Efficient Algorithms for Data Analytics in Geophysical Imaging

Joseph Lee Kump

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Mathematics

Eileen R. Martin, Chair
Mark P. Embree
Russell J. Hewett

May 4, 2021
Blacksburg, Virginia

Keywords: Algorithms, Wavelets, Cross-correlations, MASW, GPU

Efficient Algorithms for Data Analytics in Geophysical Imaging

Joseph Lee Kump

(ABSTRACT)

Modern sensing systems such as distributed acoustic sensing (DAS) can produce massive quantities of geophysical data, often in remote locations. This presents significant challenges with regards to data storage and performing efficient analysis. To address this, we have designed and implemented efficient algorithms for two commonly utilized techniques in geophysical imaging: cross-correlations, and multichannel analysis of surface waves (MASW). Our cross-correlation algorithms operate directly in the wavelet domain on compressed data without requiring a reconstruction of the original signal, reducing memory costs and improving scalabiliy. Meanwhile, our MASW implementations make use of MPI parallelism and GPUs, and present a novel problem for the GPU.

Efficient Algorithms for Data Analytics in Geophysical Imaging

Joseph Lee Kump

(GENERAL AUDIENCE ABSTRACT)

Modern sensor designs make it easier to collect large quantities of seismic vibration data. While this data can provide valuable insight, it is difficult to effectively store and perform analysis on such a high data volume. We propose a few new, general-purpose algorithms that enable speedy use of two common methods in geophysical modeling and data analytics: cross-correlation, which provides a measure of similarity between signals; and multichannel analysis of surface waves, which is a seismic imaging technique. Our algorithms take advantage of hardware and software typically available on modern computers, and the mathematical properties of these two methods.

Dedication

Dedicated to Greg "Bear" O'Bryan

A teacher, and a very good one.

Acknowledgments

First I'd like to thank my advisor, Dr. Eileen Martin, for providing the initial formulations of these research questions, and excellent advice and suggestions the entire way.
I'd also like to thank Dr. Will Ray and Dr. Derek Rountree, the contacts for the projects motivating this work. It's worth mentioning the respective grants:
Subcontract 4000175567, UT-Batelle, LLC for Oak Ridge National Laboratory
DOE Geothermal Technologies Office STTR Phase I Grant DE-SC0019630, Collaboration with Dr. Derek Rountree at Luna Innovations

Next I'd like to thank Dr. Mark Embree and Dr. Russell Hewett, both for their guidance as committee members, and for their valuable mentoring.

I should also name every other professor who has helped me reach this point, including all the great course instructors I've had. But then I'd need to list half the Math department's faculty!

Lastly, I'd like to thank my friends and family for their support. Especially my parents, sister, and (perhaps most importantly) our dog. I didn't expect the last 14 months to turn out the way they did, but you all made it far more manageable.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit

DAS Distributed Acoustic Sensing

DWT Discrete Wavelet Transform

FFT Fast Fourier Transform

GPU Graphics Processing Unit

MASW Multichannel Analysis of Surface Waves

MPI Message Passing Interface

Chapter 1

Introduction

Distributed Acoustic Sensing (DAS) is a geophysical method that applies optical interferometry to laser light travelling inside an optical fiber, which enables the measurement of the fiber's strain. This allows the optical fiber to operate as a seismic array which can sample geophysical vibration data at meter-scale intervals. DAS can be deployed in many locations where it is unfeasible to use more traditional sensors due to space, accessibility, or power constraints, and can acquire data for a wide variety of applications in geophysics [10].

DAS arrays can possess a sensor density 10-1000 times greater than traditional systems, and are capable of collecting data at high sample rates for extended periods of time. This often generates large quantities of vibration data. For example, one DAS array used for test cases in this thesis collected 500 samples per second continuously over several months, producing over two terabytes of data every week. This data volume can be much larger than quantities typical in experiments using other sensor systems. In addition, since DAS can be deployed in remote locations, available storage and hardware to process these high volumes of data may be limited. This leads to challenges regarding data management and monitoring for quality, effective archiving of data, and performing efficient data analysis [4, 11].

