 and the universal threshold λ_{UT} (Donoho & Johnstone 1994).

Nason estimates the MISE by comparing the original, noisy data f_i against interpolations of the smoothed $\hat{f}^E_{\lambda,j}$ and $\hat{f}^O_{\lambda,j}$. Equally, we could interpolate the noisy f_i and compare that directly with the estimates $\hat{f}^E_{\lambda,j}$ and $\hat{f}^O_{\lambda,j}$. Indeed, both seem valid and both have been implemented in Nason's (2013*b*) R package wavethresh. It is not clear which option would be preferable: the effect of this interpolation will be a small amount of smoothing; whether this should be on the noisy data or the already smoothed data is unclear.

6.3.2 McCoy's Delete-d

In this setting, we approach wavelet thresholding using matrix algebra. We hard threshold by removing rows of the wavelet matrix:

$$\hat{\boldsymbol{y}} = \mathcal{W}^T T\boldsymbol{\theta} = \mathcal{W}_p^T \boldsymbol{\theta}_p. \tag{6.12}$$

In this equation, $\mathcal{W}$ is a wavelet matrix, T is an $n \times n$ hard thresholding matrix defined below and $\boldsymbol{\theta}$ is a vector of length n of wavelet coefficients (elsewhere in this thesis, we use $\boldsymbol{w}$, but we change our notation here for the sake of readability and also to highlight the different scenario of treating $\boldsymbol{\theta}$ as a parameter that we are estimating).

T is a diagonal matrix with either a 1 or a 0 in the ith row if θ_i is kept or killed,

$$T_{i,i} = \begin{cases} 1 & \text{if } \theta_i \text{ is kept,} \\ 0 & \text{if } \theta_i \text{ is killed.} \end{cases}$$

$\boldsymbol{\theta}$ is assumed in the following section to be a full wavelet transform $\boldsymbol{\theta} = \{d_{j,k}, c_{0,0} : j = 0, \ldots, J-1, k = 0, \ldots, 2^j - 1\}$. Note that the scaling coefficient $c_{0,0}$ is left unaffected by thresholding. $\mathcal{W}_p$ is the $p \times n$ matrix after retaining p rows of $\mathcal{W}$ corresponding to the p retained coefficients $\boldsymbol{\theta}_p = \{\theta_{q_1}, \ldots, \theta_{q_p}\} = \{c_{0,0}, d_q : q = q_0, \ldots, q_{p-2}\}$. Here, $\{d_q\}$ is a re-indexed subset of $\{d_{j,k}\}$ after re-ordering and retaining only the $p-1$ greatest coefficients.

Remark 6.3.1. *It is unclear in this nonparametric regression setting where to use the hat notation for estimation. In Equation* (6.12)*, $\boldsymbol{y}$ is our estimate of the function $f(\boldsymbol{t})$. The matrix T could be considered as having 1s and 0s randomly distribution on its diagonals. Likewise, we could apply the same logic to $\mathcal{W}_p^T$ and $\boldsymbol{\theta}_p$. Instead, we opt to leave them out.*

Now if we consider the wavelet regression as a linear regression as in Bruce, Gao & Stuetzle (1999), then we can write down the least squares estimate of the hard-thresholded wavelet coefficients:

$$\hat{\boldsymbol{\theta}}_p^{(h)} = (\mathcal{W}_p \mathcal{W}_p^T)^{-1} \mathcal{W}_p \boldsymbol{y} = \mathcal{W}_p \boldsymbol{y},$$

using the orthogonality of $\mathcal{W}$. In practice, we are thresholding coefficients below a threshold λ, but in this linear regression setting, this λ will correspond to a number p of coefficients. The fitted values are then

$$\hat{\boldsymbol{y}} = \mathcal{W}_p^T \hat{\boldsymbol{\theta}}_p^{(h)} = \mathcal{W}_p^T \mathcal{W}_p \boldsymbol{y} = \mathcal{W}^T T \mathcal{W} \boldsymbol{y} = H_p^{(h)} \boldsymbol{y}. \tag{6.13}$$

This matrix $H_p^{(h)}$ is known as the hat/smoother matrix. The Effective Degrees of Freedom (EDF) is the trace of this matrix,

$$EDF = \mathrm{tr}(H_p^{(h)}) = \mathrm{tr}(W^T T W) = \mathrm{tr}(T) = p, \tag{6.14}$$

which, in our case of hard thresholding is p, the number of retained coefficients.

We can also define the soft thresholding estimator of $\boldsymbol{\theta}$, which is a shrinkage estimator retaining the zeros of the hard thresholding estimate but shrinking the rest of the coefficients towards 0 by the corresponding threshold λ. Our soft thresholding estimator is

$$\begin{aligned}\hat{\boldsymbol{\theta}}_p^{(s)} &= \hat{\boldsymbol{\theta}}_p^{(h)} - \lambda \bar{S}_p \boldsymbol{j}_p, \\ &= \dot{S}_{p,\lambda} \hat{\boldsymbol{\theta}}_p^{(h)} = \dot{S}_{p,\lambda} \mathcal{W}_p \boldsymbol{y},\end{aligned}$$

for $\boldsymbol{j}_p$ a vector of ones of length p and $p \times p$ diagonal matrices

$$(\bar{S}_p)_{ii} = \begin{cases} 0 & i = 0, \\ \text{sign}(\theta_p^{(h)}[i]) & \text{otherwise}, \end{cases} \tag{6.15}$$

$$(\dot{S}_{p,\lambda})_{ii} = \begin{cases} 1 & i = 0, \\ 1 - \frac{\lambda}{|\theta_p^{(h)}[i]|} & \text{otherwise}. \end{cases} \tag{6.16}$$

We have two equivalent definitions because the version with $\bar{S}_p$ is cheaper to compute, but $\dot{S}_{p,\lambda}$ is potentially more useful in the cross validation setting.

We then define $n \times n$ matrices $\bar{S}$ and $\dot{S}_\lambda$ to be diagonal matrices containing submatrices $\bar{S}_p$ and $\dot{S}_{p,\lambda}$ (respectively) on the diagonals corresponding to the p retained coefficients and with zeros on the other diagonals. This soft thresholding estimator gives new fitted values for $\boldsymbol{y}$,

$$\begin{aligned}\hat{\boldsymbol{y}}_{SOFT} &= H_p^{(h)} \boldsymbol{y} - \mathcal{W}^T \bar{S} \lambda \\ &= \mathcal{W}_p^T \dot{S}_{p,\lambda} \mathcal{W}_p \boldsymbol{y} = \mathcal{W}^T \dot{S}_\lambda \mathcal{W} \boldsymbol{y} = H_p^{(s)} \boldsymbol{y} \\ &= \mathcal{W}_p^T \dot{S}_{p,\lambda} \mathcal{W}_p \hat{\boldsymbol{y}}_{HARD}.\end{aligned}$$

Remark 6.3.2. *We note that the soft threshold estimator is not a LS estimator of $\boldsymbol{\theta}$: instead it is penalised LS. Antoniadis & Fan (2001) studied regularised wavelet estimators in a setting involving the full transform; here, we are already hard thresholding before running our LS regression - as we are using it in the following steps to work out our CV criterion. Antoniadis & Fan (2001) found that the hard and soft thresholds were in fact solutions to regularised regressions on the wavelet coefficients. However, we can "cheat" by modifying the wavelet matrix by stretching the values corresponding to the retained coefficients away from zero to coax out a LS estimator that is soft thresholding.*

If we define $\mathcal{V}_p = \left(\dot{S}_{p,\lambda}\right)^{-1} \mathcal{W}_p$, then we can also produce $\hat{\boldsymbol{\theta}}_p^{(s)}$ via

$$\begin{aligned}\hat{\boldsymbol{\theta}}_p^{(s)} &= \left(\mathcal{V}_p \mathcal{V}_p^T\right)^{-1} \mathcal{V}_p \boldsymbol{y} \\ &= \left(\left(\dot{S}_{p,\lambda}\right)^{-1} \mathcal{W}_p \mathcal{W}_p^T \left(\dot{S}_{p,\lambda}\right)^{-1}\right)^{-1} \left(\dot{S}_{p,\lambda}\right)^{-1} \mathcal{W}_p \boldsymbol{y}\end{aligned}$$

$$= \left(I \left(\dot{S}_{p,\lambda} \right)^{-2} \right)^{-1} \left(\dot{S}_{p,\lambda} \right)^{-1} \mathcal{W}_p \boldsymbol{y}$$
$$= \dot{S}_{p,\lambda} \mathcal{W}_p \boldsymbol{y}.$$

McCoy's Delete-1 Cross Validation

McCoy then further modifies the wavelet matrix to perform LOO CV and concentrates on hard thresholding. We would like to leave out the kth element from our regression. So we define $\boldsymbol{y}^{(k)}$ now with superscripts to be $\boldsymbol{y}$ missing the kth component (we now switch to superscripts in the interest of aesthetics as this section is heavy in notation). Then we let $\mathcal{W}_p^{(k)}$ be the $p \times (n-1)$ matrix obtained by removing the kth column from W_p. Accordingly, our LS estimator is now

$$\hat{\boldsymbol{\theta}}_p^{(h)(k)} = \left(\mathcal{W}_p^{(k)} (\mathcal{W}_p^{(k)})^T \right)^{-1} \mathcal{W}_p^{(k)} \boldsymbol{y}^{(k)}. \tag{6.17}$$

Thus, our hat matrix when we leave out the kth element is

$$H_p^{(h)(k)} = \mathcal{W}_p^T \left(\mathcal{W}_p^{(k)} (\mathcal{W}_p^{(k)})^T \right)^{-1} \mathcal{W}_p^{(k)}.$$

We now define $\boldsymbol{\omega}_i$ to be the ith column vector of length p of matrix $\mathcal{W}_p$. Then the CV(1) criterion in our setting becomes

$$CV(1) = \frac{1}{n} \sum_{i=0}^{n-1} \left(y_i - \boldsymbol{\omega}_i^T \boldsymbol{\theta}_p^{(i)} \right)^2, \tag{6.18}$$

which is a function of p. As in standard LOO CV, we can calculate this more efficiently than above, by using the Shermann-Morrison formula once again. We can write

$$\begin{aligned} \mathcal{W}_p^{(k)} (\mathcal{W}_p^{(k)})^T &= \mathcal{W}_p \mathcal{W}_p^T - \boldsymbol{\omega}_k \boldsymbol{\omega}_k^T \\ &= I + H_k, \end{aligned}$$

where H_k is a rank one matrix. Substituting this expression into the Shermann-Morrison formula, we obtain

$$\begin{aligned} \left(\mathcal{W}_p^{(k)} (\mathcal{W}_p^{(k)})^T \right)^{-1} &= I + \frac{\boldsymbol{\omega}_k \boldsymbol{\omega}_k^T}{1 - \boldsymbol{\omega}_k^T \boldsymbol{\omega}_k} \\ &= I + \frac{\boldsymbol{\omega}_k \boldsymbol{\omega}_k^T}{1 - a_{kk}}, \end{aligned}$$

where a_{kk} is the kth diagonal of the $n \times n$ hat matrix $H_p^{(h)}$,

$$a_{kk} = (\mathcal{W}_p^T \mathcal{W}_p)_{kk} = \boldsymbol{\omega}_k^T \boldsymbol{\omega}.$$

Plugging this result into the criterion and simplifying, we obtain the following result:

$$CV(1) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{(y_i - \hat{y}_i)^2}{(1-a_{ii})^2}, \tag{6.19}$$

using the hard thresholding estimate of $\boldsymbol{y}$, $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}_{HARD}$. Note that this is a function of p or λ.

Remark 6.3.3. *However, we note that $\hat{\boldsymbol{\theta}}_p^{(h)(k)}$ is not a true hard threshold estimator but rather derived from a hard threshold estimator. This method does not guarantee that the elements of $\hat{\boldsymbol{\theta}}_p^{(h)(k)}$ sit on the hard thresholding graph in Figure 2.13 on page 44; rather, we might have an element that satisfies*

$$0 < |\boldsymbol{\theta}_p^{(h)(k)}[i]| < \lambda. \tag{6.20}$$

As such, it is not straightforward to define a "soft threshold" estimator $\hat{\boldsymbol{\theta}}_p^{(s)(k)}$: if we use $\dot{S}_{p,\lambda}$ to shrink our coefficients then it would contain information about y_k, the left out point; on the other hand it is not clear how to define an alternative shrinking matrix $\dot{S}_{p,\lambda}^{(k)}$ because of (6.20)*. If we could define such a matrix, then this soft threshold estimator could be easily formed:*

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_p^{(s)(k)} &= \left(\left(\dot{S}_{p,\lambda}^{(k)}\right)^{-1}\mathcal{W}_p^{(k)}\left(\mathcal{W}_p^{(k)}\right)^T\left(\dot{S}_{p,\lambda}^{(k)}\right)^{-1}\right)^{-1}\left(\dot{S}_{p,\lambda}^{(k)}\right)^{-1}\mathcal{W}_p^{(k)}\boldsymbol{y}^{(k)} \\
&= \dot{S}_{p,\lambda}^{(k)}\left(\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T\right)^{-1}\dot{S}_{p,\lambda}^{(k)}\left(\dot{S}_{p,\lambda}^{(k)}\right)^{-1}\mathcal{W}_p^{(k)}\boldsymbol{y}^{(k)} \\
&= \dot{S}_{p,\lambda}^{(k)}\left(\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T\right)^{-1}\mathcal{W}_p^{(k)}\boldsymbol{y}^{(k)} \\
&= \dot{S}_{p,\lambda}^{(k)}\hat{\boldsymbol{\theta}}_p^{(h)(k)}.
\end{aligned}$$

We would then have the hat matrix

$$H_p^{(s)(k)} = \mathcal{W}_p^T\dot{S}_{p,\lambda}^{(k)}\left(\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T\right)^{-1}\mathcal{W}_p^{(k)}.$$

Issues With This LOO

As McCoy points out, this LOO criterion is often "ill-posed". This is because of two equivalent issues:

- Matrix $M = \left(\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T\right)$ is not invertible,
- Matrix diagonal $a_{ii} = 1$ for some i in $0, \ldots, n-1$.

The second point is clearly a problem for orthogonal wavelet transforms: if we do no thresholding then our smoothing matrix is clearly an identity matrix, giving all $a_{ii} = 1$. Indeed, considering the first bullet, we can easily show that M is not invertible in the case of no thresholding: eliminating a column of the full wavelet matrix $\mathcal{W}$ will give a new $n \times (n-1)$ matrix $\mathcal{W}^{(k)}$ of rank $n-1$; M in this case will be an $n \times n$ matrix of rank $n-1$ by properties of Gram

matrices, hence it is not invertible. This illustrates how we need to ensure that removing rows in thresholding and removing columns of the wavelet matrix do not both reduce the rank of the resulting matrix $\mathcal{W}_p^{(k)}$; the intersection of the column and row to be removed must contain a non-zero wavelet filter coefficient.

We can see a few ways of potentially avoiding this issue:

- Threshold wavelet coefficients in lower scales or threshold all detail coefficients in the first level (these are more of a 'hack'!),
- Use long wavelet filters (the Haar filter is the worst affected - longer filters would be better),
- Extend to MODWT wavelet transforms,
- Consider a penalised LS rather than a standard LS regression.

We might consider leaving out more points to solve the issue of $a_{ii} = 0$ in the CV criterion, however this would change the hat matrix that we are trying to invert, thus creating a different criterion. In fact, our issue of rank deficiency would be worsened by leaving out more points!

Explanation of This Cross Validation Scheme

We will illustrate the effect of this cross validation scheme using the hat matrices $H_p^{(h)}$ after thresholding and $H_p^{(h)(k)}$ following column removal for LOO. Firstly, we will consider the 8×8 Haar wavelet matrix:

$$\mathcal{W}_{HAAR} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ 1/2 & 1/2 & -1/2 & -1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & -1/2 & -1/2 \\ 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}. \quad (6.21)$$

Note that this matrix is ordered differently from (2.15) so that it more closely resembles the design matrix. We recall $\mathcal{W}^T\mathcal{W} = I$ for a wavelet matrix $\mathcal{W}$. Now we will consider the hat matrices when we leave out the first column. In this scenario, our observed vector is $\boldsymbol{y} = (y_0, y_1, \ldots, y_7)$. Our estimate following thresholding is $\hat{\boldsymbol{y}} = H_p^{(h)}\boldsymbol{y}$ and the estimate after also leaving out the first observation is $\hat{\boldsymbol{y}}^{(0)} = H_p^{(h)(0)}\boldsymbol{y}^{(0)}$. See Tables 6.1 and 6.2 for an illustration of various scenarios in this situation.

Thresholded row	$H_p^{(h)}$	$H_p^{(h)(k=0)}$
Second row of $\mathcal{W}_{HAAR}$ (producing coefficient $d_{2,0}$)	$\begin{pmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ The effect of thresholding is to average the first two coefficients	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ Leaving out y_0 means that this matrix sets the first element of $\hat{\boldsymbol{y}}^{(0)}$ equal to y_1
Sixth row of $\mathcal{W}_{HAAR}$ (producing coefficient $d_{1,0}$)	$\begin{pmatrix} 3/4 & -1/4 & 1/4 & 1/4 & 0 & 0 & 0 & 0 \\ -1/4 & 3/4 & 1/4 & 1/4 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & 3/4 & -1/4 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & -1/4 & 3/4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ The effect of thresholding is to do weighted averaging of the first four coefficients	$\begin{pmatrix} -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ This matrix sets the first element of $\hat{\boldsymbol{y}}^{(0)}$ to be $-y_1 + y_2 + y_3$ and leaves all other elements untouched
Eighth row of $\mathcal{W}_{HAAR}$ (producing coefficient $d_{0,0}$)	$\frac{1}{8}\begin{pmatrix} 7 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & 7 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 7 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 7 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 7 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & 7 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & 7 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & 7 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$

Table 6.1: Table of hat matrices for an 8×8 Haar wavelet matrix: the second column contains hat matrices from just hard thresholding of one coefficient; the third column contains hat matrices from hard thresholding then leaving out y_0 under the McCoy LOO scheme.

Thresholded row	$H_p^{(h)}$	$H_p^{(h)(k=0)}$
Second and sixth row of $\mathcal{W}_{HAAR}$ (producing coefficients $d_{2,0}, d_{1,0}$)	$\begin{pmatrix} 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & 3/4 & -1/4 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & -1/4 & 3/4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1/3 & 1/3 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 5/6 & -1/6 & 0 & 0 & 0 & 0 \\ 1/3 & -1/6 & 5/6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ The first three elements of $\hat{\boldsymbol{y}}^{(0)}$ are affected now
Second and third row of $\mathcal{W}_{HAAR}$ (producing coefficients $d_{2,0}, d_{2,1}$)	$\begin{pmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ Here we are showing an example of some thresholding where there are thresholded coefficients unaffected by the CV step	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ We see that the thresholding of $d_{0,0}$ is indeed unaffected by the CV

Table 6.2: Table of hat matrices for an 8×8 Haar wavelet matrix: the second column contains hat matrices from just hard thresholding of two coefficients; the third column contains hat matrices from hard thresholding then leaving out y_0 under the McCoy LOO scheme.

Secondly, we consider a Daubechies 4 wavelet matrix to compare the behaviour for larger filters:

$$\mathcal{W}_{D4} = \begin{pmatrix} 0.35355 & 0.35355 & 0.35355 & 0.35355 & 0.35355 & 0.35355 & 0.35355 & 0.35355 \\ 0.83652 & -0.48296 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & -0.12941 & -0.22414 \\ -0.12941 & -0.22414 & 0.83652 & -0.48296 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & -0.12941 & -0.22414 & 0.83652 & -0.48296 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 & -0.12941 & -0.22414 & 0.83652 & -0.48296 \\ 0.35377 & 0.72877 & -0.04575 & -0.51226 & -0.17075 & -0.04575 & -0.13726 & -0.17075 \\ -0.17075 & -0.04575 & -0.13726 & -0.17075 & 0.35377 & 0.72877 & -0.04575 & -0.51226 \\ -0.06470 & 0.24148 & 0.37089 & 0.54767 & 0.06470 & -0.24148 & -0.37089 & -0.54767 \end{pmatrix}.$$

The thresholding hat matrix after removing the second column of the Daubechies wavelet matrix is then

$$H_p^{(h)} = \begin{pmatrix} 0.30024 & 0.40401 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & 0.10825 & 0.18750 \\ 0.40401 & 0.76675 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & -0.06250 & -0.10825 \\ 0.00000 & 0.00000 & 1.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 1.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 & 1.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & 1.00000 & 0.00000 & 0.00000 \\ 0.10825 & -0.06250 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & 0.98325 & -0.02901 \\ 0.18750 & -0.10825 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & -0.02901 & 0.94976 \end{pmatrix}$$

$$= \begin{pmatrix} a & b & & c & d \\ b & e & 0_{2,4} & f & -c \\ & & & & \\ 0_{4,2} & & I_4 & & 0_{4,2} \\ & & & & \\ c & f & & g & h \\ d & -c & 0_{2,4} & h & i \end{pmatrix}.$$

Finally, the hat matrix after LOO is

$$H_p^{(h)(k)} = \begin{pmatrix} \frac{b}{1-a} & 0_{1,4} & \frac{c}{1-a} & \frac{d}{1-a} \\ & & & \\ & I_7 & & \\ & & & \end{pmatrix},$$

indicating that McCoy's LOO CV with longer filters will use more coefficients to estimate a missing value.