This thesis proposes novel algorithms for two methods that are frequently used in analysis of DAS data, as well as geophysical data collected by other means. The first regards wavelet cross-correlations. Wavelet compression is a common method for reducing data volume while storing an accurate approximation of the original data in geophysics and other fields, and

has been used in the storage of DAS data. It can also be employed as a form of noise reduction. Meanwhile, cross-correlation analysis is a measure of similarity between two series, which is frequently used for event detection and pattern recognition in data analytics. Typical methods for computing cross-correlations of time-series data stored in the wavelet domain require reconstructing the original data, or its approximation, which negates the benefit of using wavelet compression to reduce data volume. Here, we propose an algorithm for calculating the cross-correlation of two time-series data signals directly in the wavelet domain, without needing to reconstruct the original signal. This approach eliminates the storage requirements of reconstruction, and has the potential to scale better than existing cross-correlation algorithms on larger problem sizes.

The second method is Multichannel Analysis of Surface Waves (MASW), a common seismic exploration technique used to model the near-surface. In MASW, recording of vibrations generated by a noise source are used to construct a dispersion curve of the ground roll. Proposed models of the near surface are then used to calculate theoretical dispersion curves, which are compared to the dispersion curve derived from the collected data. DAS arrays can record vibration data corresponding to Rayleigh wave signals that make up ground roll (the main type of coherent noise in land seismic surveys). Because of this, it is viable to use MASW in conjunction with DAS to develop predictive models of the near surface. While MASW is a well-established technique, existing open-source implementations are not optimized for commonly available computing resources such as multi-core CPUs and GPUs. Thus, there is value in developing algorithms that leverage these resources.

The algorithms and resulting implementations for these two methods are generalized to work for data collected by other means, not just DAS. However, their optimizations can make them advantageous for addressing some of the challenges presented by modern applications involving DAS data, such as high data volume and limitations in readily available

computing resources and storage. In addition, the algorithm design and implementation for both methods share some common themes, particularly taking advantage of specific sparse matrix structures, such as bandedness, to improve performance.

1.1 Wavelet Cross-Correlation

Cross-correlation, also called a sliding inner product, is a measurement of similarity between two series at a time lag. Given time series functions f and g and a time-lag τ, the cross-correlation of f and g at τ is

$$(f \star g)(\tau) = \sum_{i=-\infty}^{\infty} \overline{f(i)} g(i+\tau) \text{ or } \int_{-\infty}^{\infty} \overline{f(t)} g(t+\tau) dt. \tag{1.1}$$

There are two major use cases for cross-correlation on a multichannel sensor array. The first is event detection: computing the cross-correlations of the array's channels with a signal from a known event, such as a small earthquake or the operation of a certain machine. This can identify the time and location of those events within the sensor data. The other major use case is comparing the channels of the array with each other – if two channels have a consistently high cross-correlations at a specific time lag, then that suggests waves propagate through the channels at that particular time interval. This method can help identify underlying trends in the data (it is also useful to compute the cross-correlation of a signal with itself to identify repeating trends, which is called autocorrelation).

Cross-correlation is conceptually simple and serves as an important method for analysis of sensor data. However, for a signal with N entries, computing the cross-correlation at one time lag requires an order of N operations, or $O(N)$, the same as computing an inner product. Computing cross-correlations at multiple channels and time lags (which is needed for event

detection and trend identification) can cause the number of arithmetic operations to increase quickly. Thus it is important to develop algorithms that can calculate cross-correlations efficiently. Given the size of many large sensor arrays, it is important to use methods that minimize the storage and computational costs of data analyses like cross-correlation.

The Fourier Transform

The Fourier transform is a generalization of the Fourier series that decomposes a signal into its component frequencies; in other words, it converts a signal from representation in the time domain to representation in the frequency domain. Fourier transforms, particularly discrete Fourier transforms, are useful for visualizing and processing components of signal data and are quick to compute through the use of fast Fourier transform (FFT) algorithms [17]. They also have mathematical properties that can make computing cross-correlations quicker, particularly the cross-correlation theorem: let $\mathcal{F}$ denote the Fourier transform, then

$$\mathcal{F}\{f \star g\} = \overline{\mathcal{F}\{f\}} \cdot \mathcal{F}\{g\}. \tag{1.2}$$

FFTs combined with this theorem enable efficient algorithms for computing cross-correlations. However, while this approach can improve the computational speed, it does not improve the storage costs – the transformed data requires the same storage volume as the original data. In addition, Fourier transforms do not preserve the exact times at which specific frequency components occur and therefore may not be useful for other forms of analysis. Because of these limitations, the Fourier transform generally cannot be used as a method for permanently storing sensor data.

The Wavelet Transform and Compression

Wavelets are integrable functions that are nonzero only on a finite interval and can typically

be visualized as an oscillation. Given a specific mother wavelet function, a basis can be formed by the mother wavelet, its horizontal translations (referred to as shifts), and its horizontal stretches (referred to as scales). If the shift factors used are in $\mathbb{Z}$ and the scale factors used are in $2^{\mathbb{Z}_{\geq 0}}$, then this collection of wavelets forms an orthogonal basis which can be normalized. The wavelet transform represents a signal as a linear combination of wavelets formed by one of these bases – usually shift and scale factors are selected to guarantee it is an orthonormal basis, which makes it a discrete wavelet transform, or DWT [2].

Unlike the Fourier transform, the wavelet transform preserves both time and frequency information from the original signal. It does require a similar volume of storage as the untransformed data, but this can be reduced by storing an approximation of the wavelet transform instead, which is known as wavelet compression. One of the more common techniques used for wavelet compression is thresholding, where all wavelets whose coefficients are smaller in magnitude than a particular cutoff are zeroed out. Even if compression is not used, wavelet transforms can allow for easy cross-correlation computations. Since it is a linear operator, the cross-correlations of individual wavelet functions can be precomputed in advance. From there the cross-correlation of the two signals can be calculated as a sum of these results, dependent on the DWT coefficients.

Like in many other fields, Geoscientists have employed wavelets to store and compress geophysical data since the 1990s. However, their typical use in cross-correlations and other data analysis applications involve reconstructing the original time-domain signal or an approximation using an inverse wavelet transform. Current wavelet transforms and inverse transform algorithms are fast, but this approach requires storing a volume of time-domain data equivalent in memory cost to the original signal. This requirement is not ideal for use cases where data is being collected and stored in remote locations (such as glacier and permafrost monitoring), or where the user is paying for data volume (such as cloud storage),

both of which are increasingly common in modern edge computing paradigms in geophysical data collection and other applications [7]. Developing algorithms for data analytics directly in the wavelet domain, without reconstructing the original signal, could mitigate these storage costs. In addition, wavelet-domain algorithms that take advantage of properties of compressed wavelet-domain data may see improved speed performance and scaling.

Since cross-correlation is widely used in data analytics, it provides a good example of a method that may benefit from a wavelet-domain implementation. Eventually, a collection of multiple wavelet-domain algorithms may be developed for several common methods used in analysis of geophysical data, but an effective cross-correlation algorithm provides a useful starting point. In addition, concepts that enable efficient computation of cross-correlations on DWT coefficients may also be extended to other orthogonal or near-orthogonal bases used to represent data.

1.2 MASW

Multichannel Analysis of Surface Waves (MASW) is a seismic exploration technique used to infer a layered 1D model of the subsurface. It was developed by the Kansas Geological Survey in 1999, as an enhancement to an earlier wave-propagation method called Spectral Analysis of Surface Waves (SASW).

In SASW, two receivers record the surface wave energy that travels along the ground, called the ground roll, that is generated by an impulsive source such as a hammer strike. The ground roll consists primarily of Rayleigh waves and has a special property known as dispersion: each of its frequency components has a different propagation velocity, and thus a different wavelength as well. Therefore one can use the recorded ground roll in spectral analysis to generate a dispersion curve, which plots the relationships between the frequencies