McCoy's Delete-2 and Delete-3 Cross Validation

McCoy (1999) then derives expressions for the cross validatory criteria of delete-2 and delete-3 schemes. After this point, the expansions become very onerous to derive.

6.3.3 Wavelet Delete-d Cross Validation

In this section, we consider a different tact to extend McCoy's CV schemes: we extend it to delete-d for general d via an iterative formation. The advantages of this approach are the following:

- No limitation on the number of leave-out samples d as far as the implementation is concerned,
- Simple generalisation to regularisation,
- One implementation needed for all values of d as supposed to a different implementation for each value of d.

The main disadvantage is in computational efficiency: McCoy's scheme will be faster to compute for any given d for which the CV criterion has been derived and implemented.

Iterative Sherman-Morrison algorithm

Egidi & Maponi (2006) derived an algorithm for iterative rank-m updates in a setting of solving linear equations $A\boldsymbol{x} = \boldsymbol{b}$ for an $n \times n$ matrix A.

Corollary 6.3.4 (Egidi & Maponi (2006))**.** *Let B be an $n \times n$ matrix with $\det(B) \neq 0$ and $\det(B + \boldsymbol{u}\boldsymbol{v}^T) \neq 0$, let $\boldsymbol{x}'$ be the solution to $B\boldsymbol{x} = \boldsymbol{c}$ and let $\boldsymbol{z}'$ be the solution to $B\boldsymbol{z} = \boldsymbol{u}$. Then the solution to*

$$\left(B + \boldsymbol{u}\boldsymbol{v}^T\right)\boldsymbol{x} = \boldsymbol{c},$$

is

$$\boldsymbol{x} = \boldsymbol{x}' - \frac{\boldsymbol{v}^T\boldsymbol{x}'}{1 + \boldsymbol{v}^T\boldsymbol{z}'}\boldsymbol{z}'.$$

Now suppose A can now be rewritten as m rank-one updates from a matrix A_0:

$$A = A_0 + \boldsymbol{u}_1\boldsymbol{v}_1^T + \boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \boldsymbol{u}_m\boldsymbol{v}_m^T,$$

with $\boldsymbol{x}_0$ as the solution to $A_0\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{z}_{0,k}$ as the solution to $A_0\boldsymbol{z} = \boldsymbol{u}_k$. Also let

$$A_l = A_0 + \boldsymbol{u}_1\boldsymbol{v}_1^T + \boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \boldsymbol{u}_l\boldsymbol{v}_l^T, \quad l = 1, 2, \ldots, m.$$

Then if each A_l is non-singular, then we can calculate $\boldsymbol{x}_m$, the solution to $A\boldsymbol{x} = \boldsymbol{b}$, with Algorithm 5.

Algorithm 5: Iterative Sherman-Morrison updates, Egidi & Maponi (2006)

Compute $\boldsymbol{x} = \boldsymbol{x}_0$, the solution to $A_0\boldsymbol{x} = \boldsymbol{b}$

if $m > 0$ **then**

  for $k = 1$ **to** m **do**

    compute $\boldsymbol{z} = \boldsymbol{z}_{0,k}$, the solution to $A_0\boldsymbol{z} = \boldsymbol{u}_k$

  for $l = 1$ **to** $m - 1$ **do**

$$\boldsymbol{x}_l = \boldsymbol{x}_{l-1} - \frac{\boldsymbol{v}_l^T \boldsymbol{x}_{l-1}}{1 + \boldsymbol{v}_l^T \boldsymbol{z}_{l-1,l}} \boldsymbol{z}_{l-1,l}$$

    for $k = l + 1$ **to** m **do**

$$\boldsymbol{z}_{l,k} = \boldsymbol{z}_{l-1,k} - \frac{\boldsymbol{v}_l^T \boldsymbol{z}_{l-1,k}}{1 + \boldsymbol{v}_l^T \boldsymbol{z}_{l-1,l}} \boldsymbol{z}_{l-1,l}$$

$$\boldsymbol{x}_m = \boldsymbol{x}_{m-1} - \frac{\boldsymbol{v}_m^T \boldsymbol{x}_{m-1}}{1 + \boldsymbol{v}_m^T \boldsymbol{z}_{m-1,m}} \boldsymbol{z}_{m-1,m}$$

Iterative Delete-d Cross Validation

We can use Algorithm 5 to extend delete-d Cross Validation for general d. Let

$$\begin{aligned} A &= A_0 + \boldsymbol{u}_1\boldsymbol{v}_1^T + \boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \boldsymbol{u}_d\boldsymbol{v}_d^T \\ &= \mathcal{W}_p\mathcal{W}_p^T - \boldsymbol{\omega}_{k_1}\boldsymbol{\omega}_{k_1}^T - \boldsymbol{\omega}_{k_2}\boldsymbol{\omega}_{k_2}^T - \cdots - \boldsymbol{\omega}_{k_d}\boldsymbol{\omega}_{k_d}^T \\ &= \mathcal{W}_p^{(k,d)}(\mathcal{W}_p^{(k,d)})^T, \end{aligned} \tag{6.22}$$

i.e. we have $A_0 = \mathcal{W}_p(\mathcal{W}_p)^T = I$ and $-\boldsymbol{v}_l = \boldsymbol{\omega}_{k_l} = \boldsymbol{u}_l$, for $l = 1, \ldots, d$. The slight abuse of notation of the superscript (k, d) denotes leaving out d values or columns from a vector or matrix respectively from k_1 to k_d.

The orthogonality of the wavelet matrix means that many of the calculations we require are simplified. The linear equation we are trying to solve is

$$A\hat{\boldsymbol{\theta}}_p^{(k,d)} = \mathcal{W}_p^{(k,d)}\boldsymbol{y}^{(k,d)}, \tag{6.23}$$

in the notation of the iterative Sherman-Morrison (returning $\boldsymbol{x}_m = \hat{\boldsymbol{\theta}}_p^{(k,d)}$). Equivalently, we can write this as a LS estimator of $\boldsymbol{\theta}$ as previously:

$$\hat{\boldsymbol{\theta}}_p^{(k,d)} = \left(\mathcal{W}_p^{(k,d)}(\mathcal{W}_p^{(k,d)})^T\right)^{-1}\mathcal{W}_p^{(k,d)}\boldsymbol{y}^{(k,d)}. \tag{6.24}$$

The hat matrix in this case is calculated as before:

$$H_p^{(h)(k,d)} = \mathcal{W}_p^T\left(\mathcal{W}_p^{(k,d)}(\mathcal{W}_p^{(k,d)})^T\right)^{-1}\mathcal{W}_p^{(k,d)}. \tag{6.25}$$

We calculate the initial values for Algorithm 5 as

$$\begin{aligned}\boldsymbol{x}_0 &= A_0^{-1}\mathcal{W}_p^{(k,d)}\boldsymbol{y}^{(k,d)}\\ &= \boldsymbol{\theta}_p - \boldsymbol{\omega}_{k_1}y_{k_1} - \cdots - \boldsymbol{\omega}_{k_d}y_{k_d},\\ \boldsymbol{z}_{0,l} &= A_0^{-1}\boldsymbol{u}_l\\ &= \boldsymbol{\omega}_{k_l},\end{aligned}$$

which demonstrates that we can avoid calculating matrix inversions or multiplications directly. Instead, we only have to calculate vector dot products and scalar-vector multiplications. In order to compute the CV(d) criterion 6.5, we need to work out

$$\zeta_{k,l,d} = y_{k_l} - \boldsymbol{\omega}_{k_l}^T\hat{\boldsymbol{\theta}}_p^{(k,d)} \quad \text{for } 1 \le l \le d.$$

Remark 6.3.5. *We can show that this iterative delete-d scheme yields the same result as McCoy & Konstantinou's (2002) LOO cross validation. Firstly,*

$$A = A_0 + \boldsymbol{u}_1\boldsymbol{v}_1 = \mathcal{W}_p\mathcal{W}_p^T - \boldsymbol{\omega}_k\boldsymbol{\omega}_k^T = I - \boldsymbol{\omega}_k\boldsymbol{\omega}_k^T = \mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T.$$

We have $-\boldsymbol{v}_1 = \boldsymbol{\omega}_k = \boldsymbol{u}_1$*, giving* $\boldsymbol{x}_0 = \boldsymbol{\theta}_p - \boldsymbol{\omega}_k y_k$ *and* $\boldsymbol{z}_{0,1} = \boldsymbol{\omega}_k$*. Algorithm 5 gives us*

$$\boldsymbol{x}_m = \hat{\boldsymbol{\theta}}_p^{(k)} = (\boldsymbol{\theta}_p - \boldsymbol{\omega}_k y_k) + \frac{\boldsymbol{\omega}_k^T(\boldsymbol{\theta}_p - \boldsymbol{\omega}_k y_k)}{1 - \boldsymbol{\omega}_k^T\boldsymbol{\omega}_k}\boldsymbol{\omega}_k.$$

Thus, using $\boldsymbol{\omega}_i^T\boldsymbol{\theta}_p = \hat{y}_i$ *and* $\frac{1}{1-a} = 1 + \frac{a}{1-a} = 1 + a + \frac{a^2}{1-a}$*,*

$$\begin{aligned}y_i - \boldsymbol{\omega}_i^T\hat{\boldsymbol{\theta}}_p^{(i)} &= y_i - \boldsymbol{\omega}_i^T\left((\boldsymbol{\theta}_p - \boldsymbol{\omega}_i y_i) + \frac{\boldsymbol{\omega}_i^T(\boldsymbol{\theta}_p - \boldsymbol{\omega}_i y_i)}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}\boldsymbol{\omega}_i\right)\\ &= y_i - \hat{y}_i + \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i y_i - \frac{\boldsymbol{\omega}_i^T\hat{y}_i\boldsymbol{\omega}_i}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i} + \frac{\boldsymbol{\omega}_i^T(\boldsymbol{\omega}_i^T\boldsymbol{\omega}_i)y_i\boldsymbol{\omega}_i}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}\\ &= y_i\left(1 + \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i + \frac{(\boldsymbol{\omega}_i^T\boldsymbol{\omega}_i)^2}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}\right) - \hat{y}_i\left(1 + \frac{\boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}\right)\\ &= \frac{y_i - \hat{y}_i}{1 - \boldsymbol{\omega}_i^T\boldsymbol{\omega}_i}.\end{aligned}$$

Substituting this into (6.18)*, we yield Equation* (6.19) *as desired.*

Regularised Iterative Delete-d Estimator

Now we consider a regularised version of A in Equation (6.22) to give a shrinkage estimator of $\boldsymbol{\theta}$ instead of (6.24). We can consider a ridge regression (see Section A.1.3 for details) instead of the previous least squares regression. This regularises A using parameter γ to ensure it is still invertible, and still retains a closed form solution. Moreover, since we were not able to derive a

soft threshold cross validation estimator, this provides an alternative form of shrinkage (albeit proportional rather than by a constant shift). In this setting, our estimator of $\boldsymbol{\theta}$ becomes

$$\hat{\boldsymbol{\theta}}_{p,\gamma}^{(k,d)} = \left(\mathcal{W}_p^{(k,d)}(\mathcal{W}_p^{(k,d)})^T + \gamma I\right)^{-1} \mathcal{W}_p^{(k,d)} \boldsymbol{y}^{(k,d)}. \tag{6.26}$$

Our hat matrix is:

$$H_{p,\gamma}^{(k,d)} = \mathcal{W}_p^T \left(\mathcal{W}_p^{(k,d)}(\mathcal{W}_p^{(k,d)})^T + \gamma I\right)^{-1} \mathcal{W}_p^{(k,d)}. \tag{6.27}$$

We can still efficiently compute an iterative rank-d update using Algorithm 5. We have

$$A_o = \mathcal{W}_p(\mathcal{W}_p)^T + \gamma I = I(1+\gamma),$$

and

$$-\boldsymbol{v}_l = \boldsymbol{\omega}_{k_l} = \boldsymbol{u}_l.$$

Then we can write

$$A = A_0 - \boldsymbol{\omega}_{k_1}\boldsymbol{\omega}_{k_1}^T - \boldsymbol{\omega}_{k_2}\boldsymbol{\omega}_{k_2}^T - \cdots - \boldsymbol{\omega}_{k_d}\boldsymbol{\omega}_{k_d}^T,$$

as before. We can also write down our initial conditions for the algorithm:

$$\begin{aligned}
\boldsymbol{x}_0 &= A_0^{-1}\mathcal{W}_p^{(k,d)}\boldsymbol{y}^{(k,d)} \\
&= \frac{1}{1+\gamma}\left(\boldsymbol{\theta}_p - \boldsymbol{\omega}_{k_1}y_{k_1} - \cdots - \boldsymbol{\omega}_{k_d}y_{k_d}\right), \\
\boldsymbol{z}_{0,l} &= A_0^{-1}\boldsymbol{u}_l \\
&= \frac{\boldsymbol{\omega}_{k_l}}{1+\gamma},
\end{aligned}$$

The regularised hat matrix can be written out as

$$H_{p,\gamma}^{(h)(k)} = \left(\frac{1}{1+\gamma}\right)\mathcal{W}_p^T\left[I + \frac{\left(\frac{1}{1+\gamma}\right)\boldsymbol{\omega}_k\boldsymbol{\omega}_k^T}{1+\gamma-\boldsymbol{\omega}_k^T\boldsymbol{\omega}_k}\right]\mathcal{W}_p^{(k)}.$$

We extend Table 6.1 to add the effect of regularisation in Table 6.3. By comparing the two, we note the addition of shrinkage in the regularised version.

We can then pick specific values of γ such that we get sensible estimates of $\boldsymbol{\theta}$. To see the effect of γ on thresholding, let us consider the effect of this regularised approach when we are not leaving any coefficients out but still hard thresholding (i.e. Equation (6.26) with $d = 0$). This would give

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_{p,\gamma} &= \left(\mathcal{W}_p(\mathcal{W}_p)^T + \gamma I\right)^{-1}\mathcal{W}_p\boldsymbol{y} \\
&= \frac{1}{1+\gamma}\mathcal{W}_p\boldsymbol{y}.
\end{aligned}$$

Thresholded row	$H_{p,\gamma}^{(h)(k=0)}$	
$d_{2,0}$	$\begin{pmatrix} a & 0 & 0 & 0 & 0 & 0 & 0 \\ a & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & b & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & b \end{pmatrix}$	$a = \frac{1}{2\gamma+1}$ $b = \frac{1}{\gamma+1}$
$d_{1,0}$	$\begin{pmatrix} -a & a & a & 0 & 0 & 0 & 0 \\ b & c & c & 0 & 0 & 0 & 0 \\ c & b & -c & 0 & 0 & 0 & 0 \\ c & -c & b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d \end{pmatrix}$	$a = \frac{1}{4\gamma+1}$ $b = \frac{3\gamma+1}{(4\gamma+1)(\gamma+1)}$ $c = \frac{\gamma}{(4\gamma+1)(\gamma+1)}$ $d = \frac{1}{\gamma+1}$
$d_{0,0}$	$\begin{pmatrix} -a & -a & -a & a & a & a & a \\ b & -c & -c & c & c & c & c \\ -c & b & -c & c & c & c & c \\ -c & -c & b & c & c & c & c \\ c & c & c & b & -c & -c & -c \\ c & c & c & -c & b & -c & -c \\ c & c & c & -c & -c & b & -c \\ c & c & c & -c & -c & -c & b \end{pmatrix}$	$a = \frac{1}{8\gamma+1}$ $b = \frac{7\gamma+1}{(8\gamma+1)(\gamma+1)}$ $c = \frac{\gamma}{(8\gamma+1)(\gamma+1)}$

Table 6.3: Table of regularised hat matrices for an 8×8 Haar wavelet matrix: the second column contains regularised hat matrices from hard thresholding then leaving out y_0 under the regularised McCoy LOO scheme.

This as a thresholding rule would be

$$\eta^{\gamma}(d, \lambda) = \frac{\eta_H(d, \lambda)}{1+\gamma}, \tag{6.28}$$

which is hard thresholding with shrinkage. We thus consider this regularised delete-d estimator as the shrinkage estimator.

Whilst we initially considered this regularised estimator of $\boldsymbol{\theta}$ in (6.26) as a modification to ensure we can still produce an estimator when the matrix A is singular, this turns out to be palliative: whilst we might indeed be able to invert the matrix A with regularisation, the resulting estimator is not helpful. To illustrate this, let us consider $\mathcal{W}_p^{(k)}$ to be a $p \times (n-1) = 7 \times 7$ Haar matrix with row 1 removed for thresholding and column $k = 3$ removed for validation. This reduces the rank of $\mathcal{W}_p^{(k)}$ to 6. So $\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T$ is not invertible, but $\mathcal{W}_p^{(k)}(\mathcal{W}_p^{(k)})^T + \gamma$ can be invertible. However, for small γ, we find that $H_{p,\gamma}^{(k,d)} \approx \mathcal{W}_p^T \mathcal{W}_p^{(k)}$, which has a zero row as its first row. This is clearly not useful, as it will always give $\hat{y}_0 = 0$.

However, this regularisation does have the benefit of shrinking the validation points. Indeed,

in CV we only use Equations (6.26) to evaluate estimates of $\boldsymbol{y}$ at the validation points that we have left out. In fact, if we reconstruct the entirety of $\boldsymbol{y}$ from (6.26) then all wavelet coefficients are shrunk proportionately and we lose too much of the energy of the signal.

6.3.4 Regularised Iterative V-fold Cross Validation

McCoy (1999) leaves out blocks of d coefficients which seems sensible in a wavelet application where we might expect to have autocorrelation in the y_i values by analogy to the autoregressive models (see Section 6.2). However, for increasing d we find that this has an adverse effect: it worsens the of invertibility of matrices by adding more stress to localised regions of thresholded coefficients; it also degrades the reconstructions in these localised regions.

Instead, we consider instead spacing out our $k_1, \ldots, k_d$. This design is more suited to a V-fold cross validation scheme. Our initial vector is of length $n = 2^J$, so we can pick $V = 2^{j_V}$ for $j_V = 1, \ldots, J$ to divide the set $\{y_1, \ldots, y_n\}$ into subsets of equal cardinality $n/V = 2^{J-j_V}$. We could then maximise our spacing over all V sets by assigning subsets $\{y_{k_i^v} : k_i^v = v \mod n\}$ for $v = 1, \ldots, V$. V-fold cross validation still works when using subsets of roughly equal cardinality. So we are not restricted to powers of two, which is an interesting feature of this methodology. A V-fold cross validation would hence follow the following procedure:

- Select a value V for which to perform V-fold CV. Select regularisation parameter γ.
- Divide the set $\{y_0, \ldots, y_{n-1}\}$ into subsets $\{y_{k_1^v}, \ldots, y_{k_{d(v)}^v} : k_i^v = v \mod n, i = 0, \ldots, n-1\}$, where $d(v)$ is simply the cardinality of the set $\{k_i^v\}$, roughly n/V.
- Take the DWT of $\boldsymbol{y}$, and obtain its corresponding matrix $\mathcal{W}$.
- Calculate an upper bound of the threshold λ for the entire $\boldsymbol{y}$ using the universal threshold. Find the value of p corresponding to this λ.
- For each $v = 1, \ldots, V$,
  - Calculate $\hat{\boldsymbol{\theta}}_{p,\gamma}^{(k,d(v))}$ using (6.26),
  - Construct $\hat{y}_{k_1^v}, \ldots, \hat{y}_{k_{d(v)}^v}$ by left multiplying $\hat{\boldsymbol{\theta}}_{p,\gamma}^{(k,d(v))}$ by $\mathcal{W}_p^T$,
- Calculate the cross validatory criterion

$$CV(d(V)) = \frac{1}{V} \sum_{v=1}^{V} \left[\sum_{i=1}^{d(v)} (y_{k_i^v} - \hat{y}_{k_i^v})^2 \right].$$

- Minimise $CV(d(V))$ using golden section search to select λ_{CV}.

We note that this λ_{CV} will be used for standard hard thresholding without regularisation. Furthermore, whilst we use the wavelet matrix $\mathcal{W}$, in practice the wavelet transforms should be performed by the standard pyramid algorithm.

6.4 Simulation Study

We conducted a simulation study comparing our V-fold method with Nason's twofold cross validation and the standard policies of SURE thresholding and universal thresholding. We compare these methods using MSE of the reconstructed signal using the selected threshold. Note that whilst the SURE threshold is designed to minimise MSE, the universal threshold is designed to remove all noise - so we include it more as a benchmark. We also calculated the oracle hard threshold in each simulation which is the actual threshold that simulation that minimises the MSE. We use the signals Doppler, Heavisine, Blocks and Cubic (which we defined in Section 2.3) of length 512, to which we add Gaussian white noise with a SNR of 5. We ran 100 trials for each signal. We use slightly cryptic notation to keep the tables compact: N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G. We ran our CV schemes with regularisation parameter values 0, 0.1 and 0.2, but we only include here the best performing schemes. Recall that $V = n$ fold CV is the same as LOO CV.

The main results can be seen in Table 6.4, in which we record the mean and standard deviation (in parentheses) of the MSE for each method rounded to 3 significant figures. Further tables can be found in the Appendix section D.1 in Tables D.1, D.2, D.3 and D.4 which display, respectively the number of trials that each CV scheme beat (or tied for first) all other CV schemes, beat the SURE scheme, lost to the universal threshold and made the oracle threshold. We note that Nason's twofold with soft threshold scheme even beat the (hard) oracle threshold in a few trials! These tables provide additional insight on the robustness of the different schemes compared to Table 6.4 which report average performance.

For the Doppler signal, the best performing schemes are Nason's soft thresholding CV scheme and also our two and threefold schemes with regularisation. We note that SURE thresholding here also performs well and even beats our LOO scheme. Nason's schemes and LOO perform best for the Heavisine signal, and the regularisation significantly worsens the performance of our LOO schemes; this is likely to be due to excess shrinkage at the discontinuity of the signal. For the Blocks signal, we find that all of our tabulated CV schemes perform better than Nason's CV on average, except for the LOO with no regularisation which also loses out to SURE thresholding. Lastly, LOO with no regularisation and Nason's CV with hard thresholding perform excellently with the cubic signal. Unfortunately, our schemes with low V all struggle to beat even the universal threshold. This is even more apparent in Table D.3.

Signal	Oracle	SURE	Univ	N2h	N2s	M2.1	M2.2	M3.2
Doppler	0.55	0.677	2.24	0.65	**0.627**	0.653	**0.612**	**0.622**
	(0.085)	(0.103)	(0.214)	(0.102)	**(0.0765)**	(0.118)	**(0.0982)**	**(0.109)**
Heavisine	25.2	37.4	89.1	**31.9**	**35.3**	46.0	35.8	39.1
	(5.52)	(6.36)	(10.7)	**(7.15)**	**(5.27)**	(16.4)	(12.4)	(14.0)
Blocks	33.0	45.3	213	42.7	42.4	**38.1**	38.6	38.7
	(3.53)	(6.67)	(19.6)	(9.86)	(4.67)	**(4.74)**	(5.23)	(5.8)
Cubic	0.424	1.08	1.95	**0.667**	0.97	1.97	1.21	1.43
	(0.0969)	(0.355)	(0.452)	**(0.293)**	(0.237)	(0.965)	(0.673)	(0.795)

Signal	M4.2	M5.2	M6.2	M7.2	M8.2	M512	M512.1	M512.2
Doppler	0.641	0.649	0.664	0.673	0.678	0.785	0.913	0.702
	(0.115)	(0.117)	(0.124)	(0.126)	(0.125)	(0.0976)	(0.147)	(0.136)
Heavisine	41.1	44.6	46.0	48.3	50.4	**32.1**	86.2	53.5
	(14.2)	(14.8)	(15.2)	(17.1)	(17.7)	**(7.21)**	(22.7)	(18.4)
Blocks	39.6	38.6	38.4	38.5	38.4	60.0	39.2	**38.0**
	(6.48)	(6.02)	(5.61)	(5.55)	(5.66)	(10.7)	(5.55)	**(5.51)**
Cubic	1.74	2.00	2.08	2.29	2.38	**0.58**	5.51	3.01
	(0.957)	(1.04)	(1.08)	(1.16)	(1.19)	**(0.244)**	(1.22)	(1.27)

Table 6.4: Results from a simulation study comparing the MSE of different Cross Validation schemes on wavelet signals with SNR of 5. We tabulate the mean and standard deviation for each CV scheme from 100 trials. N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G. We have highlighted the best performing methods in bold for each.

Chapter 7

Wavelets for Irregularly-Spaced Data

The orthogonal DWT is not designed to work on time series whose time points are irregularly-spaced. However, data can be irregular time series: examples of this appear in astronomy data (Mondal & Percival 2012) and financial "tick" data (Dacorogna, Gencay, Muller, Olsen & Pictet 2001). High frequency data is naturally large in dimension, thus requiring fast implementations. Moreover, we derive a new optimal gridding scheme for interpolation which is computationally-demanding and would thus benefit from a parallel implementation.

In this chapter we give a brief review of the literature on wavelet methods for irregularly spaced data. Then we describe Kovac's (1998) interpolation scheme. We introduce some new extensions to this method to optimise the placement of the interpolation gridding. Whilst Kovac fixes a particular grid to any irregularly-spaced data, we believe that there is value in shifting the placement of the grid, provided we can derive a way of finding an optimal shift. In particular, we introduce a new entropy-based method which we demonstrate works well in simulation studies. This means we can build a data-adaptive interpolation grid for irregularly spaced data rather than setting a grid arbitrarily and thus we can minimise the error introduced by interpolation.

7.1 Review of the Literature

There are a number of papers in the wavelet literature that deal with the quandary of irregularly-spaced data, which can be divided into four categories:

1. Treating the data as though they are regularly spaced and applying the DWT as normal.
2. Interpolating the data onto a regular grid then proceeding as normal. In this case, one can keep track of the variance matrix of the coefficients to improve thresholding performance.
3. Calculating exact wavelet coefficients for the given time points. This does not preserve

the orthogonality of the standard wavelet transform.

4. Using second generation wavelets that are also non-orthogonal and lose the traditional scale properties. These are methods based on the lifting scheme.

There is an excellent review of the methods existent at the time by Nunes, Knight & Nason (2006).

7.1.1 Treating the Data as Though They Are Regularly Spaced

One proposed method of dealing with irregularly spaced data is to treat it as regularly spaced. There are likely to be issues with this approach because a time-frequency decomposition assumes that the data are regularly spaced. However, Cai & Brown (1999) prove that the DWT applied to what they call data with "random uniform design" (that is the irregular dispersion is uniformly distributed) with the universal threshold gives an estimator that is within a logarithmic factor of the minimax rate of convergence with a family of Hölder classes. In a previous paper, Cai & Brown (1998) overcome the issue of irregularly spaced data using a preconditioning matrix: however this method requires "fixed design" i.e. non randomly spaced data. Chicken (2003) proposes a block thresholding scheme for time series with uniformly-distributed time points. Porto, Morettin, Percival & Aubin (2009) consider irregularly spaced data with correlated noise, and threshold as though they are regularly spaced using level-dependent thresholds, provided a few assumptions on the first two moments of the time distribution and noise are satisfied. The key advantage of these methods treating the data as though they are regularly spaced is they enable us to use our full set of wavelet techniques. However, especially if the time points are not uniformly spaced, we will likely see artefacts as the wavelet transform is heavily reliant on equispaced points to make a valid time-frequency transformation. Theoretically, this is an invalid assumption to make even if the results are acceptable.

7.1.2 Interpolation

Interpolation allows us to quickly project an irregularly-spaced series onto a regular grid. This is attractive as it can be done quickly and then the full arsenal of wavelet techniques is available. However, potential issues are that data is not fully used and we lose detail in the highest frequencies as the interpolation can add or remove artefacts. In practice though we might discard the highest frequency wavelet coefficients as in a noisy setting these may be largely masked by noise, so losing detail by interpolation may not be problematic. Moreover, we can take into account of the interpolation during thresholding, so we are not throwing away the information. Arne Kovac studied this for his PhD thesis (Kovac 1998) and a subsequent paper (Kovac & Silverman 2000), in which he uses (respectively) simple but effective schemes of linear interpolation and linear regression. We explore these in more detail later on in Section 7.2. Hall

& Turlach (1997) use local averaging and interpolation on data with both stochastic and non-dyadic design. Antoniadis & Fan (2001) use a regularised interpolation scheme that minimises a Sobolev norm that gives low risk signal estimation for their Regularised One Step Estimator estimation. However, this involves an extra smoothing parameter s and if we do not know the smoothness of the underlying signal, it is unclear how to choose this.

7.1.3 Exact Wavelet Coefficients

Mondal & Percival (2012) use adaptive *Slepian*, non-orthogonal, wavelets that maximise the squared gain function in the frequency bands. They also provide estimators for the wavelet variance, despite the lack of orthogonality. Previous to this, they published work on wavelet variance analysis for "gappy" data i.e. regular data with missing values (Mondal & Percival 2008). Sardy, Percival & Bruce (1999) compare four different methods for handling unequally spaced data: *isometric* (which comes under the methods of Section 7.1.1), *sampling* (which is interpolation 7.1.2), a Haar-specific asymmetric extension and finally *exact integration* that derives wavelet filter coefficients from the CWT and belongs in this section. By picking exact wavelet coefficients according to spacing of the data we are theoretically sound, however this would not allow for fast computation. Indeed, one of the main benefits of the wavelet transform is its computational efficiency from the pyramid algorithm, so this method would not be useful in practice.

7.1.4 Second Generation Wavelets

Second generation wavelets are based on the lifting scheme (see Section 3) introduced by Sweldens (1998) which was initially stated to be compatible with irregular samples. He published two papers in the following year with Daubechies et al. that demonstrate lifting schemes involving subdivision that can be used to form a multiresolution analysis (Daubechies, Guskov, Schroder & Sweldens 1999, Daubechies, Guskov & Sweldens 1999). A subsequent paper specifies the MRA of subdivision schemes (Daubechies, Runborg & Sweldens 2004). A number of authors have also published papers on lifting schemes in a multidimensional setting using subdivision (Jansen, Nason & Silverman 2001, Valette & Prost 2004, Delouille, Jansen & von Sachs 2006): Jansen et al. (2001) add smoothing using Bayesian shrinkage; Valette & Prost (2004) extend MRA theory to irregularly spaced triangular 3D meshes; Delouille et al. (2006) develop 2D lifting schemes using second order neighbours and careful consideration of boundary coefficients. Then Vanraes, Jansen & Bultheel (2002) modify the 1D lifting scheme to better suit a set of irregularly spaced points, namely in minimising the update step coefficients, giving up vanishing moments but retaining prediction symmetry and changing the assignment of "odd" and "even" points when they are distributed such that the gap between neighbouring points is greater than the gap between pairs of odd or even points. Whilst this stabilises the bias on smoothed reconstructions, it is unclear how this scheme affects the convergence of the subdivision scheme and

hence whether we can define an MRA. Jansen, Nason & Silverman (2004) extend their previous work to formalise the method of "lifting one coefficient at a time" (LOCAAT) which disregards the idea of splitting the data into "odd" and "even" points. Instead, they pick one point at each level of transform to predict using its neighbours. They consider this methodology in a multi-dimensional setting, using least squares, Voronoi cells and trees to determine the neighbours of a point. Nunes et al. (2006) expand on the framework of LOCAAT to formalise an adaptive lifting scheme for nonparametric regression. This iteratively removes the point with the finest detail and predicts it using its neighbours. The method is adaptive in the order of regression used to predict the point and in the number of neighbours considered; the best combination is taken to give the smallest detail coefficient. This can nevertheless still be computer in $\mathcal{O}(n)$ time and invertibility of the transform is ensured by reversing the exact sequence of transform steps. This is a clear departure from the world of first generation wavelets as they discard the property of orthogonality of the wavelet and scaling functions, the transformation is no longer linear (it is dependent on the removal order and prediction choices) and we lose our discrete dyadic scale structure (although one can be artificially constructed). They calculate the affect of the transform on the variance structure of the coefficients but neglect any correlations in the off-diagonals. Knight & Nason (2008) extend this adaptive lifting scheme to a "nondecimated" lifting scheme: instead of picking points to remove with the smallest detail, they consider a large number P of paths of trajectories specifying the order in which to remove points. Thus, at a computational cost, they can produce a structural representation representing how often a wavelet coefficients appear in artificial scales, and they can perform wavelet shrinkage on each of the P transformations then average the reconstructions. They show this outperforms adaptive lifting in nonparametric regression. Knight, Nunes & Nason (2011) use these nondecimated lifting transformations to define a wavelet periodogram of irregularly spaced data. Knight, Nason & Nunes (2017) use the nondecimated LOCAAT transform and the artificial wavelet levels to estimate the Hurst long-memory parameter in the presence of irregularly spaced or missing data. Lastly, Hamilton, Nunes, Knight & Fryzlewicz (2018) introduces complex decimated and non-decimated LOCAAT which allow them to analyse phase in irregularly-spaced data and define a periodogram and cross-periodogram, as well as providing smooth reconstructions. Whilst these methods perform well in nonparametric regression, we are not able to use orthogonal wavelet basis functions and the associated theory that has been built from them including constructing a MRA and wavelet variance. Nevertheless, these methods have been developed beyond the material we present in this chapter.

7.2 Kovac's Interpolation Scheme

Kovac (1998, 2000) uses a linear matrix mapping from the unequally-spaced data to equally-spaced data. Then, as the wavelet transform can be performed by a matrix multiplication (see Section 2.2.1), this can be performed together with the linear mapping. As a result of this, we can derive the covariance matrix of the matrix coefficients to improve the thresholding

performance. In this section, we describe his interpolation scheme as in his thesis (Kovac 1998). Kovac also does this linear mapping using local linear regression (Kovac & Silverman 2000). The advantage of this alternative scheme is that it uses all the data: linear interpolation only uses the points either side of a regular grid point.

7.2.1 Problem Setup

We have a collection of pairs of values $\{y_i, t_i\}_i$ for $i = 0, \ldots, n-1$ for some integer n, where each y_i is a noisy evaluation of a function f:

$$y_i = f(t_i) + \xi_i, \quad \xi_i \sim N(0, \sigma^2). \tag{7.1}$$

The t_i are irregularly spaced on the interval $[0, 1]$, which is the domain of the function f. This is the same as Equation (2.22) on page 35, except that here our time points are irregularly spaced.

7.2.2 Kovac's Linear Interpolation Scheme

We can choose an integer J to define a grid of new, regularly-spaced, time points,

$$\tilde{t}_k = \left(k + \frac{1}{2}\right) 2^{-J}. \tag{7.2}$$

Kovac chooses J so that $n_R = 2^J$, the number of regular grid points, is the smallest power of two not smaller than n. Kovac calculates interpolations of the function at these new time points as follows:

$$\tilde{y}_k = \begin{cases} y_0 & \text{if } \tilde{t}_k < t_0, \\ y_{n-1} & \text{if } \tilde{t}_k > t_{n-1}, \\ y_i + \frac{(\tilde{t}_k - t_i)(y_{i+1} - y_i)}{(t_{i+1} - t_i)} & \text{if } t_i \leq \tilde{t}_k \leq t_{i+1}. \end{cases} \tag{7.3}$$

If there is more than one pair of time points (t_i, t_{i+1}) satisfying $t_i \leq \tilde{t}_k \leq t_{i+1}$, then he takes the smallest such pair. This is for computational convenience, and only happens if an irregular time point coincides with a regular gridded time point. See Figure (7.1) for an illustration of this interpolation scheme.

Since this interpolation scheme is a linear mapping, (7.3) can be written as a matrix operation:

$$\tilde{\boldsymbol{y}} = R\boldsymbol{y}, \tag{7.4}$$

for some matrix that carries out (7.3) for a particular $\boldsymbol{y}$ and grid size.

This is particularly neat, seeing as the wavelet transform itself can be performed using a matrix multiplication $\boldsymbol{d} = \mathcal{W}\tilde{\boldsymbol{y}}$ if $\mathcal{W}$ is a wavelet matrix. Then, by multiplying $\mathcal{W}$ and R, we can form

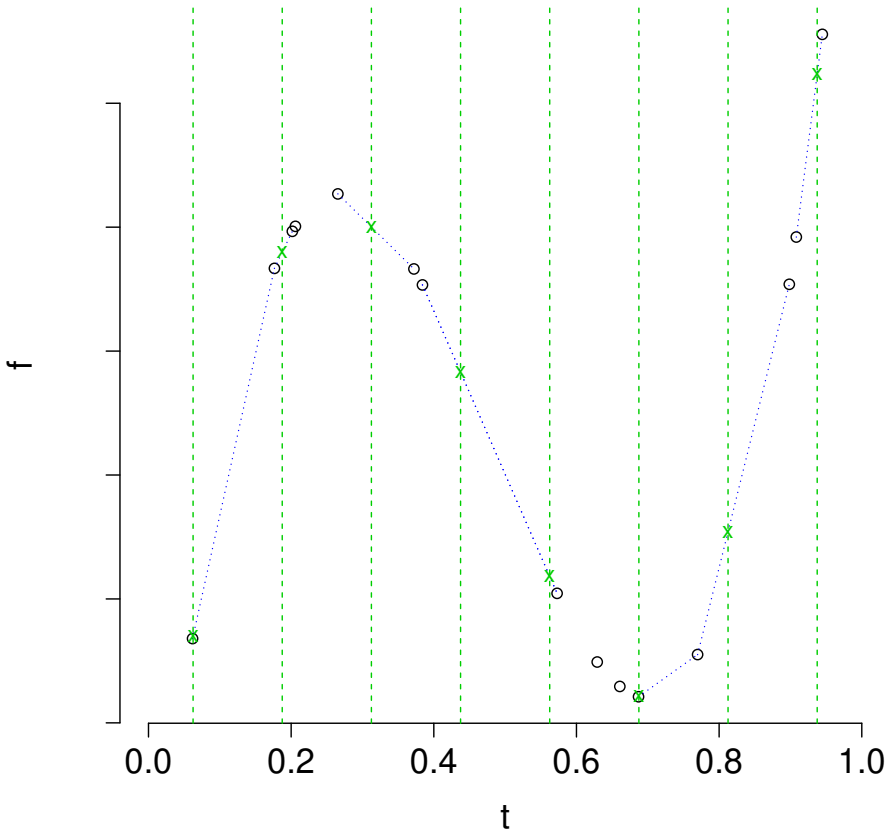


Figure 7.1: An illustration of Kovac's interpolation scheme on a smooth function. The circular points are irregularly spaced values $f(t_i)$. These are interpolated on a grid (dashed vertical lines) to regular points marked with a cross. The dotted lines between irregular (circular) points indicate those that are being interpolated. Note that not all irregular points are necessarily used to create the regular points.

a new matrix, say $\mathcal{W}_R = \mathcal{W}R$ that directly transforms and interpolates the original, irregular series, $\boldsymbol{y}$.

7.2.3 Smoothing

Kovac uses this single matrix transform to threshold the wavelet coefficients, $\boldsymbol{d}$, to give $\tilde{\boldsymbol{d}}$, before inverting the transform to give a smoothed estimate, $\hat{f}$, of f. However, because the transformation is non-orthogonal, the standard assumption of an i.i.d. normal noise as in Equation (2.23) on page 36 does not hold. He derives a coefficient-dependent threshold that uses the effect of the full transformation of linear mapping and wavelet transform on the variance of the wavelet coefficients.

If we initially consider the irregular data to have i.i.d. $N(0, \sigma^2)$ noise as in Equation (7.1) then the noise of the interpolated data will be normally distributed with zero mean and covariance matrix

$$\sigma^2 \Sigma^J = \sigma^2 R R^T = \Sigma, \tag{7.5}$$

using Kovac's notation.

From this, we can then calculate the variance of the wavelet coefficients as

$$\sigma^2 \mathcal{W} \Sigma^J \mathcal{W}^T. \tag{7.6}$$

Kovac has written fast algorithms for calculating the required elements of these matrices; these are mainly zero except for the diagonals.

Kovac does not, however, revert to a "smoothed" version of the original, irregular, series but to the regular grid. This is more useful for function estimation but not for seeing the effect of the smoothing on the original series.

7.2.4 Extensions

Whilst this method has been demonstrated to work well, there are some possible extensions that can be made:

- The number of regular grid points is chosen to be J *so that* 2^J *is the smallest power of two not smaller than* n. In his publications, Kovac states that the MSE decreases for increasing J until such a point that it levels off: for smaller values of J, the grid does not make use of all the irregularly spaced data; then it will use all points for J that gives a grid size smaller than $\min_i(t_{i+1} - t_i)$. If we are not using the standard DWT (we could, say, use the MODWT) then this number of grid points can be a tunable parameter according to either the distribution of time points in the irregular series or the smoothness of the function we are estimating. Moreover, the positioning of the grid points is fixed as in Equation (7.2).

- Kovac only considers inverting the wavelet transform to the regular, gridded points. Inverting to the irregular points is non-trivial, because the matrix $\mathcal{W}_R$ is not necessarily square and thus not generally invertible. Moreover, simple linear interpolation once again of the regular, already interpolated values, back onto the irregular time points will give approximation errors that would be undesirable once we have already smoothed the wavelet coefficients.

- Kovac states that linear interpolation/regression are not the only possible ways of calculating the values for the regular grid: splines, for example, would give smoother interpolated values.