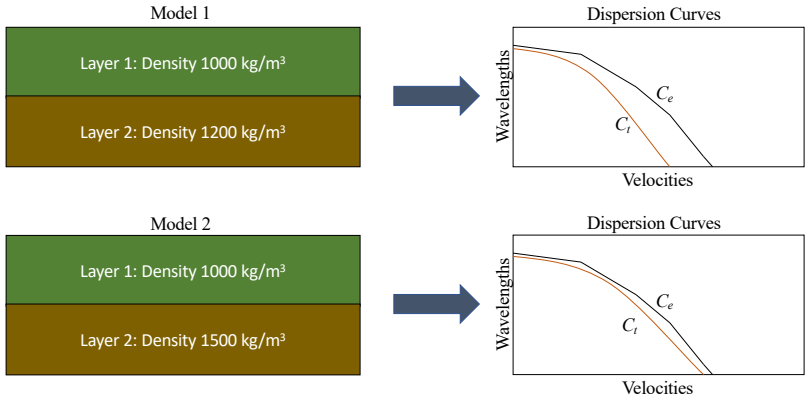

Figure 1.1: Simplified example of dispersion with multiple proposed near-surface models. Model 2 (which has a different density in the second layer) produces a theoretical dispersion curve C_t closer to the experimentally derived dispersion C_e, so it is likely a better model.

and velocities of each component of the ground roll (or alternatively the wavelengths and velocities). Using only two receivers causes the SASW method to collect a large quantity of unnecessary noise in this process, which is difficult to reduce. MASW mitigates this problem by using an array of multiple receivers (such as a DAS array), making it easier to isolate the useful ground roll and construct its dispersion curve. This part of the method is called MASW dispersion, and only needs to be done once for any particular ground site and set of collected seismic data [13, 14].

Once a dispersion curve of the near-surface ground roll is acquired, we can then propose and assess the potential accuracy of 1D models of the near surface using MASW inversion, also known as backcalculation. Given a proposed near-surface model consisting of multiple layers, each with its own thickness, density, compressional wave velocity, and shear wave velocity, we can develop a theoretical dispersion curve of the ground roll. This can then be compared to the experimental dispersion curve produced from spectral analysis of the collected data to assess the overall accuracy of the proposed model. The closer the theoretical and experimental dispersion curves are in value, the more likely the model is reasonably accurate - a simplified example of this is shown in Figure 1.1. We often need to perform

MASW inversion on a large number of proposed near-surface models to acquire a satisfactory result, so unlike MASW dispersion, it is typically run many times for a given problem. Different methods are available to develop theoretical dispersion curves. MASWaves, a commonly used MATLAB MASW implementation, uses a technique called the stiffness matrix method. This method requires the formation of many small, symmetric heptadiagonal matrices. Depending on the proposed model, this is typically between 10^4 and 10^5 matrices of size $n \times n$, where $20 \leq n \leq 100$. We then need to asses the singularity of these matrices, possibly by using determinants or singular values.

Since MASW does not require invasive drilling or sample collection, it is a commonly used method for near-surface imaging in geotechnical engineering. There are many existing open-source implementations, freely available for use. However, there are currently no open-source implementations that optimize MASW inversion for multiple compute cores or GPUs, specifically the computation of the theoretical dispersion curve, which can be run hundreds of times or more for various near-surface models proposed for a single problem. Multiple CPU cores and GPUs are both fairly common features for engineering workstations, so it is advantageous to design parallelized MASW algorithms to take advantage of these computing resources.

In addition to there being a practical need for GPU implementations, MASW inversion using the stiffness matrix method provides a use case for a potentially novel problem on the GPU. There are many existing libraries, such as cuBLAS, that provide various CUDA functions for operations on large matrices, which are relevant to a range of problems in scientific computing. However, there are few existing CUDA library functions for performing operations on a large array of smaller matrices concurrently, and none that take advantage of performance-improving matrix properties such as bandedness (at least to our knowledge). Since the stiffness matrix method requires assessment of the singularity of a large number of

small, heptadiagonal matrices, it provides context for a novel type of problem to implement on GPUs.

1.3 Thesis Overview and Contributions

This thesis consists of algorithm descriptions for both of these problems: wavelet-domain cross-correlations using both dense and sparse DWT coefficients, and MASW inversion optimized for multiple CPUs and GPUs. The wavelet-domain cross-correlation algorithms are covered in chapter 2, while the parallelized MASW inversion is covered in chapter 3. Sections for chapter 2 are:

1. A brief overview;
2. Theoretical properties used in the algorithm design;
3. Algorithm description;
4. Analytically derived error bounds;
5. Test cases and examples.