7.3 Optimal Gridding

We would like to minimise the error due to interpolation by finding an optimal gridding scheme. Kovac found that increasing the number of grid points improved the performance of his scheme up to a certain J, where it would eventually converge. We introduce a generalisation of his grid points by allowing a horizontal shift in the positions of the $\tilde{t}_k$:

$$\tilde{t}_k = (k+s)\,2^{-J}, \text{ where } s \in [0,1]. \tag{7.7}$$

This generalisation of course coincides with (7.2) at $s = 0.5$. Thus we would like to "optimally" pick J and s, where we consider the "optimal" gridding to be the one that minimises the MSE of construction at these grid points $\tilde{t}_k$:

$$truemse = \frac{1}{n_r}\sum_{k=0}^{n_r-1}\left(f(\tilde{t}_k) - \hat{f}(\tilde{t}_k)\right)^2. \tag{7.8}$$

For a signal with no noise, our estimate $\hat{f}(\tilde{t}_k)$ is simply the interpolated value $\tilde{y}_k$. Clearly, in practice, we won't know the true function f, so we can't measure this explicitly. Let us define various cost measures that we might expect to be good predictors of a successful gridding scheme. Intuitively, we would like the position of our grid (the $\tilde{t}_k$) to align well with as many of the irregular points (the t_i) as possible; or, better still, we would like to align to the pairs (t_i, y_i) that are worst behaved in that they would contribute considerably to the MSE were they not matched. In this respect, Figure 7.1 appears to be a fairly reasonable: the function is well-behaved and a few of the $\tilde{t}_k$ align almost exactly with the t_i. We used three different cost measures to ascertain information about the error of interpolation; we call these *mdist*, *interpmse* and *mentropy*. These are defined as follows:

$$mdist = \frac{1}{n_r}\sum_{k=0}^{n_r-1}|\tilde{t}_k - t_{\eta(\tilde{t}_k)}|, \tag{7.9}$$

where $\eta(\tilde{t}_k) = \arg\min_i |t_i - \tilde{t}_k|$ denotes the nearest irregular point to the regular grid point. By minimising *mdist*, we hope to find a gridding that aligns well with the irregular points.

Remark 7.3.1. *During investigation on mdist, we noticed that it converges to some value for increasing n_r. In fact, we can calculate the asymptotic limit of mdist to be:*

$$\lim_{n_r\to\infty} mdist(\boldsymbol{t}) = \frac{1}{4}\sum_{i=1}^{n-2}[t_{i+1} - t_i]^2 + \frac{1}{2}\left(t_0^2 + (1 - t_n - 1)^2\right),$$

where the limit depends entirely on the spacing of the vector $\boldsymbol{t}$.

$$interpmse = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - y_i^\star)^2, \tag{7.10}$$

where $\boldsymbol{y}^\star$ is our signal $\boldsymbol{y}$ re-interpolated at the irregular time points $\boldsymbol{t}$ from the regular points $\tilde{\boldsymbol{y}}$; we want to grasp the error from linear interpolation with this measure.

$$mentropy = \sum_{m=0}^{M-1}\frac{d_m}{||\tilde{\boldsymbol{y}}||}\log\left(\frac{d_m}{||\tilde{\boldsymbol{y}}||}\right)^2, \tag{7.11}$$

for concatenated levels of re-indexed detail coefficients d_m, $m = 0, \ldots, M-1$. This is the Shannon entropy as defined in Equation (2.21) on page 35, which Coifman & Wickerhauser (1992) used to define a *best basis* of the DWPT (see Section 2.6). We are deliberately vague in our d_m notation because Taswell (1996) proposes only calculating a proportion of the wavelet packet library; in our case, we are simply considering the DWT rather than the DWPT and only calculate a partial transform. We found that two levels of detail coefficients were insufficient, but three levels gave good performance.

Hence, we would like to find values of J and s that minimise *truemse* by minimising one of *mdist*, *interpmse* and *mentropy*. Note the slight abuse of notation here. We are altering our grid points $\tilde{t}_k$ which vary with both J and s, and all of these measures could be considered functions of s, J, $\boldsymbol{t}$ and $\boldsymbol{y}$ - however, to incorporate all of these into the notation would be burdensome.

We found, like Kovac, that increasing the number of regular points n_r improved the gridding (see Figure 7.2), but because of the limitations of the DWT requiring it to be a power of two it is sensible to fix this at a reasonable value and focus on altering our shift s. So we fix n_r as Kovac did at the first power of two greater than n, i.e. $n = 2^J$, where $J = \text{ceil}(\log_2 n)$ (where $\text{ceil}(x)$ denotes the ceiling of x, the smallest integer greater than or equal to x). We also show in Figure 7.2 that there appears to be value in selecting an optimal value of shift s as we are limited in our choices of n_r.

7.3.1 Smoothing of the Irregularly-Spaced Data

Once we have performed the Kovac smoothing, we have estimates $\hat{f}(\tilde{t}_k)$, i.e. estimates of the true function evaluated at the regular grid points $\tilde{t}_k$. But what if we would like $\hat{f}(t_i)$, i.e. estimates of the true function evaluated at the original, irregularly-spaced points. Two options for "inverting" our transform are:

(i) Re-interpolate

(ii) Adjust y_i according to $\hat{f}(\tilde{t}_k)$

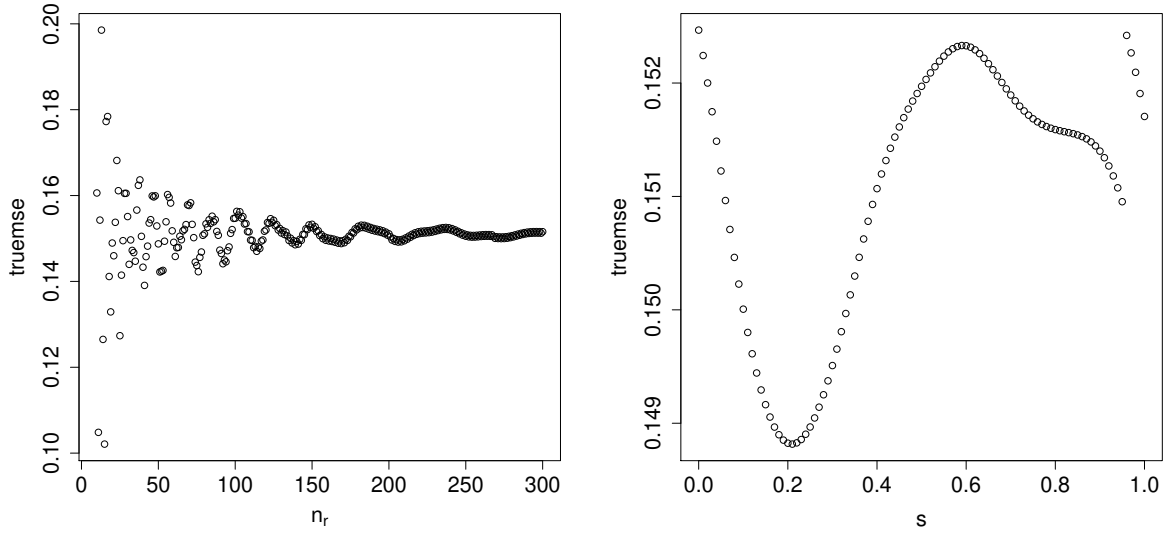


Figure 7.2: We illustrate the effect of changes to the gridding on interpolating a noise-free Doppler signal sampled at 100 irregularly-spaced time points. We plot truemse (the MSE at the regular grid points, (7.8)) against n_r, the number of regular grid points in the left hand plot. In the right hand plot, we fix $n_r = 128$ and plot truemse against s, horizontal shift of the grids.

In the first option, we re-interpolate our gridded data $(\tilde{t}_k, \hat{f}(\tilde{t}_k))$ back onto the original irregular time points to give $\hat{f}(t_i)$. We might be concerned here of introducing further errors from interpolating again to the irregular points on top of the initial errors from interpolating initially onto the regular grid. This is partly what we are trying to gauge with *interpmse* (7.10). However, if the gridding is somehow optimal, then we should not be accumulating too much error from interpolating twice.

In the second option, we consider adjusting y_i to form $\hat{f}(t_i)$ by considering the effect of smoothing the regularly spaced $\tilde{y}_k$ to $\hat{f}(\tilde{t}_k)$. i.e. to avoid interpolating again, we can just add an adjustment of the weighted averages of the two surrounding regular gridded thresholding adjustments.

$$\hat{f}(t_i) = y_i + \left[\left(\hat{f}(\tilde{t}_{k-1}) - \tilde{y}_{k-1} \right) \frac{(\tilde{t}_k - t_i)}{(\tilde{t}_k - \tilde{t}_{k-1})} + \left(\hat{f}(\tilde{t}_k) - \tilde{y}_k \right) \frac{(t_i - \tilde{t}_{k-1})}{(\tilde{t}_k - \tilde{t}_{k-1})} \right],$$

for i and k such that $\tilde{t}_{k-1} < t_i \leq \tilde{t}_k$.

7.3.2 Simulation Studies

We conducted two simulation studies: the first to see which of our three measures is best at minimising *truemse* and the second to compare the standard Kovac algorithm and our entropy-based gridding against other out-of-the-box methods in the context of signal estimation at the

original irregular t_i. In both studies, we consider the four wavelet test functions of doppler, heavisine, blocks and cubic (described in Section 2.3). We took $n = 100$ (irregular datapoints) and we fixed $n_r = 128$ (regular grid points), changing s in increments of 0.01 from 0 to 1. For *mentropy*, we used three levels of detail coefficients from an LA8 transform. We did 1000 trials.

Simulation Study 1: Minimising *truemse*

For our first simulation study, we just considered selecting the best grid for a set of irregularly-spaced t_i. We did not use wavelet transforms except in the calculation of *mentropy*, as we did not consider signal estimation here. We compared our three measures on four wavelet test functions with SNR of "infinite" (i.e. no noise) and 5 (i.e. clear noise).

We measured the value of *truemse* corresponding to the best gridding as determined by each of our three measures *mdist*, *interpmse* and *mentropy*, and compared this to the actual minimum of *truemse* for an oracle gridding as well as the value when using Kovac's standard gridding. We show the mean and standard deviation of these values for *truemse* in Table 7.2 for signals with no noise. Note that the differences between the methods averaged over all trials is slight. In Table 7.1, we count the number of trials that each method out of standard Kovac, *mdist*, *interpmse* and *mentropy* produced the smallest value of *truemse* for signals with no noise: we can see that all measures produced optimal griddings in some cases, and *mentropy* appears to be an overall winner. Note that the Doppler and Blocks signals are particularly difficult signals to recover from irregularly-spaced data: if the positions of the data points miss key areas (the start of the signal for Doppler where the frequency component is very high and the jumps in the Blocks signal) then no gridding will give good results.

Signals	Kovac	mdist	interpmse	mentropy
Doppler	111	**339**	268	298
Heavisine	64	355	144	**512**
Blocks	377	225	62	**413**
Cubic	45	184	280	**597**

Table 7.1: Results from a simulation study to find the best performing measure out of *mdist*, *interpmse* and *mentropy* in picking a grid to minimise *truemse* (the MSE at the regular grid points, (7.8)) for different signals with no noise. We compare these to the standard Kovac gridding too. The tabulated values are the number of times each gridding scheme produced the smallest *truemse* out of 1000 trials (note that there may be ties for first place so the rows do not sum to 1000). We have highlighted the best performing method in bold for each.

We give in Appendix D.2.1 the results for noisy signals with SNR of 5. It is clear from Tables D.5 and D.6 that this noise obfuscates our measures and worsens their performance for predicting a grid that minimises *truemse*.

Simulation Study 2: Smoothing Irregular Datapoints

In our second simulation study, we compared methods of smoothing the original, irregularly spaced, noisy data (t_i, y_i) to create estimates $(t_i, \hat{f}(t_i))$. We sampled 100 points from four standard wavelet test functions with added Gaussian noise giving a SNR of 5. In each case, we did 1000 trials. We tested Kovac's method using both reconstruction methods of Section 7.3.1. We compared these with two out-of-the-box methods. The details of competing methods are:

- kov.re : This is the standard Kovac algorithm for forming the $\tilde{y}_k$, which we thresholded using Kovac's modified SURE thresholding scheme to $\hat{f}(\tilde{t}_k)$. We then reconstructed the $\hat{f}(t_i)$ using re-interpolation.
- kov.adj: As above, but reconstructing $\hat{f}(t_i)$ using the adjustment method.
- bg.re: The Kovac algorithm with entropy-based best gridding to form the $\tilde{y}_k$. These were thresholded using Kovac's modified SURE thresholding scheme to $\hat{f}(\tilde{t}_k)$. The $\hat{f}(t_i)$ were then reconstructed using re-interpolation.
- bg.adj: As above, but with using the adjustment method for reconstruction of the $\hat{f}(t_i)$.
- spl.smooth: Spline smoothing, with smoothing parameter set by GCV.
- ker.smooth: Kernel smoothing with Nadaraya-Watson kernel-weighted average using Gaussian kernel. We set a bandwidth of 0.1, which modified the quantiles of the kernel.

The results of the simulation study can be found in Table 7.3. Firstly, we notice that the kernel smoothing method is not well-suited to the wavelet test signals and the lack of automatic parameter selection inhibits its performance out-of-the-box. We picked a bandwidth that gave some visually appealing smoothing. The spline smoothing on the other hand performed very well thanks to its ability to naturally work with irregularly spaced data and automatic smoothing parameter selection. Unsurprisingly, the spline smoothing method was by far the best for the cubic signal. Turning to the Kovac methods, we see little impact of our two reconstruction schemes; however, the different gridding schemes give different results. The entropy-based best gridding scheme performs well in the Doppler and Blocks signals, where it is important to position the grids to capture fast changes in the signal. However, in the Heavisine and Cubic signals, the standard Kovac gridding was superior - possibly because the best gridding scheme overfitted points with large noise components. Indeed, if we increase the SNR to 10, we see that the best gridding then beats standard Kovac gridding for the Heavisine and also proves to be superior to spline smoothing for the Doppler and Blocks signals. See Table D.7 in Appendix D.2.2 for results of the study with SNR of 10.

Signals	Oracle	Kovac	mdist	interpmse	mentropy
Doppler	0.08517 (0.002824)	0.08629 (0.002865)	**0.08604 (0.002882)**	0.08613 (0.002841)	0.08609 (0.002902)
Heavisine	11.21 (0.1282)	11.23 (0.1305)	**11.22 (0.1283)**	11.23 (0.1301)	**11.22 (0.128)**
Blocks	5.581 (0.3706)	5.623 (0.3761)	5.628 (0.3727)	5.639 (0.3725)	**5.621 (0.3701)**
Cubic	0.6252 (0.001875)	0.6256 (0.002034)	0.6255 (0.001916)	0.6256 (0.002077)	**0.6253 (0.001893)**

Table 7.2: Results from a simulation study on irregularly spaced data from four wavelet signals with no noise. We tried to minimise *truemse* (the MSE at the regular grid points, (7.8)) using four schemes for selecting grid points: standard Kovac gridding and those minimising measures *mdist*, *interpmse* and *mentropy*. We tabulate mean and standard deviation (in parentheses) of the achieved *truemse* for each of these gridding schemes alongside the oracle gridding scheme. We have highlighted the best performing methods in bold for each.

Signals	kov.re	kov.adj	bg.re	bg.adj	spl.smooth	ker.smooth
Doppler	0.8376 (0.208)	0.8704 (0.2376)	**0.3828 (0.1095)**	**0.3777 (0.1036)**	**0.3525 (0.1602)**	4.938 (2.399)
Heavisine	18.18 (5.119)	18.34 (5.301)	28.08 (6.01)	28.11 (6.029)	**9.204 (2.709)**	183 (153.9)
Blocks	52.21 (18.86)	52.23 (18.51)	**19.68 (5.847)**	**19.42 (5.725)**	**20.5 (8.813)**	198.6 (75.13)
Cubic	0.5196 (0.2023)	0.5198 (0.2005)	1.359 (0.2931)	1.361 (0.2918)	0.2195 (0.122)	**2.555 (2.253)**

Table 7.3: Results from a simulation study smoothing irregularly spaced datapoints with SNR of 5. We tabulate mean and standard deviation (in parentheses) of the MSE calculated at the irregular datapoints of six competing methods: kov.re, Kovac's wavelet interpolation with re-interpolation onto the irregular points; kov.adj, Kovac's wavelet interpolation with adjustments to reconstruct the irregular points; bg.re, entropy-based best gridding for Kovac's wavelet interpolation with re-interpolation; bg.adj, entropy-based best gridding with adjustments; spline smoothing; kernel smoothing. We have highlighted the best performing methods in bold for each.

Chapter 8

Conclusion

In this thesis, we have explored GPU-accelerated wavelet transforms for the analysis of high frequency data. In Chapter 5, we presented a GPU-accelerated library, WaveCUDA, with considerable speedups. This library utilises new lifting scheme algorithms which we developed in Chapter 3. This library is well-suited for use with large datasets, as we are likely to see in high frequency data. Another property of high frequency data is irregular spacing: we addressed this topic in Chapter 7. Our main contribution is the development of WaveCUDA, whilst other contributions include a new wavelet cross validation scheme (outlined in Chapter 6) and a new interpolation scheme for irregularly spaced data (introduced in Chapter 7). Both the cross validation and interpolation schemes are computationally intensive and justify the use of fast implementations, which is all the more necessary with large datasets.

8.0.1 Further Work

We gave lifting scheme algorithms for four common wavelet filters in Chapter 3, however these are by no means the only available wavelet filters. We considered orthogonal wavelet filters up to length 8; we could factorise longer filters which are used in applications requiring more vanishing moments, for example. An interesting area of research could be to consider lifted wavelets that use non-dyadic filters (e.g. the discrete orthogonal M-band decompositions of Steffen & Heller (1993)) that scale appropriately to fit the structure of any given data series: in reality, few real datasets actually match wavelets' dyadic design.

We gave an outline of the functionality of the WaveCUDA R package in Chapter 5. There are also a few functions that we are yet to implement in CUDA. The DWPT is more expensive to compute than the DWT and the associated best basis algorithm could benefit from a CUDA implementation. We have not yet written code to perform SURE thresholding, which is one of the most popular thresholding schemes. Our new cross validation scheme could be vastly sped up with a CUDA implementation. Lastly, we have not explored wavelets in higher dimensions,

where further parallelisation can be exploited to gain immense speedups.

We did not consider primary resolution in Chapter 6. Setting a specific primary resolution would be very straightforward as it would simply require devoting more rows in the wavelet matrix to scaling coefficients and then optimising over fewer (detail) coefficients. However, selecting primary resolution via our V-fold cross validation would be time consuming if we adopted the grid search method of Nason (2002). We could even evaluate all thresholds and primary resolutions for hard thresholding, as $n-1$ detail coefficients only gives us a maximum of $n-1$ different thresholds. We would also like to derive a soft thresholding estimator, as currently we are only selecting thresholds suitable for hard thresholding. Lastly, we would see a big improvement from a MODWT version of our method. This would however require some modification of our method as we would no longer have an orthogonal wavelet matrix. We could use a Moore-Penrose generalised inverse, for which Deng (2011) has adapted the Sherman Morrison formula. An immediate extension of our Delete-d CV scheme would be in imputing or "interpolating" missing values using the wavelet basis rather than simple linear interpolation, for example. We should be able to redefine our matrix $\mathcal{W}_p$ to be missing the rows corresponding to the first (or first available) detail coefficient corresponding to each missing value. Then we could reconstruct $\boldsymbol{y}$ as in the delete-d scheme, ready to use in some signal estimation application.

In Chapter 7, it would be straightforward to apply our optimal gridding using MODWT in the entropy calculation, allowing us to search over different values of n that are not just powers of two as well as varying the shift s. It would be interesting to see how this compares to state-of-the-art optimal gridding schemes outside of the wavelet literature. Kovac's signal estimation scheme could then be modified to create a fast algorithm for calculating the variance of the interpolated MODWT wavelet coefficients. Furthermore, a GPU implementation of our best grid search would ensure it can be computed quickly: entropy calculations for different values of n and s could be computed in parallel, and combined with GPU wavelet implementations.

Bibliography

Abramovich, F. & Benjamini, Y. (1996), 'Adaptive thresholding of wavelet coefficients', *Comput. Stat. Data Anal.* **22**(4), 351–361.

Abramovich, F., Sapatinas, T. & Silverman, B. W. (1998), 'Wavelet thresholding via a Bayesian approach', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **60**(4), 725–749.

Allan, D. W. (1966), 'Statistics of Atomic Frequency Standards', *Proc. IEEE* **54**(2), 221–230.

Amato, U., Antoniadis, A. & Pensky, M. (2006), 'Wavelet kernel penalized estimation for non-equispaced design regression', *Stat. Comput.* **16**(1), 37–55.

Amato, U. & Vuza, D. (1998), 'Wavelet simultaneous approximation from samples affected by noise', *Comput. Math. with Appl.* **36**(5), 101–111.

Amina, M., Kodogiannis, V., Petrounias, I. & Tomtsis, D. (2012), 'A hybrid intelligent approach for the prediction of electricity consumption', *Int. J. Electr. Power Energy Syst.* **43**(1), 99–108.

Antoniadis, A. & Fan, J. (2001), 'Regularization of wavelet approximations', *J. Am. Stat. Assoc.* **96**(455), 939–967.

Antoniadis, A., Gregoire, G. & McKeague, I. (1994), 'Wavelet methods for curve estimation', *J. Am. Stat. Assoc.* **89**(428), 1340–1353.

Arlot, S. & Celisse, A. (2010), 'A survey of cross-validation procedures for model selection', *Stat. Surv.* **4**, 40–79.

Arlot, S. & Lerasle, M. (2014), 'Why V = 5 is enough in V -fold cross-validation', *The arXiv* (1).

Barber, S. & Nason, G. P. (2004), 'Real nonparametric regression using complex wavelets', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **66**(4), 927–939.

Bates, D. & Maechler, M. (2015), *Matrix: Sparse and Dense Matrix Classes and Methods.* R package version 1.2-2.
URL: *http://CRAN.R-project.org/package=Matrix*

Bertsekas, D. (1999), *Nonlinear programming*, 2 edn, Athena Scientific, Belmont.

Brockwell, A. & Davis, R. (1991), *Time Series: Theory and Methods*, 2 edn, Springer-Verlag, New York.

Bruce, A., Gao, H.-Y. & Stuetzle, W. (1999), 'Subset-selection and ensemble methods for wavelet de-noising', **9**, 167–182.

Buckner, J., Seligman, M. & Wilson, J. (2013), 'gputools: A few GPU enabled functions'. R package version 0.28.
URL: *http://cran.r-project.org/package=gputools*

Cai, T. & Brown, L. (1998), 'Wavelet shrinkage for nonequispaced samples', *Ann. Stat.* **26**(5), 1783–1799.

Cai, T. & Brown, L. (1999), 'Wavelet estimation for samples with random uniform design', *Stat. Probab. Lett.* **42**(3), 313–321.

Černá, D., Finěk, V. & Najzar, K. (2008), 'On the exact values of coefficients of coiflets', *Cent. Eur. J. Math.* **6**(1), 159–169.

Chicken, E. (2003), 'Block thresholding and wavelet estimation for nonequispaced samples', *J. Stat. Plan. Inference* **116**, 113–129.

Cho, H. & Fryzlewicz, P. (2015), 'Multiple change-point detection for high-dimensional time series via sparsified binary segmentation', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **77**(2), 475—-507.

Claypoole, R., Baraniuk, R. & Nowak, R. (1998), 'Adaptive wavelet transforms via lifting', *Proc. 1998 IEEE Int. Conf. Acoust. Speech Signal Process.* **3**, 1513–1516.

Cohen, A., Daubechies, I. & Feauveau, J.-C. (1992), 'Biorthogonal bases of compactly supported wavelets', *Commun. Pure Appl. Math.* **XLV**(1992), 485–560.

Cohen, A., Daubechies, I. & Vial, P. (1993), 'Wavelets on the interval and fast wavelet transforms', *Appl. Comput. Harmon. Anal.* **1**, 54–81.

Cohen, E. & Walden, A. (2010), 'A statistical study of temporally smoothed wavelet coherence', *IEEE Trans. Signal Process.* **58**(6), 2964–2973.

Coifman, R. R. & Donoho, D. L. (1995), Translation invariant de-noising, *in* A. Antoniadis & G. Oppenheim, eds, 'Wavelets Stat. Lect. Notes Stat.', Vol. 103, Springer New York, New York, pp. 125–150.

Coifman, R. R. & Wickerhauser, M. V. (1992), 'Entropy-based algorithms for best basis selection', *IEEE Trans. Inf. Theory* **38**(2), 713–718.

Constantine, W. & Percival, D. (2013), *wmtsa: Wavelet Methods for Time Series Analysis*. R package version 2.0–0.
URL: *http://cran.r-project.org/package=wmtsa*

Craven, P. & Wahba, G. (1979), 'Smoothing noisy data with spline functions - estimating the correct degree of smoothing by the method of generalized cross-validation', *Numer. Math.* **31**(4), 377–403.

Dacorogna, M., Gencay, R., Muller, U., Olsen, R. & Pictet, O. (2001), *An Introduction to High-Frequency Finance*, 1st edn, Academic Press Inc, San Diego.

Daubechies, I. (1988), 'Orthonormal bases of compactly supported wavelets', *Commun. Pure Appl. Math.* **XLI**(0).

Daubechies, I. (1992), *10 Lectures On Wavelets*, 8th edn, Society for Industrial and Applied Mathematics, Philadelphia.

Daubechies, I. (1993), 'Orthonormal bases of compactly supported wavelets II. variations on a theme', *SIAM J. Math. Anal.* **24**(2), 499–519.

Daubechies, I., Guskov, I., Schroder, P. & Sweldens, W. (1999), 'Wavelets on irregular point sets', *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* **357**(1760), 2397–2413.

Daubechies, I., Guskov, I. & Sweldens, W. (1999), 'Regularity of irregular subdivision', *Constr. Approx.* **15**, 381–426.

Daubechies, I., Runborg, O. & Sweldens, W. (2004), 'Normal multiresolution approximation of curves', *Constr. Approx.* **20**(3), 399–463.

Daubechies, I. & Sweldens, W. (1998), 'Factoring wavelet transforms into lifting steps', *J. Fourier Anal. Appl.* **4**(3), 247–269.

Delouille, V., Jansen, M. & von Sachs, R. (2006), 'Second-generation wavelet denoising methods for irregularly spaced data in two dimensions', *Signal Processing* **86**(7), 1435–1450.

Deng, C. Y. (2011), 'A generalization of the Sherman-Morrison-Woodbury formula', *Appl. Math. Lett.* **24**(9), 1561–1564.

Donoho, D. (1995), 'Nonlinear solution of linear inverse problems by waveletvaguelette decomposition', *Appl. Comput. Harmon. Anal.* **2**(2), 101–126.

Donoho, D. & Johnstone, I. (1995), 'Adapting to unknown smoothness via wavelet shrinkage', *J. Am. Stat. Assoc.* **90**(432), 1200–1224.

Donoho, D. & Johnstone, J. (1994), 'Ideal spatial adaptation by wavelet shrinkage', *Biometrika* **81**(3), 425–455.

Dowle, M. & Srinivasan, A. (2018), *data.table: Extension of 'data.frame'*. R package version 1.11.4.
URL: *https://CRAN.R-project.org/package=data.table*

Eckley, I. a. & Nason, G. P. (2014), 'Spectral correction for locally stationary Shannon wavelet processes', *Electron. J. Stat.* **8**(1), 184–200.

Eckley, I. A., Nason, G. P. & Treloar, R. L. (2010), 'Locally stationary wavelet fields with application to the modelling and analysis of image texture', *J. R. Stat. Soc. Ser. C (Applied Stat.* **59**(4), 595–616.

Efron, B. (1982), *The Jacknife, the Bootstrap and Other Resampling Plans*, Society for Industrial and Applied Mathematics, Philadelphia.

Egidi, N. & Maponi, P. (2006), 'A Sherman-Morrison approach to the solution of linear systems', *J. Comput. Appl. Math.* **189**(1-2), 703–718.

Feng, Z., Liang, M. & Chu, F. (2013), 'Recent advances in time-frequency analysis methods for machinery fault diagnosis: A review with application examples', *Mech. Syst. Signal Process.* **38**(1), 165–205.

Fryzlewicz, P. (2007), 'Unbalanced Haar technique for nonparametric function estimation', *J. Am. Stat. Assoc.* **102**(480), 1318–1327.

Fryzlewicz, P. (2013), 'Wild Binary Segmentation for multiple change-point detection', *Ann. Stat.* **42**(6), 1–32.

Fryzlewicz, P. & Nason, G. P. (2006), 'Haar-Fisz estimation of evolutionary wavelet spectra', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **68**(4), 611–634.

Fryzlewicz, P. & Ombao, H. (2009), 'Consistent classification of nonstationary time series using stochastic wavelet representations', *J. Am. Stat. Assoc.* **104**(485), 299–312.

Fryzlewicz, P. & Timmermans, C. (2015), 'SHAH: SHape-Adaptive Haar wavelets for image processing', *J. Comput. Graph. Stat.* **8600**(August), 00–00.

Gao, H.-Y. & Bruce, A. (1997), 'Waveshrink with firm shrinakge', *Stat. Sin.* **7**(4), 855–874.

Gencay, R., Selçuk, F. & Whitcher, B. (2001), *An Introduction to Wavelets and Other Filtering Methods in Finance and Economics*, Academic Press Inc.

Girardi, M. & Sweldens, W. (1997), 'A new class of unbalanced Haar wavelets that form an unconditional basis for Lp on general measure spaces', *J. Fourier Anal. Appl.* **3**(4), 457–474.

Grossmann, A. & Morlet, J. (1984), 'Decomposition of Hardy functions into square integrable wavelets of constant shape', *SIAM J. Math. Anal.* **15**(4).

Haar, A. (1910), 'Zur theorie der orthogonalen funktionen systeme', *Math. Ann.* **69**(3), 331–371.

Hall, P. & Turlach, B. (1997), 'Interpolation methods for nonlinear wavelet regression with irregularly spaced design', *Ann. Stat.* **25**(5), 1912–1925.

Hamilton, J., Nunes, M. A., Knight, M. I. & Fryzlewicz, P. (2018), 'Complex-Valued Wavelet Lifting and Applications', *Technometrics* **60**(1), 48–60.

Hastie, T., Tibshirani, R. & Friedman, G. (2009), *The Elements Of Statistical Learning*, 2 edn, Springer-Verlag, New York.

Ho, J. & Hwang, W.-L. (2013), 'Wavelet Bayesian network image denoising.', *IEEE Trans. Image Process.* **22**(4), 1277–90.

Hopf, M. & Ertl, T. (1999), Hardware based wavelet transformations, *in* 'Vision, Model. Vis.', pp. 317–328.

Huang, H. Y., Ombao, H. & Stoffer, D. S. (2004), 'Discrimination and classification of nonstationary time series using the SLEX model', *J. Am. Stat. Assoc.* **99**(467), 763–774.

Hurvich, C. & Tsai, C. (1998), 'A crossvalidatory AIC for hard wavelet thresholding in spatially adaptive function estimation', *Biometrika* **85**(3), 701–710.

Ikuzawa, T., Ino, F. & Hagihara, K. (2016), 'Reducing memory usage by the lifting-based discrete wavelet transform with a unified buffer on a GPU', *J. Parallel Distrib.Comput* **94**, 44–55.

Intel (2016), Intel 64 and IA-32 architectures optimization reference manual, Technical report, Intel.
URL: *http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf*

Jansen, M., Malfait, M. & Bultheel, A. (1997), 'Generalized cross validation for wavelet thresholding', *Signal Processing* **56**(1).

Jansen, M., Nason, G. & Silverman, B. (2001), 'Scattered data smoothing by empirical Bayesian shrinkage of second-generation wavelet coefficients', *Wavelet Appl. Signal Image Process. IX Proc. SPIE* **4478**, 87–97.

Jansen, M., Nason, G. & Silverman, B. (2004), 'Multivariate nonparametric regression using lifting', *Tech. Rep. 0417, Dep. Math. Univ. Bristol, UK* .

Jensen, A. & la Cour-Harbo, A. (2001), *Ripples In Mathematics*, 1 edn, Springer-Verlag, Berlin, Heidelberg.

Jiang, W. & Ortega, A. (1999), Parallel architecture for the discrete wavelet transform based on the lifting factorization, *in* 'Proc. SPIE 3817, Parallel Distrib. Methods Image Process. III', Vol. 3817, pp. 2–13.

Johnstone, I. & Silverman, B. (1997), 'Wavelet threshold estimators for data with correlated noise', *J. R. Stat. Soc. Ser. B (Statistical Methodol.)* **59**(2), 319–351.

Killick, R., Eckley, I. a. & Jonathan, P. (2013), 'A wavelet-based approach for detecting changes in second order structure within nonstationary time series', *Electron. J. Stat.* **7**, 1167–1183.

Kingsbury, N. (2001), 'Complex wavelets for shift invariant analysis and filtering of signals', *Appl. Comput. Harmon. Anal.* **10**(3), 234–253.

Knight, M. I., Nason, G. P. & Nunes, M. a. (2017), 'A wavelet lifting approach to long-memory estimation', *Stat. Comput.* **27**(6), 1453–1471.

Knight, M. & Nason, G. (2008), 'A nondecimated lifting transform', *Stat. Comput.* **19**(1), 1–16.

Knight, M., Nunes, M. & Nason, G. (2011), 'Spectral estimation for locally stationary time series with missing observations', *Stat. Comput.* **22**(4), 877–895.

Kovac, A. (1998), Wavelet thresholding for unequally spaced data, PhD Thesis, Bristol.

Kovac, A. & Silverman, B. (2000), 'Extending the scope of wavelet regression methods by coefficient-dependent thresholding', *J. Am. Stat. Assoc.* **95**(449), 172–183.

Lange, K. (2010), *Numerical Analysis for Statisticians*, 2 edn, Springer-Verlag, New York.

Lei, B., Soon, I., Zhou, F., Li, Z. & Lei, H. (2012), 'A robust audio watermarking scheme based on lifting wavelet transform and singular value decomposition', *Signal Processing* **92**(9), 1985–2001.

Lilly, J. M. & Olhede, S. C. (2010), 'On the analytic wavelet transform', *IEEE Trans. Inf. Theory* **56**(8), 4135–4156.

Lilly, J. M. & Olhede, S. C. (2012), 'Generalized morse wavelets as a superfamily of analytic wavelets', *IEEE Trans. Signal Process.* **60**(11), 6036–6041.

Liu, P. C. (1994), 'Wavelet spectrum analysis and ocean wind waves', *Wavelets Geophys.* **4**, 151–166.

Mallat, S. (1989*a*), 'A theory for multiresolution signal decomposition: the wavelet representation', *IEEE Trans. Pattern Anal. Mach. Intell.* **11**(7).

Mallat, S. (1989*b*), 'Multiresolution approximations and wavelet orthonormal bases of $L^2(\mathbb{R})$', *Trans. Am. Math. Soc.* **315**(1), 69–69.

Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. (2015), 'Maple'.

Maslen, M. & Abbott, P. (2000), 'Automation of the lifting factorisation of wavelet transforms', *Comput. Phys. Commun.* **127**(2-3), 309–326.

McCoy, E. (1999), Wavelet regression: a penalty function approach, *in* 'Proceeding 52nd Sess. Int. Stat. Inst.', pp. 2–5.

McCoy, E. & Konstantinou, K. (2002), 'Wavelet basis selection via cross-validation', *Working Paper, Imperial College London* pp. 1–11.

McGinnity, K., Varbanov, R. & Chicken, E. (2017), 'Cross-validated wavelet block thresholding for non-Gaussian errors', *Comput. Stat. Data Anal.* **106**, 127–137.

McQuarie, A. & Tsai, C.-L. (1998), *Regression and Time Series Model Selection*, 1 edn, World Scientific, Singapore.

Meyer, Y. (1986), 'Principe dincertitude, bases hilbertiennes et algèbres d'opérateurs', *Séminaire Bourbaki* (662), 209–223.

Mondal, D. & Percival, D. (2008), 'Wavelet variance analysis for gappy time series', *Ann. Inst. Stat. Math.* **62**(5), 943–966.

Mondal, D. & Percival, D. (2012), Slepian wavelet variances for regularly and irregularly sampled time series, *in* E. Feigelson & G. Jogesh Babu, eds, 'Stat. Challenges Mod. Astron. V, Lect. Notes Stat.', Springer New York, New York, chapter 38, pp. 403–418.

Montana, G., Nash, P. & Szeremi, V. (2013), 'HiPLARM: High Performance Linear Algebra in R'. R package version 0.1.
URL: *http://cran.r-project.org/web/packages/HiPLARM*

Morlet, J., Arens, G., Fourgeau, E. & Glard, D. (1982), 'Wave propagation and sampling theory-part I: complex signal and scattering in multilayered media', *Geophysics* **47**(2), 203–221.

Nason, G. (1995), Choice of the threshold parameter in wavelet function estimation, *in* A. Antoniadis & G. Oppenheim, eds, 'Lect. Notes Stat. Wavelets Stat.', Vol. 103, Springer-Verlag, New York, pp. 261–280.

Nason, G. (1996), 'Wavelet shrinkage using cross-validation', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **58**(2), 463–479.

Nason, G. (2002), 'Choice of wavelet smoothness, primary resolution and threshold in wavelet shrinkage', *Stat. Comput.* **12**, 219–227.

Nason, G. (2008), *Wavelet Methods In Statistics With R*, 1 edn, Springer, New York.

Nason, G. (2013*a*), 'A test for second-order stationarity and approximate confidence intervals for localized autocovariances for locally stationary time series', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **75**(5), 879–904.

Nason, G. (2013*b*), *wavethresh: Wavelets statistics and transforms*. R package version 4.6.6.
URL: *http://cran.r-project.org/package=wavethresh*

Nason, G., von Sachs, R. & Kroisandt, G. (2002), 'Wavelet processes and adaptive estimation of the evolutionary wavelet spectrum', *J. R. Stat. Soc. Ser. B (Statistical Methodol.* **62**(2), 271–292.

Nelson, J. D., Gibberd, a. J., Nafornita, C. & Kingsbury, N. (2017), 'The locally stationary dual-tree complex wavelet model', *Stat. Comput.* (October), 1–16.

Norwood, B. & Killick, R. (2018), 'Long memory and changepoint models: a spectral classification procedure', *Stat. Comput.* **28**(2), 291–302.

Nunes, M., Knight, M. & Nason, G. (2006), 'Adaptive lifting for nonparametric regression', *Stat. Comput.* **16**(2), 143–159.

NVIDIA Corporation (2014*a*), *Cuda C programming guide*, NVIDIA Corporation.

NVIDIA Corporation (2014*b*), *CUDA Compiler Driver NVCC*, NVIDIA Corporation.

Oh, H. S., Kim, D. & Lee, Y. (2009), 'Cross-validated wavelet shrinkage', *Comput. Stat.* **24**(3), 497–512.

Olhede, S. C. & Metikas, G. (2009), 'The monogenic wavelet transform', *IEEE Trans. Signal Process.* **57**(9), 3426–3441.

Olhede, S. C. & Walden, A. T. (2004*a*), ''Analytic' wavelet thresholding', *Biometrika* **91**(4), 955–973.

Olhede, S. & Walden, a. T. (2004*b*), 'The Hilbert spectrum via wavelet projections', *Proc. R. Soc. A Math. Phys. Eng. Sci.* **460**(2044), 955–975.

Ombao, H. C., Raz, J. a., von Sachs, R. & Malow, B. a. (2001), 'Automatic statistical analysis of bivariate nonstationary time series', *J. Am. Stat. Assoc.* **96**(454), 543–560.

Ombao, H., Raz, J., Von Sachs, R. & Guo, W. (2002), 'The SLEX model of a non-stationary random process', *Ann. Inst. Stat. Math.* **54**(1), 171–200.

Page, D. (2009), *Practical Introduction to Computer Architecture*, 1 edn, Springer, London.

Park, T., Eckley, I. & Ombao, H. (2014), 'Estimating time-evolving partial coherence between signals via multivariate locally stationary wavelet processes', *IEEE Trans. Signal Process.* **62**(20), 5240–5250.

Percival, D. (2002), 'An introduction to the wavelet variance and its statistical properties'.

Percival, D. B. & Guttorp, P. (1994), 'Long-memory processes, the Allan variance and wavelets', *Wavelets Geophys.* **4**, 325–344.

Percival, D. P. (1995), 'On estimation of the wavelet variance', *Biometrika* **82**(3), 619.

Percival, D. & Walden, A. (2000), *Wavelet Methods For Time Series Analysis*, 1st edn, Cambridge University Press, Cambridge.

Petersen, K. & Pedersen, M. (2012), *The Matrix Cookbook*, november edn.

Plett, M. I. (2007), 'Transient detection with cross wavelet transforms and wavelet coherence', *IEEE Trans. Signal Process.* **55**(5 I), 1605–1611.

Porto, R., Morettin, P., Percival, D. & Aubin, E. (2009), 'Wavelet Shrinkage for Regression Models with Random Design and Correlated Errors', *Appl. Phys.* pp. 1–23.

Press, W., Teukolsky, S., Vetterling, W. & Flannery, B. (2007), *Numerical Recipes*, 3 edn, Cambridge University Press, Cambridge.

Quan, T. M. & Jeong, W.-k. (2016), 'A fast discrete wavelet transform using hybrid parallelism on GPUs', *IEEE Trans. Parallel Distrib. Syst.* **27**(11), 3088–3100.

R Core Team (2015), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL: *https://www.R-project.org/*

Revolution Analytics & Weston, S. (2014), *foreach: Foreach looping construct for R.* R package version 1.4.2.
URL: *http://CRAN.R-project.org/package=foreach*

Revolution Analytics & Weston, S. (2015), *doParallel: Foreach Parallel Adaptor for the 'parallel' Package.* R package version 1.0.10.
URL: *http://CRAN.R-project.org/package=doParallel*

Sang, Y.-F. (2013), 'A review on the applications of wavelet transform in hydrology time series analysis', *Atmos. Res.* **122**, 8–15.