Sections for chapter 3 are:

1. A brief overview;
2. Serial MASW algorithm description;
3. Parallelized algorithm description (using MPI);
4. GPU algorithm description (using CUDA);

5. Test cases and performance evaluation.

Chapters 2 and 3 can be read out of order, though the sections within each should be read in order. A final overall conclusion is in chapter 4. The author made all design choices for the described algorithms, and wrote the software implementations. This thesis contains multiple original contributions:

- Novel properties of cross-correlations for wavelet and scaling functions, that can be used for their efficient computation and storage (section 2.2);
- An algorithm for computing the accurate temporal cross-correlation of two time-series signals stored in the wavelet domain (section 2.3), plus software implementation;
- Theoretical error bounds for cross-correlations on wavelet-compressed (via thresholding) signals compared to the original signal cross-correlation (section 2.4);
- An MASW implementation that runs on multiple compute cores using MPI (section 3.3);
- An MASW implementation for use with GPUs (section 3.4), which is optimized for computing determinants of heptadiagonal matrices;

All software implementations are shown to have efficient performance results with certain advantages over current standard methods.

Chapter 2

Cross-Correlations in the Wavelet Domain

2.1 Overview of Problem

Wavelet compression is already used as a means to store data at a reduced volume. However, current cross-correlation implementations perform an inverse wavelet transform on this data, then compute the cross-correlations on these reconstructed signals, using either traditional sliding inner products or FFTs with the cross-correlation theorem. These approaches are reasonably fast, but do not take full advantage of the sparsity in the wavelet-compressed data, and require extensive memory.

There are existing methods that compute useful components of the cross-correlation on wavelet domain data. For example, search techniques have been developed for detecting stochastic gravitational waves using the sign correlation test on data stored in the wavelet domain [8]. There has also been exploration of using other analytics methods, such as convolutional neural networks, on wavelet-domain data, though these have been used for image processing and not geophysical applications [3]. However, an algorithm for accurately computing the temporal cross-correlation in the wavelet domain has not yet been designed.

As an alternative to take advantage of certain aspects of wavelet compression, we propose an

algorithm and implementation for computing cross-correlations in the wavelet domain - since the discrete wavelet transform (DWT) stores a signal as a linear combination of wavelet and scaling functions, we can precompute the cross-correlations of these individual functions with each other, then calculate the overall cross-correlation as a linear combination of these stored values with the wavelet coefficients, analogous to a weighted inner product or a Rayleigh quotient, though not necessarily an instance of these.

2.2 Properties Used For Algorithm

Suppose we have two sets of time-series data "signals" (such as from two sensors in a DAS array), denoted d^r and d^s, represented by their coefficients in a level J wavelet basis $\{d_j^{(r)}[n]\}_{j,n} \cup \{a_J^{(r)}[n]\}_n$ generated by a DWT [2]. We can store these coefficients as vectors $\vec{x}^{(r)}$ and $\vec{x}^{(s)}$. Given the set of cross-correlations which run through all permutations of these scale and shift factors, we can form a correlation matrix $W^{(\tau)}$ consisting of these cross-correlations at time lag τ, stored in the same order as in $\vec{x}^{(r)}$ and $\vec{x}^{(s)}$. From there, we can compute the cross-correlations of the two signals at τ by using a vector-matrix-vector multiplication, structured as

$$(d^r \star d^s)(\tau) = \vec{x}^{(r)T} W^{(\tau)} \vec{x}^{(s)}. \tag{2.1}$$

To store $W^{(\tau)}$ and compute $\vec{x}^{(r)T} W^{(\tau)} \vec{x}^{(s)}$ efficiently, we will use four properties. First, for the ease of implementation, we will break $W^{(\tau)}$ into $(J+1)^2$ blocks, themselves correlation matrices, based on their scaling factors. Note that ψ is the wavelet function corresponding to J detail levels, and ϕ is the scaling function corresponding to one approximation level:

$$W^{(\tau)} = \begin{pmatrix} [(\psi_{1,n} \star \psi_{1,m})(\tau)]_{n,m} & [(\psi_{1,n} \star \psi_{2,m})(\tau)]_{n,m} & \dots & [(\psi_{1,n} \star \phi_m)(\tau)]_{n,m} \\ [(\psi_{2,n} \star \psi_{1,m})(\tau)]_{n,m} & [(\psi_{2,n} \star \psi_{2,m})(\tau)]_{n,m} & \dots & [(\psi_{2,n} \star \phi_m)(\tau)]_{n,m} \\ \vdots & \vdots & \ddots & \vdots \\ [(\phi_n \star \psi_{1,m})(\tau)]_{n,m} & [(\phi_n \star \psi_{2,m})(\tau)]_{n,m} & \dots & [(\phi_n \star \phi_m)(\tau)]_{n,m} \end{pmatrix} \tag{2.2}$$

Let $\vec{x}_j$ denote the entries of $\vec{x}$ for coefficients of ψ_j, and $W_{j,k}^{(\tau)} = [(\psi_{j,n} \star \psi_{k,m})(\tau)]_{n,m}$ for $j = 1, \dots J+1$ (where ϕ is treated as the level $J+1$ function). Correlation block matrices along the main diagonal are square, since the number of coefficients for each signal at a particular scaling factor is the same. Matrices below the diagonal will have more columns than rows, since lower scaling factors will have more coefficients than higher scaling factors, while matrices above the diagonal will have transposed dimensions. The scaling function ϕ has the same number of coefficients as the wavelet function at level J, ψ_J, so the inequality is not strict for block matrix dimensions off of the main diagonal.

Assuming our coefficient vectors $\vec{x}$ are stored in the same order (coefficients for $\psi_1, \dots, \psi_J$, then ϕ), then we can perform our cross-correlation as a series of vector-matrix-vector products:

$$(d^r \star d^s)(\tau) = \vec{x}^{(r)T} W^{(\tau)} \vec{x}^{(s)} = \sum_{j=1}^{J+1} \sum_{k=1}^{J+1} \vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}. \tag{2.3}$$

Implementing the wavelet-domain cross-correlation this way is advantageous, since each correlation matrix $W_{j,k}^{(\tau)}$ consists of cross-correlations between functions with fixed scaling factors. The size of $W_{j,k}^{(\tau)}$ is dependent on the number of shift factors for each set of functions, which is in turn dependent on the length of the signals. However, we can use properties of $W_{j,k}^{(\tau)}$ to show it has a specific sparse structure, and its total number of unique entries is only

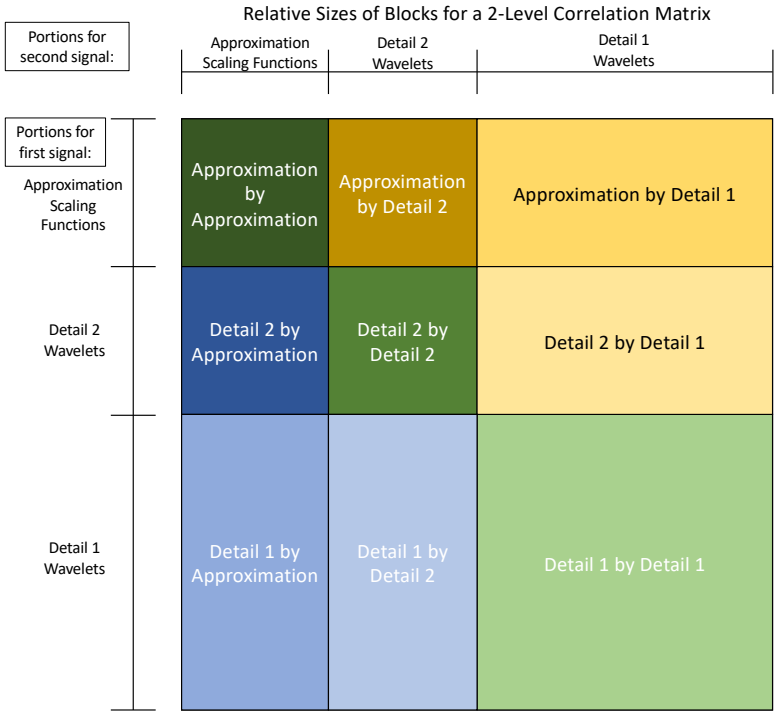

Figure 2.1: Relative sizes of blocks in a correlation matrix. Same-level submatrices are square, while different level matrices are rectangular with their size ratio proportional to the differences in scaling factors. "Detail 1 by Detail 1" corresponds to $W_{1,1}^{(\tau)}$.