Sardy, S., Percival, D. & Bruce, A. (1999), 'Wavelet shrinkage for unequally spaced data', *Stat. Comput.* **9**, 65–75.

Selesnick, I., Baraniuk, R. & Kingsbury, N. (2005), 'The dual-tree complex wavelet transform', *IEEE Signal Process. Mag.* **22**(6), 123–151.

Serroukh, A., Walden, A. & Percival, D. (2000), 'Statistical properties and uses of the wavelet variance estimator for the scale analysis of time series.', *J. Am. Stat. Assoc.* **95**(449), 184–196.

Shao, J. (1993), 'Linear model selection by cross-validation', *J. Am. Stat. Assoc.* **88**(422), 486–494.

Stallman, R. M. & the GCC Developer Community (2015), *Using the GNU Compiler Collection*, GNU Software Press.

Steffen, P. & Heller, P. (1993), 'Theory of regular M-band wavelet bases', *IEEE Trans. Signal Process.* **41**(12), 3497–3511.

Strang, G. & Nguyen, T. (1997), *Wavelets and Filter Banks*, 2nd edn, Wellesley-Cambridge Press, Wellesley.

Sweldens, W. (1995), 'The lifting scheme: a new philosophy in biorthogonal wavelet constructions', *Wavelet Appl. Signal Image Process. III* pp. 68–79.

Sweldens, W. (1996), 'The lifting scheme: a custom-design construction of biorthogonal wavelets', *Appl. Comput. Harmon. Anal.* **3**(2), 186–200.

Sweldens, W. (1998), 'The lifting scheme: a construction of second generation wavelets', *SIAM J. Math. Anal.* **29**(2), 511–546.

Taswell, C. (1996), 'Satisficing search algorithms for selecting near-best bases in adaptive tree-structured wavelet transforms', *IEEE Trans. Signal Process.* **44**(10), 2423–2438.

Tenllado, C. & Setoain, J. (2008), 'Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting', *IEEE Trans. Parallel Distrib. Syst.* **19**(3), 299–310.

Torrence, C. & Compo, G. (1998), 'A practical guide to wavelet analysis', *Bull. Am. Meteorol. Soc.* .

Valette, S. & Prost, R. (2004), 'Wavelet-based multiresolution analysis of irregular surface meshes', *IEEE Trans. Vis. Comput. Graph.* **10**(2), 113–22.

Van Der Laan, W., Jalba, A. & Roerdink, J. (2011), 'Accelerating wavelet lifting on graphics hardware using CUDA', *IEEE Trans. Parallel Distrib. Syst.* **22**(1), 132–146.

Vanraes, E., Jansen, M. & Bultheel, A. (2002), 'Stabilised wavelet transforms for non-equispaced data smoothing', *Signal Processing* **82**(12), 1979–1990.

Vidakovic, B. (1998), 'Nonlinear wavelet shrinkage with Bayes rules and Bayes factors', *J. Am. Stat. Assoc.* **93**(441), 173–179.

Vidakovic, B. (1999), *Statistical Modeling by Wavelets*, 1st edn, Wiley, New York.

Waton, J. & McCoy, E. (2016), Lifting the LA8 transform, *in* '5th Comput. Math. Comput. Geom. Stat. (CMCGS 2016)', pp. 127–133.

Weyrich, N. & Warhola, G. (1998), 'Wavelet shrinkage and generalized cross validation for image denoising', *IEEE Trans. Image Process.* **7**(1), 82–90.

Whitcher, B. (2013), *waveslim: Basic wavelet routines for one-, two- and three-dimensional signal processing*. R package version 1.7.3.
URL: *http://cran.r-project.org/package=waveslim*

Wickham, H. (2016), *ggplot2: Elegant Graphics for Data Analysis*, Springer-Verlag New York.

Wickham, H., Danenberg, P. & Eugster, M. (2018), *roxygen2: In-Line Documentation for R.* R package version 6.1.0.
URL: *https://CRAN.R-project.org/package=roxygen2*

Wickham, H., Hester, J. & Chang, W. (2018), *devtools: Tools to Make Developing R Packages Easier.* R package version 1.13.6.
URL: *https://CRAN.R-project.org/package=devtools*

Zhang, F. (2011), *Matrix Theory: Basic Results and Techniques*, 2 edn, Springer, New York.

Zhu, L., Yadong, Z., Zhang, D., Wang, D., Wang, H. & Wang, X. (2015), 'Parallel multi-level 2D-DWT on CUDA GPUs and its application in ring artefact removal', *Concurr. Comput. Pract. Exp.* **27**, 5188–5205.

Appendix A

Miscellaneous Background

In this chapter, we introduce a few diverse topics from statistics and mathematics for applications in subsequent chapters. We will use the penalised linear regression in cross validation in Chapter 6 to derive wavelet coefficients directly from the wavelet matrix even when removing coefficients for validation and losing our dyadic structure; the golden section search will be our method here for minimising the cross validation criterion. We will use spline and kernel smoothing in a simulation study in Chapter 7 as comparisons with our new wavelet methods.

Much of the material from this chapter is adapted from Hastie et al. (2009), which is an excellent guide to statistical learning.

A.1 Penalised Linear Regression

In this section, we will introduce penalised linear regression starting with standard linear regression (with no penalty) and progressing to lasso and ridge regression. For a more in-depth introduction, see Hastie et al. (2009) and the references therein, on which this section is based.

Let $\mathcal{Y}$ and $\mathcal{X}$ be random variables with $\mathcal{Y}$ our response or dependent variable and $\mathcal{X}$ the p-vector predictor or independent variables. We observe $\boldsymbol{y} = (y_1, \ldots, y_n)$ and $X = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^T$, an $n \times p$ matrix of predictors $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$. X is also known as a design matrix.

Remark A.1.1. *Note that we change our indexing in this chapter to be* $1, \ldots, n$ *rather than* $0, \ldots, n-1$ *which works well for wavelets and ties in with our C implementations. However, with parametric regression, the* 0 *subscript is often reserved for intercepts. So we adopt a change of notation to be consistent with the literature.*

We would like to predict $\boldsymbol{y}$ using a linear model

$$\hat{y}_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{ij}.$$

β_0 in this model is an intercept, which we may or may not be estimating. If we are estimating an intercept, then we would add a column of ones to our design matrix X. Then, in matrix form, we have,

$$\hat{\boldsymbol{y}} = X\boldsymbol{\beta},$$

with $\boldsymbol{\beta}$ a vector $(\beta_0, \ldots, \beta_p)$ or $(\beta_1, \ldots, \beta_p)$ depending on whether we have modelled an intercept. We would like to estimate $\boldsymbol{\beta}$ by some penalised least squares method.

The general case of penalised linear regression estimates $\boldsymbol{\beta}$ using, in Langrangian form,

$$\tilde{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{n} \left[y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right]^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q \right\}, \tag{A.1}$$

where $\lambda \geq 0$ is the parameter controlling shrinkage. Increasing λ will reduce the variance of our coefficients $\boldsymbol{\beta}$ whilst adding bias.

A.1.1 Standard Linear Regression

In the case of a standard linear regression, we have no penalty on the size of our coefficients β_j, so the estimate of $\boldsymbol{\beta}$ is simply the least squares fit:

$$\hat{\boldsymbol{\beta}}^{LS} = \arg\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{n} \left[y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right]^2 \right\}. \tag{A.2}$$

In matrix form, we are trying to minimise the RSS (residual sum of squares):

$$\mathrm{RSS}(\boldsymbol{\beta}) = (\boldsymbol{y} - X\boldsymbol{\beta})^T(\boldsymbol{y} - X\boldsymbol{\beta}).$$

This has a simple closed solution

$$\hat{\boldsymbol{\beta}}^{LS} = (X^T X)^{-1} X^T \boldsymbol{y}.$$

We can also define the *hat matrix*, which projects $\boldsymbol{y}$ to its fitted values $\hat{\boldsymbol{y}}$:

$$\hat{\boldsymbol{y}} = X(X^T X)^{-1} X^T \boldsymbol{y} = H\boldsymbol{y}.$$

The degrees of freedom can be obtained from the trace of this hat matrix:

$$DF = \mathrm{tr}(H),$$

which is p for standard least squares regression, the number of estimated parameters.

A.1.2 Lasso

The Lasso (least absolute shrinkage and selection operator) is a special case of (A.1) with $q = 1$:

$$\hat{\boldsymbol{\beta}}^{lasso} = \arg\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{n} \left[y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right]^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \tag{A.3}$$

The solution to this is calculated by quadratic programming. In the special case of an orthonormal design matrix X, lasso has an explicit solution of

$$\mathrm{sign}(\hat{\beta}_j)(|\hat{\beta}_j| - \lambda)_+,$$

where the notation $x_+ = \min(0, x)$. This shrinkage by a constant shift of λ is called *soft thresholding*, which we introduce in the context of wavelets in Section 2.8. This is in fact the same solution as wavelet soft thresholding and can be considered as a penalised linear regression with an $\mathcal{L}_1$ norm and an orthonormal design matrix (Antoniadis & Fan 2001).

A.1.3 Ridge Regression

Ridge regression is the special case of (A.1) with $q = 2$:

$$\hat{\boldsymbol{\beta}}^{ridge} = \arg\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{n} \left[y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right]^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}. \tag{A.4}$$

Ridge regression is usually run with centred inputs, removing the need for the intercept term β_0. In matrix form, we are trying to minimise the penalised RSS

$$\text{RSS}(\lambda) = (\boldsymbol{y} - X\boldsymbol{\beta})^T(\boldsymbol{y} - X\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}.$$

The ridge regression solution is

$$\hat{\boldsymbol{\beta}}^{ridge} = (X^TX + \lambda I)^{-1}X^T\boldsymbol{y}. \tag{A.5}$$

In a situation with an orthonormal design matrix, this simplifies to

$$\hat{\boldsymbol{\beta}}^{ridge} = \frac{1}{1+\lambda}\hat{\boldsymbol{\beta}}^{LS},$$

a modification of the LS solution but with each component shrunk proportionally towards 0.

The effective degrees of freedom in ridge regression is given by the trace of its hat matrix:

$$\begin{aligned} EDF(\lambda) &= \text{tr}(X(X^TX + \lambda I)^{-1}X^T) \\ &= \text{tr}(H_\lambda). \end{aligned}$$

A.2 Nonparametric Regression

In this section of nonparametric regression, we consider the setup

$$y_i = f(x_i) + \xi_i, \quad \xi_i \sim N(0, \sigma^2), \quad i = 0, \ldots, n-1. \tag{A.6}$$

This is similar to our notation for wavelets, except in the wavelet literature it is common to substitute t_i for x_i as wavelets are frequently used in time series applications.

A.2.1 Spline Smoothing

This section is also predominantly based on material from Hastie et al. (2009). See for example Press, Teukolsky, Vetterling & Flannery (2007) or Lange (2010) for a more numerical introduction to cubic splines. Splines are piecewise polynomials used for interpolation and smoothing.

Let us consider the linear basis expansion of a function f, defined for $x \in \mathbb{R}$,

$$f(x) = \sum_{m=0}^{M-1} \beta_m h_m(x),$$

with basis functions $h_m(x)$. The *cubic spline* fits piecewise cubic polynomials with continuous first and second derivatives at its knots κ_0 and κ_1 (the connection points of the piecewise polynomial functions). The cubic spline has the following basis functions:

$$\begin{aligned} &h_0(x) = 1, \quad &&h_1(x) = x, \quad &&h_2(x) = x^2, \\ &h_3(x) = x^3, \quad &&h_4(x) = (x-\kappa_0)_+^3, \quad &&h_5(x) = (x-\kappa_1)_+^3. \end{aligned}$$

Nonlinear piecewise polynomials suffer from increased bias at their boundaries, so the *natural cubic spline* is designed to be linear beyond its boundary knots. It uses K basis functions to represent the spline with K knots:

$$N_0(x) = 1, \quad N_1(x) = x, \quad N_{k+2}(x) = d_k(x) - d_{K-2}(x),$$

with

$$d_k = \frac{(x-\kappa_k)_+^3 - (x-\kappa_{K-1})_+^3}{\kappa_{K-1} - \kappa_k}.$$

Smoothing splines are used to minimise the following expression:

$$\mathrm{RSS}(f, \lambda) = \sum_{i=0}^{n-1} \left[y_i - f(x_i)\right]^2 + \lambda \int \left[f''(t)\right]^2 dt. \tag{A.7}$$

In fact, this has a unique, finite-dimensional minimiser of the natural cubic spline:

$$f(x) = \sum_{j=0}^{n-1} N_j(x)\theta_j.$$

If we define $(N)_{ij} = N_j(x_i)$ and $(\Omega_N)_{jk} = \int N_j''(t) N_k''(t) dt$, then we can rewrite (A.7) for a natural spline basis in matrix form:

$$\mathrm{RSS}(\boldsymbol{\theta}, \lambda) = (\boldsymbol{y} - N\boldsymbol{\theta})^T (\boldsymbol{y} - N\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}^T \Omega_N \boldsymbol{\theta},$$

which has the solution

$$\hat{\boldsymbol{\theta}} = (N^T N + \lambda \Omega_N)^{-1} N^T \boldsymbol{y}.$$

This is a generalisation of the ridge regression problem outlined in Section A.1.3.

Spline smoothing is sensitive to the choice of the smoothing parameter λ. In (A.7) we note that $\lambda = 0$ would allow for any function f that interpolates the data $\boldsymbol{y}$, whilst $\lambda = \infty$ does not tolerate any second derivative. The *hat matrix* for natural splines, coinciding with that in regression, H_λ is defined by

$$\begin{aligned} \hat{\boldsymbol{y}} &= N(N^T N + \lambda \Omega_N)^{-1} N^T \boldsymbol{y} \\ &= H_\lambda \boldsymbol{y}. \end{aligned}$$

S_λ depends only on the x_i and λ. The effective degrees of freedom is given by the trace the hat matrix:

$$EDF(\lambda) = \mathrm{tr}(H_\lambda).$$

Selection of the parameter λ is a bias/variance trade-off. We can choose λ for example by fixing the degrees of freedom or by cross validation (see Chapter 6 for an introduction to cross validation).

A.2.2 Kernel Smoothing

For a more in depth introduction to kernel smoothing, see for example Hastie et al. (2009), on which this section is based. Kernel smoothing estimates a function $f(x)$ by fitting local models at each observation point $x = z$. Points around z are weighted according to a kernel $K_\lambda(z, x_i)$, tuned by the parameter λ which controls the size of the local region considered. A kernel can have the form

$$K_\lambda(x, z) = D\left(\frac{|x - z|}{h_\lambda(z)}\right),$$

with D some positive real valued function that is non-increasing for points at increasing distances from z and kernel width $h_\lambda(z)$. Often, a kernel width of $h_\lambda(z) = \lambda$ is used to keep the bias constant. Two common kernels with compact support are the *Epanechnikov quadratic kernel* and the *tri-cube kernel* which have the following D functions:

$$D(t) = \begin{cases} 3/4(1 - t^2) & \text{if } |t| \leq 1, \\ 0 & \text{otherwise,} \end{cases} \quad \text{for the Epanechnikov quadratic kernel,}$$

$$D(t) = \begin{cases} (1 - |t|^3)^3 & \text{if } |t| \leq 1, \\ 0 & \text{otherwise,} \end{cases} \quad \text{for the tri-cube kernel.}$$

Another choice of D without compact support might be the Gaussian density function

$$D(t) = \phi(t).$$

We could fit a simple averaging model such as the *Nadaraya-Watson kernel-weighted average*:

$$\hat{f}(z) = \frac{\sum_{i=0}^{n-1} K_\lambda(z, x_i) y_i}{\sum_{i=0}^{n-1} K_\lambda(z, x_i)}.$$

Alternatively, a local linear regression limits the bias at the boundaries to first order. This estimates a separate weighted LS problem in neighbourhoods about each point z, and is only evaluated at the points z. For this, we solve

$$\min_{\alpha(z), \beta(z)} \left\{ \sum_{i=0}^{n-1} K_\lambda(z, x_i) \left[y_i - \alpha(z) - \beta(z) x_i \right]^2 \right\}.$$

The kernel width parameter λ is another bias/variance trade-off: smaller values of λ will give larger variance and lower bias whilst larger values will have the opposite effect. If $h_\lambda(z) = \lambda$ is chosen, then λ will correspond to the radius of the support region in the case of the Epanechnikov

quadratic and tri-cube kernels; for the Gaussian kernel λ will represent the standard deviation. We might again pick λ using cross validation (see Chapter 6) or by specifying effective degrees of freedom for linear estimators, for example.

A.3 Golden Section Search

The golden section search is an algorithm for finding the minimum $f(m)$ of a function defined on the real line which is appropriate for unimodal functions (i.e. functions f that monotonically increase either side of their minimum). This search method is guaranteed to find the minimum of a unimodal function provided we have a bracket for this minimum. We use the golden ratio to formulate our narrowing bracketing interval choice which improves the speed of convergence over a naive search method (Bertsekas 1999). We base this section upon material from Press et al. (2007) and Lange (2010).

We start with the points $a < b < c$ such that (a, c) brackets the minimum (i.e. $m \in (a, c)$) and $b \in (a, c)$ too with $f(b) < f(c)$ and $f(b) < f(a)$. We would like to progressively reduce this bracketing interval until we find the minimiser m. We then choose another value d at which to next evaluate our function. It is natural to choose this in the larger of the intervals (a, b) and (b, c). Let us assume this is (a, b). If we find $f(d) < f(b)$ then we get a new bracketing interval of $(a_1, c_1) = (a, b)$. On the other hand, if $f(d) > f(b)$ then the new bracketing interval would be $(a_1, c_1) = (b, c)$. We continue in this manner narrowing the interval (a_i, c_i) until we reach the stopping criterion of

$$|c_k - a_k| > \tau(|b_k| + |d_k|),$$

for tolerance τ given by the square root of the machine accuracy (Press et al. 2007). This comes from the Taylor expansion of the function at its minimum.

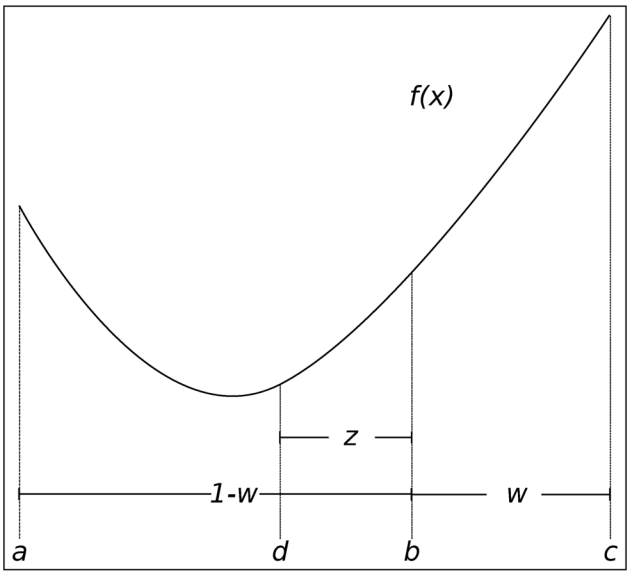


Figure A.1: An illustration of the method of golden section search. Points a and c form our initial bracket of the minimum of $f(x)$, and b and d are midpoints where we initially evaluate the function. Values w, $1 - w$ and z represent fractions of the width of this bracket (a, c).

The golden section search gets its name from the method of choosing the points b_i and d_i. Let

w be the fraction of the distance between a and c where b lies, i.e.,

$$w = \frac{c-b}{c-a} \quad \text{and} \quad 1-w = \frac{b-a}{c-a}.$$

Then suppose $1-w$ is larger, so we select d in (a,b). Then let

$$z = \frac{b-d}{c-a}.$$

See Figure A.1 for an illustration of this. Now, depending on whether $f(d) < f(b)$ or $f(d) > f(b)$, this leaves a fraction of either $1-w$ or $w+z$ respectively of the original (a,c) bracket remaining to be explored. Setting these equal gives $z = 1-2w$ and ensures we mitigate against the worst case scenario. Then, because we apply this search process iteratively, the location of d should have been chosen in the same manner. So, by scale similarity,

$$\frac{z}{1-w} = w.$$

This gives us the quadratic $w^2 - 3w + 1$ which has the solution

$$w = \frac{3-\sqrt{5}}{2}.$$

Thus, we are picking values $w = 1 - \frac{1}{\phi}$ and $1-w = \frac{1}{\phi}$ from either side of our bracket, pertaining to the golden ratio

$$\phi = \frac{1+\sqrt{5}}{2}.$$

Appendix B

Exact Lifted LA8 Coefficients

We provide in Figure B.1 the Euclidean Algorithm for computing the exact LA8 lifting coefficients in terms of the coefficients $c_i, i = 0, \ldots, 7$. The expansions for q_{41}, q_{42} and K each fill more than an A4 page, so instead we prefer to provide this algorithm to produce them.

We calculated our lifting step as

$$s(z) = \gamma_2 z^2 + \gamma_1 z + \gamma_0.$$

We now give expansions for the γ_i.

$$\begin{aligned}
\gamma_0 =&(-c_0c_4c_7^3 + c_0c_5c_6c_7^2 + c_1c_4c_6c_7^2 - c_1c_5c_6^2c_7 + c_2^2c_7^3 - 2c_2c_3c_6c_7^2 - c_2c_4c_5c_7^2 + c_2c_5^2c_6c_7 + c_3^2c_6^2c_7 \\
&+ c_3c_4^2c_7^2 + -c_3c_4c_5c_6c_7)/\big((c_0^3c_7^3 - 3c_0^2c_1c_6c_7^2 - c_0^2c_2c_5c_7^2 - 2c_0^2c_3c_4c_7^2 + 3c_0^2c_3c_5c_6c_7 + c_0^2c_4c_5^2c_7 \\
&- c_0^2c_5^3c_6 + 3c_0c_1^2c_6^2c_7 + 3c_0c_1c_2c_4c_7^2 - c_0c_1c_2c_5c_6c_7 + c_0c_1c_3c_4c_6c_7 - 3c_0c_1c_3c_5c_6^2 \\
&- 2c_0c_1c_4^2c_5c_7 + 2c_0c_1c_4c_5^2c_6 + c_0c_2^2c_3c_7^2 - 2c_0c_2c_3^2c_6c_7 - c_0c_2c_3c_4c_5c_7 + c_0c_2c_3c_5^2c_6 + c_0c_3^3c_6^2 \\
&+ c_0c_3^2c_4^2c_7 - c_0c_3^2c_4c_5c_6 - c_1^3c_6^3 - 3c_1^2c_2c_4c_6c_7 + 2c_1^2c_2c_5c_6^2 + c_1^2c_3c_4c_6^2 + c_1^2c_4^3c_7 - c_1^2c_4^2c_5c_6 \\
&- c_1c_2^3c_7^2 + 2c_1c_2^2c_3c_6c_7 + c_1c_2^2c_4c_5c_7 - c_1c_2^2c_5^2c_6 - c_1c_2c_3^2c_6^2 - c_1c_2c_3c_4^2c_7 + c_1c_2c_3c_4c_5c_6)c_7\big)
\end{aligned}$$

$$\begin{aligned}
\gamma_1 =&\big(- c_0c_2c_7^3 + c_0c_3c_6c_7^2 + c_1c_2c_6c_7^2 - c_1c_3c_6^2c_7 - c_1c_4^2c_7^2 + 2c_1c_4c_5c_6c_7 - c_1c_5^2c_6^2 + c_2^2c_5c_7^2 \\
&- 2c_2c_3c_5c_6c_7 - c_2c_4c_5^2c_7 + c_2c_5^3c_6 + c_3^2c_5c_6^2 + c_3c_4^2c_5c_7 - c_3c_4c_5^2c_6 + (c_0c_4c_7^2 - c_0c_5c_6c_7 \\
&- c_1c_4c_6c_7 + c_1c_5c_6^2 - c_2^2c_7^2 + 2c_2c_3c_6c_7 + c_2c_4c_5c_7 - c_2c_5^2c_6 - c_3^2c_6^2 - c_3c_4^2c_7 + c_3c_4c_5c_6)c_5\big)/ \\
&\big((c_0^3c_7^3 - 3c_0^2c_1c_6c_7^2 - c_0^2c_2c_5c_7^2 - 2c_0^2c_3c_4c_7^2 + 3c_0^2c_3c_5c_6c_7 + c_0^2c_4c_5^2c_7 - c_0^2c_5^3c_6 + 3c_0c_1^2c_6^2c_7 \\
&+ 3c_0c_1c_2c_4c_7^2 - c_0c_1c_2c_5c_6c_7 + c_0c_1c_3c_4c_6c_7 - 3c_0c_1c_3c_5c_6^2 - 2c_0c_1c_4^2c_5c_7 + 2c_0c_1c_4c_5^2c_6 \\
&+ c_0c_2^2c_3c_7^2 - 2c_0c_2c_3^2c_6c_7 - c_0c_2c_3c_4c_5c_7 + c_0c_2c_3c_5^2c_6 + c_0c_3^3c_6^2 + c_0c_3^2c_4^2c_7 - c_0c_3^2c_4c_5c_6 \\
&- c_1^3c_6^3 - 3c_1^2c_2c_4c_6c_7 + 2c_1^2c_2c_5c_6^2 + c_1^2c_3c_4c_6^2 + c_1^2c_4^3c_7 - c_1^2c_4^2c_5c_6 - c_1c_2^3c_7^2 + 2c_1c_2^2c_3c_6c_7 \\
&+ c_1c_2^2c_4c_5c_7 - c_1c_2^2c_5^2c_6 - c_1c_2c_3^2c_6^2 - c_1c_2c_3c_4^2c_7 + c_1c_2c_3c_4c_5c_6)c_7\big)
\end{aligned}$$

$$
\begin{aligned}
a_0 &:= g_e = c_1 + c_3 z^{-1} + c_5 z^{-2} + c_7 z^{-3} \\
b_0 &:= g_o = c_0 z + c_2 + c_4 z^{-1} + c_6 z^{-2} \\
q_1 &:= \frac{c_7}{c_6} z^{-1} + \frac{\frac{c_5 - c_7 c_4}{c_6}}{c_6} \\
r_1 &:= -b_0 q_1 + a_0 \\
a_1 &:= b_0 = a_{11} z + a_{12} + a_{13} z^{-1} + a_{14} z^{-2} \\
b_1 &:= r_1 = b_{11} z + b_{12} + b_{13} z^{-1} \\
q_2 &:= \frac{a_{13} - \frac{a_{14} b_{12}}{b_{13}}}{b_{13}} + \frac{a_{14}}{b_{13}} z^{-1} = q_{21} + q_{22} z^{-1} \\
r_2 &:= -b_1 q_2 + a_1 \\
a_2 &:= b_1 = a_{21} z + a_{22} + a_{23} z^{-1} \\
b_2 &:= r_1 = b_{21} z + b_{22} \\
q_3 &:= \frac{a_{23}}{b_{22}} z^{-1} + \frac{a_{21}}{b_{21}} = q_{31} + q_{32} z^{-1} \\
r_3 &:= -b_2 q_3 + a_2 \\
a_3 &:= b_2 = a_{31} z + a_{32} \\
b_3 &:= r_3 = b_{31} \\
q_4 &:= \frac{a_{31}}{b_{31}} z + \frac{a_{32}}{b_{31}} = q_{41} z + q_{42} \\
r_4 &:= -b_3 q_4 + a_3 = 0 \\
a_4 &:= b_3 = a_{41} =: K \\
b_4 &:= r_4 = 0
\end{aligned}
$$

Figure B.1: Euclidean algorithm for dividing the odd and even parts of the scaling filter for the LA8 wavelet. This algorithm can be used to calculate the exact forms of the lifting coefficients in terms of the original wavelet coefficients.

$$
\begin{aligned}
\gamma_2 =&(c_1c_3c_4c_6c_7 - c_2c_3c_4c_5c_7 + c_0^2c_1c_2c_5c_6c_7 - c_0^2c_1c_3c_4c_6c_7 + c_0^2c_2c_3c_4c_5c_7 + 3c_0c_1^2c_2c_4c_6c_7 \\
&- 2c_0c_1c_2^2c_3c_6c_7 - c_0c_1c_2^2c_4c_5c_7 + c_0c_1c_2c_3c_4^2c_7 - 3c_0^3c_3c_5c_6c_7 - 3c_0^2c_1c_2c_4c_7^2 + 3c_0^2c_1c_3c_5c_6^2 \\
&+ 2c_0^2c_1c_4^2c_5c_7 - 2c_0^2c_1c_4c_5^2c_6 + 2c_0^2c_2c_3^2c_6c_7 - c_0^2c_2c_3c_5^2c_6 + c_0^2c_3^2c_4c_5c_6 - 2c_0c_1^2c_2c_5c_6^2 \\
&- c_0c_1^2c_3c_4c_6^2 + c_0c_1^2c_4^2c_5c_6 + c_0c_1c_2^2c_5^2c_6 + c_0c_1c_2c_3^2c_6^2 + 3c_0c_3c_5c_6c_7 + c_1c_2c_4c_7^2 - 2c_1c_3c_5c_6^2 \\
&- c_1c_4^2c_5c_7 + c_1c_4c_5^2c_6 - 2c_2c_3^2c_6c_7 + c_2c_3c_5^2c_6 - c_3^2c_4c_5c_6 - c_0^4c_7^3 + c_0^2c_7^3 + c_3^2c_4^2c_7 + 3c_0^3c_1c_6c_7^2 \\
&+ c_3^3c_6^2 + (c_0c_4c_7^2 - c_0c_5c_6c_7 - c_1c_4c_6c_7 + c_1c_5c_6^2 - c_2^2c_7^2 + 2c_2c_3c_6c_7 + c_2c_4c_5c_7 - c_2c_5^2c_6 \\
&- c_3^2c_6^2 - c_3c_4^2c_7 + c_3c_4c_5c_6)c_3 + c_0^3c_5^3c_6 - c_0^2c_3^3c_6^2 + c_0c_1^3c_6^3 - c_0c_5^3c_6 + (c_0c_2c_7^2 - c_0c_3c_6c_7 \\
&- c_0c_4c_5c_7 + c_0c_5^2c_6 - c_1c_2c_6c_7 + c_1c_3c_6^2 + c_1c_4^2c_7 - c_1c_4c_5c_6)c_5 - c_0c_1c_2c_3c_4c_5c_6 + \\
&+ c_1^2c_6^2c_7 + c_2^2c_3c_7^2 + c_0^3c_2c_5c_7^2 + 2c_0^3c_3c_4c_7^2 - c_0^3c_4c_5^2c_7 - 3c_0^2c_1^2c_6^2c_7 - c_0^2c_2^2c_3c_7^2 - c_0^2c_3^2c_4^2c_7 \\
&- c_0c_1^2c_4^3c_7 + c_0c_1c_2^3c_7^2 - 2c_0c_1c_6c_7^2 - c_0c_2c_5c_7^2 - 2c_0c_3c_4c_7^2 + c_0c_4c_5^2c_7)/\big((c_0^3c_7^3 - 3c_0^2c_1c_6c_7^2 \\
&- c_0^2c_2c_5c_7^2 - 2c_0^2c_3c_4c_7^2 + 3c_0^2c_3c_5c_6c_7 + c_0^2c_4c_5^2c_7 - c_0^2c_5^3c_6 + 3c_0c_1^2c_6^2c_7 + 3c_0c_1c_2c_4c_7^2 \\
&- c_0c_1c_2c_5c_6c_7 + c_0c_1c_3c_4c_6c_7 - 3c_0c_1c_3c_5c_6^2 - 2c_0c_1c_4^2c_5c_7 + 2c_0c_1c_4c_5^2c_6 + c_0c_2^2c_3c_7^2 \\
&- 2c_0c_2c_3^2c_6c_7 - c_0c_2c_3c_4c_5c_7 + c_0c_2c_3c_5^2c_6 + c_0c_3^3c_6^2 + c_0c_3^2c_4^2c_7 - c_0c_3^2c_4c_5c_6 - c_1^3c_6^3 \\
&- 3c_1^2c_2c_4c_6c_7 + 2c_1^2c_2c_5c_6^2 + c_1^2c_3c_4c_6^2 + c_1^2c_4^3c_7 - c_1^2c_4^2c_5c_6 - c_1c_2^3c_7^2 + 2c_1c_2^2c_3c_6c_7 \\
&+ c_1c_2^2c_4c_5c_7 - c_1c_2^2c_5^2c_6 - c_1c_2c_3^2c_6^2 - c_1c_2c_3c_4^2c_7 + c_1c_2c_3c_4c_5c_6)c_7\big)
\end{aligned}
$$

Appendix C

WaveCUDA

Listing C.1: Serial implementation of forward Haar DWT using moving window

```
int fHaar(real* x, uint len, uint skip, uint nlevels){
  real tmp;
  uint i;
  if(skip < (1 << nlevels)){
    // we have something to do
    for (i=0;i<len;i+=2*skip){
      tmp = (x[i] - x[i+skip])*invR2;
      x[i] = (x[i] + x[i+skip])*invR2;
      // x[i] now holds the latest scaling coeff
      x[i+skip] = tmp;
      // x[i+skip] now holds the latest detail coeff
    }
    return(fHaar(x,len,skip<<1,nlevels));
  }
  return(0);
}
```

Listing C.2: DWT memory layout

```
/*
  x is our vector

  x                        transform
  i
  0   | 0 |   |s20|   |s10|   |s00|
  1   | 0 |   |d20|   |d20|   |d20|
  2   | 5 |   |s21|   |d10|   |d10|
  3   | 4 |   |d21|   |d21|   |d00|
  4   | 8 |   |s22|   |s11|   |d11|
  5   | 6 |   |d22|   |d22|   |d22|
  6   | 7 |   |s23|   |d11|   |d11|
  7   | 3 |   |d23|   |d23|   |d23|

      skip=1  skip=2  skip=4  skip=8

  ----> forward transform ---->
  <--- backward transform <----

  where sjk (djk) is the scaling (detail) coefficient number k at level j.
*/
```

Listing C.3: Serial implementation of Forward Daub4 DWT using moving window

```
int fDaub4(real* x, uint len, uint skip, uint nlevels){
  real tmp, x0, xskip;
  uint i;
  if(skip < (1 << nlevels)){
    //store first 2 entries of vector
    x0=x[0]; xskip=x[skip];
    //do loop from i=0 -> i < len-3*skip : does transform to store in i=0 -> i=len-2*skip
    for(i=0;i<(len - 3*skip);i+=2*skip){
      tmp = x[i]*C3 - x[i+skip]*C2 + x[i+2*skip]*C1 - x[i+3*skip]*C0; //detail
      x[i] = x[i]*C0 + x[i+skip]*C1 + x[i+2*skip]*C2 + x[i+3*skip]*C3; //scaling
      x[i+skip] = tmp;
    }
    //do last cycle of loop manually
    tmp =  x[len-2*skip]*C3 - x[len-skip]*C2 + x0*C1 - xskip*C0;
    x[len-2*skip] = x[len-2*skip]*C0 + x[len-skip]*C1 + x0*C2 + xskip*C3;
    x[len-skip] = tmp;
    return(fDaub4(x,len,skip<<1,nlevels));
  }
  return(0);
}
```

Listing C.4: Serial implementation of Forward Daub4 DWT using lifting scheme

```
int flDaub4(real* x, uint len, uint skip, uint nlevels){
  real xfirst;
  uint i;
  // with periodic boundary conditions
  if(skip < (1 << nlevels)){

    for(i=0;i<len;i+=2*skip){
      x[i] = x[i] + x[i+skip]*C10;
      // s1[l] = x[2l] + sqrt(3)*x[2l+1]
    }

    //do first cycle manually
    x[skip] = x[skip] - x[0]*C11 - x[len-2*skip]*C12;
    for(i=2*skip;i<len;i+=2*skip){
      x[i+skip] = x[i+skip] - x[i]*C11 - x[i-2*skip]*C12;
      // d1[l] = x[2l+1] - s1[l]*sqrt(3)/4 - s1[l-1]*(sqrt(3)-2)/4
    }

    for(i=0;i<(len-2*skip);i+=2*skip){
      x[i] = x[i] - x[i+3*skip];
      // s2[l] = s1[l] - d1[l+1]
    }
    x[len-2*skip] = x[len-2*skip] - x[skip];
    //do last cycle manually

    for(i=0;i<len;i+=2*skip){
      x[i] = x[i]*C13;
      // s[l] = s2[l]*(sqrt(3)-1)/sqrt(2)
      x[i+skip] = x[i+skip]*C14;
      // d[l] = d1[l]*(sqrt(3)+1)/sqrt(2)
    }

    return(flDaub4(x,len,skip<<1,nlevels));
  }
  return(0);
}
```

Listing C.5: Packet-ordered Haar MODWT memory layout

```
/*
  data structure used is the following:
  x is our vector

  eg,

  x    = [x0    x1    x2    x3    x4    x5    x6    x7    ]

  which is transformed to vector xdat

  xdat = c( x1a  , x1b  , x2aa , x2ab , x2ba , x2bb ,
            x3aaa, x3aab, x3aba, x3abb, x3baa, x3bab, x3bba, x3bbb )

  where c( , ) is R concatenation notation and

  x1a  = [s20   d20   s21   d21   s22   d22   s23   d23  ]
  is the first packet (with no shift) for level 2, i.e. s20 = (1/sqrt(2)) * (x[1] + x[0]), d20
      = (1/sqrt(2)) * (x[1] - x[0])

  x1b  = [s20^1 d20^1 s21^1 d21^1 s22^1 d22^1 s23^1 d23^1]
  is the second packet for level 2, i.e. s20^1 = (1/sqrt(2)) * (x[2] + x[1])

  x2aa = [s10   d10   s11   d11   ]
  is first packet of level 1, i.e. s10 = (1/sqrt(2)) * (x1a[2] + x1a[0]) = (1/sqrt(2)) * (s20 +
       s21)

  x2ab = [s10^1 d10^1 s11^1 d11^1 ]
  is the second packet of level 1, s10^1 = (1/sqrt(2)) * (x1a[4] + x1a[2]) = (1/sqrt(2)) * (s21
       + s22)

  x2ba = [s10^2 d10^2 s11^2 d11^2 ]
  is the third packet of level 1, s10^2 = (1/sqrt(2)) * (x1b[2] + x1b[0]) = (1/sqrt(2)) * (s20
      ^1 + s21^1)

  x2bb = [s10^3 d10^3 s11^3 d11^3 ]
  is the fourth packet of levels 1, s10^3 = (1/sqrt(2)) * (x1b[4] + x1b[2]) = (1/sqrt(2)) * (
      s21^1 + s22^1)

  x3aaa= [s00 d00 ]
  is the first packet of level 3, s00 = (1/sqrt(2)) * (x2aa[2] + x2aa[0]) = (1/sqrt(2)) * (s10
      + s11)

  and x3aab, x3aba, x3abb, x3baa, x3bab, x3bba, x3bbb defined similarly

  This version works by copying the "s" coefficieints to the next level of then transforming in
       place.

*/
```

Listing C.6: Time-ordered Haar MODWT memory layout

```
/*
  data structure used is the following:
  x is our vector

  eg,

  x    = [x0    x1    x2    x3    x4    x5    x6    x7    ]

  which is transformed to vector xdat

  xdat = c( s_{2} , d_{2} , s_{1} , d_{1} , s_{0} , d_{0} )

  where c( , ) is R concatenation notation and

  s_{j} = [sj0 sj1 sj2 sj3 sj4 sj5 sj6 sj7]
  d_{j} = [dj0 dj1 dj2 dj3 dj4 dj5 dj6 dj7]

  these sji and dji are time-ordered coefficients, i.e.
  s_{j}[i] = (1/sqrt(2)) * (s_{j+1}[i] + s_{j+1}[i + skip])
  d_{j}[i] = (1/sqrt(2)) * (s_{j+1}[i] - s_{j+1}[i + skip])

  This version works by reading the "s" coefficieints from the previous level of then writing
      the transform in the current level.

*/
```

Listing C.7: CUDA simple Haar kernel

```
__global__ void Haar_kernel_shared(real* x, const uint len, const uint skip){
  uint i = (blockDim.x * blockIdx.x + threadIdx.x)*skip<<1;
  // counter inside vector x
  uint ish = threadIdx.x<<1;
  // counter inside shared vector x_work

  __shared__ real x_work[2*BLOCK_SIZE];

  if (i < len){
    x_work[ish] = x[i];
    x_work[ish+1] = x[i + skip];
  }

  //__syncthreads();

  if (i < len){
    x[i] = (x_work[ish] + x_work[ish+1])*invR2;
    x[i+skip] = (x_work[ish] - x_work[ish+1])*invR2;
  }

}
```

Listing C.8: CUDA multilevel Haar kernel

```
__global__ void Haar_kernel_shared_ml_f(real* x, const uint len, const uint skip, const uint
    levels){
  uint i = (blockDim.x * blockIdx.x + threadIdx.x)*skip<<1;
  // counter inside vector x
  uint ish = threadIdx.x<<1;
  // counter inside shared vector x_work
  uint li; // counter representing the number of levels
  real tmp;
  __shared__ real x_work[BLOCK_SIZE<<1];
  uint skipwork = 1;

  // copy into shared memory
  if (i < len){
    x_work[ish] = x[i];
    x_work[ish+1] = x[i + skip];
  }

  __syncthreads();

  //then we loop over a few levels

  for(li = 0; li < levels; li++){

    if (ish < (BLOCK_SIZE<<1)){
      tmp = (x_work[ish] - x_work[ish+skipwork])*invR2;
      x_work[ish] = (x_work[ish] + x_work[ish+skipwork])*invR2;
      x_work[ish+skipwork] = tmp;
    }

    ish=ish<<1; skipwork=skipwork<<1;
    __syncthreads();
  }

  ish = threadIdx.x<<1;

  if (i < len){
    x[i] = x_work[ish];
    x[i + skip] = x_work[ish+1];
  }

}
```

Listing C.9: C6 kernel shared memory

```
/*
  Shared memory has following, structure:

   _ _ _ _  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  _ _ _ _
  |        |                                              |        |
  |_ _ _ _ |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |_ _ _ _ |

  |<---6-->|<---------(2*BLOCK_SECT_ML2)---------------->|<---6-->|
  |<---------------(  2 * BLOCK_SIZE_ML2  )---------------------->|

  indices...
  0, 1, 2, 3, ..., k, k1, ..., k5
  used in 1st level of transform (where k=2*BLOCK_SECT_ML2+6) k1 is k+1 etc

  then indices...
  3, 4, 5, ..., k, k1, ..., k3
  used in 2nd level of transform

  First & last coefficients contain shared memory boundary coefficients* for the transform
      levels.
  *shared memory boundary coefficients: for the first & last shared memory blocks, they hold
      periodic boundary coefficient points; for all other blocks, they hold the boundary
      coefficients of the previous/following memory block.

  */
```

Listing C.10: CUDA multilevel C6 kernel

```
__global__ void C6_kernel_shared_f_ml2(real* x, const uint len, const uint skip, real* bdrs,
    const uint lenb){
  int a = 8, b = 8;
  int i = (BLOCK_SECT_ML2 * blockIdx.x + threadIdx.x -a/2)*skip << 1;
  // i = -16*skip,..., -2*skip, 0, 2*skip, ... , len-2*skip, len, ... , len + 10*skip
  // for each block, we have, e.g. i = -12*skip, ..., -2*skip, 0, 2*skip, ..., 2*skip*
      BLOCK_SECT_ML2-2*skip, 2*BLOCK_SECT_ML2*skip, 2*skip*BLOCK_SECT_ML2+2*skip, ..., 2*skip*
      BLOCK_SECT_ML2+10*skip
  uint ish = (threadIdx.x)<<1;
  uint ishlast = (BLOCK_SIZE_ML2-1)<<1;
  uint li;
  __shared__ real x_work[BLOCK_SIZE_ML2<<1];
  uint skipwork = 1;

  write_wvt_shared_gen(x,bdrs,len,skip,i,ish,a,b,BLOCK_SECT2,x_work);
  __syncthreads();

  for(li = 0; li < 2; li++){

    //1st cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-2)&&(ish>=2)) ||
         ((li==1)&&(ish<=ishlast-4)&&(ish>=4)) ){
      lift_cyc_1(x_work,ish,skipwork,FWD);
    }
    __syncthreads();


    //2nd cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-4)&&(ish>=2)) ||
        ((li==1)&&(ish<=ishlast-8)&&(ish>=4)) ){
      lift_cyc_2(x_work,ish,skipwork,FWD);
    }
    __syncthreads();


    //3rd cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-4)&&(ish>=2)) ||
        ((li==1)&&(ish<=ishlast-8)&&(ish>=4)) ){
      lift_cyc_3(x_work,ish,skipwork,FWD);
    }
    __syncthreads();


    //4th cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-4)&&(ish>=4)) ||
        ((li==1)&&(ish<=ishlast-8)&&(ish>=8)) ){
      lift_cyc_4(x_work,ish,skipwork,FWD);
    }
    __syncthreads();


    //5th cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-4)&&(ish>=4)) ||
        ((li==1)&&(ish<=ishlast-8)&&(ish>=8)) ){
      lift_cyc_5(x_work,ish,skipwork,FWD);
    }
    __syncthreads();
```

```
    //6th cycle ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if( ((li==0)&&(ish<=ishlast-4)&&(ish>=4)) ||
        ((li==1)&&(ish<=ishlast-8)&&(ish>=8)) ){
      lift_cyc_6(x_work,ish,skipwork,FWD);
    }
    __syncthreads();


    if(li==0) ish=ish<<1; skipwork=skipwork<<1;

  }


  // Now transform level is done. We copy shared array x_work back into x

  ish = (threadIdx.x)<<1;

  if( (ish>=a) && (ish<2*BLOCK_SECT_ML2+b) && (i<len)){
    x[i] = x_work[ish];
    x[i + skip] = x_work[ish+1];
  }


}
```

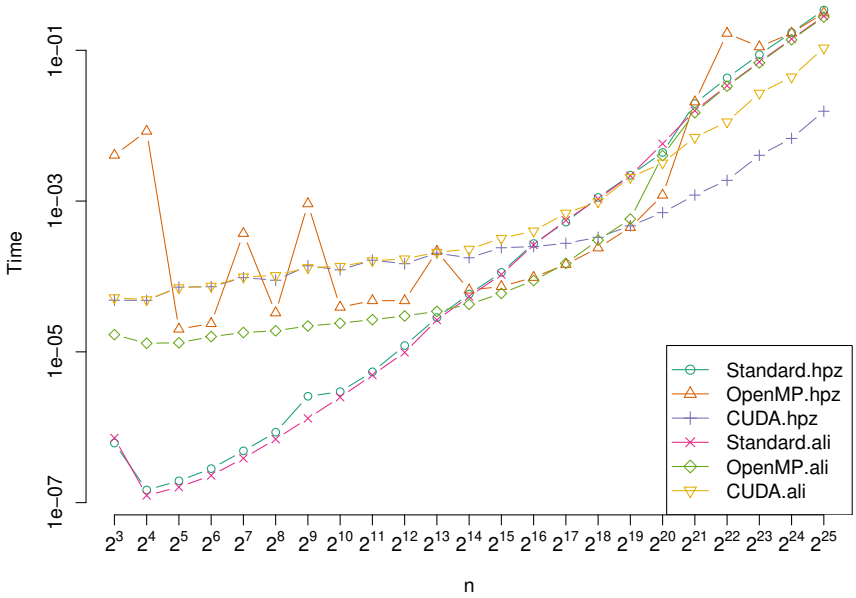


Figure C.1: Graph of timings for the Haar DWT on a log scale. We compare the Standard DWT CPU implementation, the lifted DWT on CPU and a CUDA implementation of the lifted DWT, on both a desktop (hpz) and laptop (ali). We plot the average time for each transform from 100 runs of the same signal.

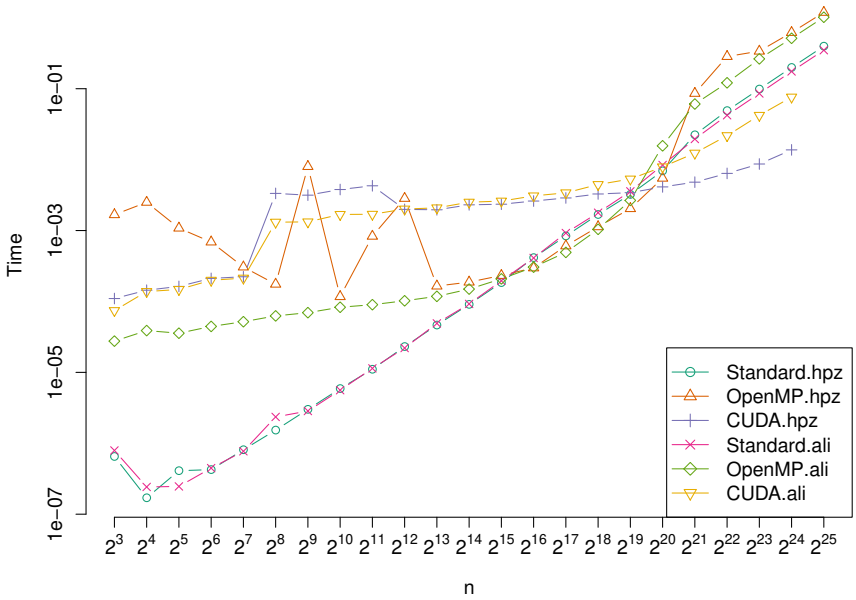


Figure C.2: Graph of timings for the D4 DWT on a log scale. We compare the Standard DWT CPU implementation, the lifted DWT on CPU and a CUDA implementation of the lifted DWT, on both a desktop (hpz) and laptop (ali).

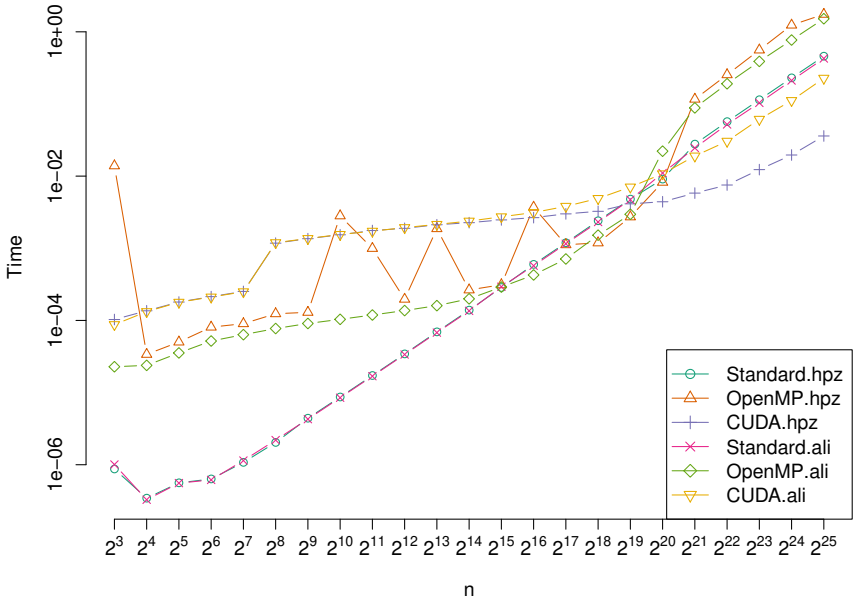


Figure C.3: Graph of timings for the C6 DWT on a log scale. We compare the Standard DWT CPU implementation, the lifted DWT on CPU and a CUDA implementation of the lifted DWT, on both a desktop (hpz) and laptop (ali).

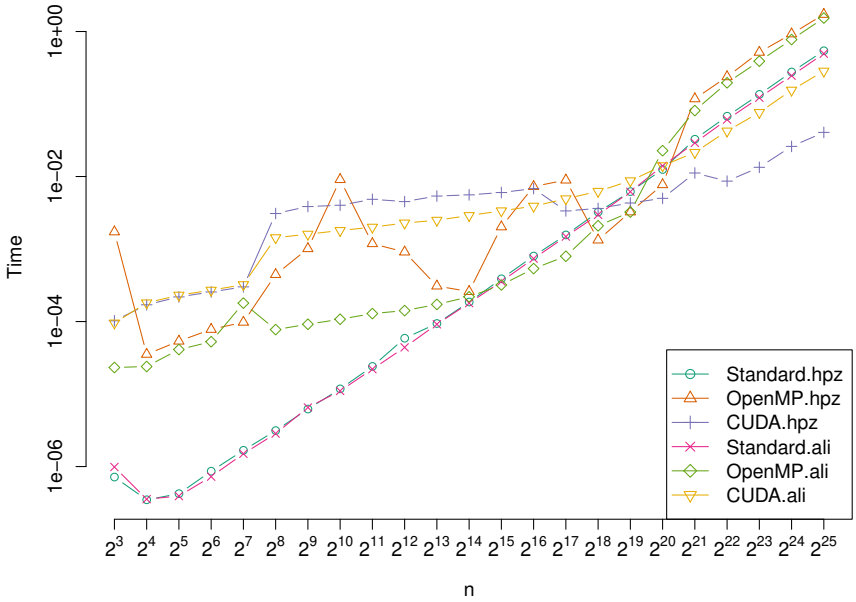


Figure C.4: Graph of timings for the LA8 DWT on a log scale. We compare the Standard DWT CPU implementation, the lifted DWT on CPU and a CUDA implementation of the lifted DWT, on both a desktop (hpz) and laptop (ali).

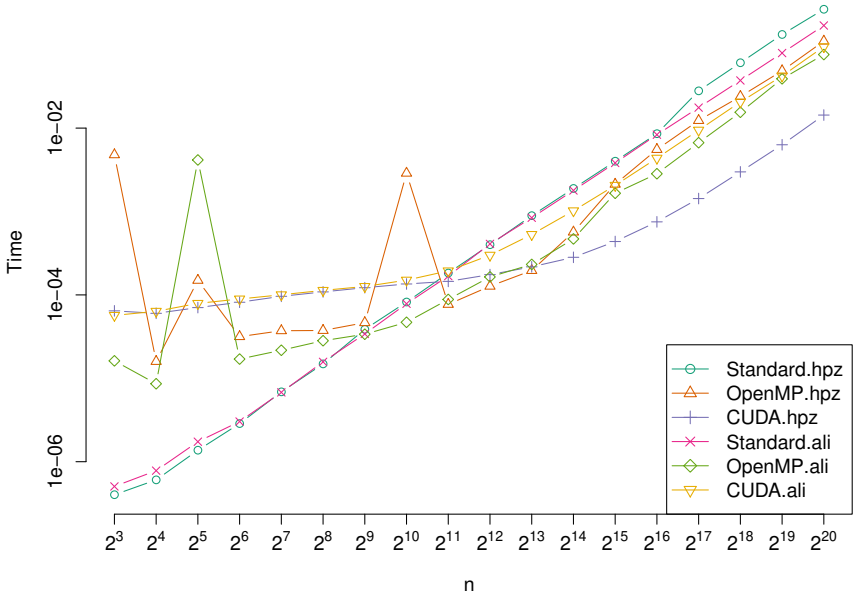


Figure C.5: Graph of timings for the Haar MODWT on a log scale. We compare serial CPU, parallel CPU and CUDA implementations of the time-ordered MODWT, on both a desktop (hpz) and laptop (ali). We plot the average time for each transform from 100 runs.

Appendix D

Simulation Studies Supplementary Tables

D.1 Wavelet Cross Validation

Signal	N2h	N2s	M2.1	M2.2	M3.2	M4.2	M5.2	M6.2	M7.2	M8.2	M512	M512.1	M512.2
Doppler	17	27	12	**32**	26	16	12	8	12	9	1	0	8
Heavisine	35	7	13	29	18	14	8	6	5	4	**40**	1	2
Blocks	16	3	22	**24**	19	16	16	22	18	18	0	13	21
Cubic	56	10	3	19	12	11	5	5	2	1	**75**	0	1

Table D.1: Results from a simulation study comparing different Cross Validation schemes on wavelet signals with SNR of 5. We tabulate the number of times out of 100 trials that each scheme *achieves the lowest MSE* out of all considered CV schemes. Note that we considered more schemes than we could include in the table and some schemes tied for first place, so the rows are unlikely to sum to 100. N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G.

Signal	N2h	N2s	M2.1	M2.2	M3.2	M4.2	M5.2	M6.2	M7.2	M8.2	M512	M512.1	M512.2
Doppler	54	**82**	58	68	67	59	58	53	53	51	10	7	47
Heavisine	**79**	78	32	58	51	43	34	30	29	25	73	3	22
Blocks	68	70	84	80	76	75	80	81	82	83	5	78	**85**
Cubic	87	80	21	51	38	28	22	22	18	17	**92**	0	7

Table D.2: Results from a simulation study comparing different Cross Validation schemes on wavelet signals with SNR of 5. We tabulate the number of times out of 100 trials that each scheme *beats the SURE threshold* by having a lower MSE. Note that we considered more schemes than we could include in the table. N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G.

Signal	N2h	N2s	M2.1	M2.2	M3.2	M4.2	M5.2	M6.2	M7.2	M8.2	M512	M512.1	M512.2
Doppler	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Heavisine	**0**	**0**	2	**0**	**0**	**0**	1	1	2	3	**0**	48	4
Blocks	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Cubic	2	**0**	54	18	26	43	54	57	63	62	1	99	79

Table D.3: Results from a simulation study comparing different Cross Validation schemes on wavelet signals with SNR of 5. We tabulate the number of times out of 100 trials that each scheme *loses to the universal threshold* by having a higher MSE. Note that we considered more schemes than we could include in the table. N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G.

Signal	N2h	N2s	M2.1	M2.2	M3.2	M4.2	M5.2	M6.2	M7.2	M8.2	M512	M512.1	M512.2
Doppler	2	**14**	3	12	10	8	5	3	6	3	0	0	3
Heavisine	9	0	6	17	10	9	6	4	3	2	**19**	1	1
Blocks	3	0	5	**5**	1	2	1	4	2	3	0	1	3
Cubic	37	1	1	11	7	5	3	3	1	0	**48**	0	0

Table D.4: Results from a simulation study comparing different Cross Validation schemes on wavelet signals with SNR of 5. We tabulate the number of times out of 100 trials that each scheme *achieves the Oracle MSE* by selecting a threshold that gives the same MSE (or lower, in the case of N2s on occasions). Note that we considered more schemes than we could include in the table. N2h and N2s are Nason's twofold hard and soft results, and MV.G denotes our method i.e. a V-fold version of McCoy's scheme with regularisation G. We have highlighted the best performing methods in bold for each.

D.2 Wavelets for Irregularly-Spaced Data

D.2.1 Simulation Study 1: Minimising *truemse*

In Section 7.3.2, we left out Tables D.5 and D.6 as our gridding schemes were not well suited to noisy data. We include them here instead. See Section 7.3.2 for full details on the simulation study.

Signals	Kovac	mdist	interpmse	mentropy
Doppler	188	295	230	**302**
Heavisine	245	264	203	**312**
Blocks	**365**	217	85	**365**
Cubic	273	**275**	208	268

Table D.5: Results from a simulation study to find the best performing measure out of *mdist*, *interpmse* and *mentropy* in picking a grid to minimise *truemse* (the MSE at the regular grid points, (7.8)) for different signals with SNR of 5. We compare these to the standard Kovac gridding too. The tabulated values are the number of times each gridding scheme produced the smallest *truemse* out of 1000 trials (note that there may be ties for first place so the rows do not sum to 1000). We have highlighted the best performing methods in bold for each.

Signals	Oracle	Kovac	mdist	interpmse	mentropy
Doppler	0.0871 (0.004948)	0.08867 (0.005042)	**0.08846 (0.005042)**	0.08856 (0.004999)	**0.0884 (0.005023)**
Heavisine	11.35 (0.512)	11.46 (0.5156)	**11.45 (0.5178)**	11.46 (0.5133)	**11.45 (0.516)**
Blocks	5.662 (0.4232)	**5.725 (0.4306)**	5.733 (0.4271)	5.746 (0.428)	**5.724 (0.4247)**
Cubic	0.6369 (0.03025)	**0.6427 (0.03054)**	**0.6429 (0.0305)**	0.6431 (0.0303)	**0.6429 (0.03088)**

Table D.6: Results from a simulation study on irregularly spaced data from four wavelet signals with SNR of 5. We tried to minimise *truemse* (the MSE at the regular grid points, (7.8)) using four schemes for selecting grid points: standard Kovac gridding and those minimising measures *mdist*, *interpmse* and *mentropy*. We tabulate mean and standard deviation (in parentheses) of the achieved *truemse* for each of these gridding schemes alongside the oracle gridding scheme. We have highlighted the best performing methods in bold for each.

D.2.2 Simulation Study 2: Smoothing Irregular Datapoints

In Section 7.3.2, we omitted Table D.7 which we include here instead. This uses a high value of SNR of 10 which is typically less useful as the denoising task is easier, but here we demonstrate that our best gridding by entropy is well suited to such a scenario. See Section 7.3.2 for full details on the simulation study.

Signals	kov.re	kov.adj	bg.re	bg.adj	spl.smooth	ker.smooth
Doppler	0.6799 (0.1906)	0.7319 (0.2406)	**0.1094 (0.03056)**	**0.1112 (0.03355)**	0.2137 (0.1642)	5.916 (3.423)
Heavisine	8.488 (2.309)	8.655 (2.4)	7.798 (1.744)	7.76 (1.65)	**3.947 (1.236)**	198.2 (169.9)
Blocks	43.91 (19.24)	43.98 (18.84)	**6.534 (2.216)**	**6.366 (2.12)**	14.36 (8.432)	226.9 (101.3)
Cubic	0.1797 (0.0554)	0.1794 (0.05534)	0.3555 (0.07601)	0.3552 (0.07469)	**0.06574 (0.03156)**	2.505 (2.298)

Table D.7: Results from a simulation study smoothing irregularly spaced datapoints with SNR of 10. We tabulate mean and standard deviation (in parentheses) of the MSE calculated at the irregular datapoints of six competing methods: kov.re, Kovac's wavelet interpolation with re-interpolation onto the irregular points; kov.adj, Kovac's wavelet interpolation with adjustments to reconstruct the irregular points; bg.re, entropy-based best gridding for Kovac's wavelet interpolation with re-interpolation; bg.adj, entropy-based best gridding with adjustments; spline smoothing; kernel smoothing. We have highlighted the best performing methods in bold for each.