dependent on the scaling factors j, k. This sparsity reduces both the volume of data required to store the entries of the individual correlation matrices, and the number of floating-point operations required to compute the cross-correlation with this method. First, we will prove a preliminary claim.

Claim 1: Time-Shift Transform The cross-correlation of two wavelet or scaling functions after being time shifted is equal to a time-shifted version of their original cross-correlation. Let $\theta^{(g)}, \theta^{(h)}$ denote two arbitrary wavelet or scaling functions in our given wavelet family (they may be different, hence the different superscripts). Then

$$(\theta_{j,n'}^{(g)} \star \theta_{k,m'}^{(h)})(\tau) = (\theta_{j,n}^{(g)}(t) \star \theta_{k,m}^{(h)})(\tau + 2^k(m - m') - 2^j(n - n')). \tag{2.4}$$

The calculation to show this is done by adding and subtracting multiples of n and m to n' and m', and performing the change of variables $s = t + 2^j(n - n')$:

$$\begin{aligned}
&\left(\theta^{(g)}_{j,n'} \star \theta^{(h)}_{k,m'}\right)(\tau) \\
&= \int_{-\infty}^{\infty} \theta^{(g)}_{j,n'}(t)\theta^{(h)}_{k,m'}(t+\tau)dt \\
&= \int_{-\infty}^{\infty} 2^{-j/2}\theta^{(g)}\left(\frac{t-2^jn'}{2^j}\right) 2^{-k/2}\theta^{(h)}\left(\frac{t+\tau-2^km'}{2^k}\right) dt \\
&= \int_{-\infty}^{\infty} 2^{-j/2}\theta^{(g)}\left(\frac{(t+2^jn-2^jn')-2^jn}{2^j}\right) 2^{-k/2}\theta^{(h)}\left(\frac{t+\tau-2^km'}{2^k}\right) dt \\
&= \int_{-\infty}^{\infty} 2^{-j/2}\theta^{(g)}\left(\frac{s-2^jn}{2^j}\right) 2^{-k/2}\theta^{(h)}\left(\frac{s+\tau-2^km'-2^j(n-n')}{2^k}\right) ds \\
&= \int_{-\infty}^{\infty} 2^{-\frac{j}{2}}\theta^{(g)}\left(\frac{s-2^jn}{2^j}\right) 2^{-\frac{k}{2}}\theta^{(h)}\left(\frac{s+\tau+2^k(m-m')-2^j(n-n')-2^km}{2^k}\right) ds \\
&= \left(\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m}\right)(\tau + 2^k(m-m') - 2^j(n-n')).
\end{aligned}$$

Now that we have proven this claim, we define, prove, and apply the Toeplitz-like, Banded-like, Symmetric, and Cyclic properties to reduce the memory footprint of the $W^{(\tau)}$ representation.

Property 1: Toeplitz-Like Suppose we are computing the cross-correlations of two wavelet or scaling functions, $\theta^{(g)}_{j,n+l}$, $\theta^{(h)}_{j,m+l}$, with the same scaling factor, j. We can then use Claim 1 to show

$$(\theta^{(g)}_{j,n+\ell} \star \theta^{(h)}_{j,m+\ell})(\tau) = (\theta^{(g)}_{j,n} \star \theta^{(h)}_{j,m})(\tau + 2^j(-\ell) - 2^j(-\ell)) = (\theta^{(g)}_{j,n} \star \theta^{(h)}_{j,m})(\tau). \tag{2.5}$$

Let $w_{n,m}$ be the row n, column m entry within a given block submatrix, $W^{(\tau)}_{j,j}$, storing the cross-correlations of same-scale wavelet functions. Equation (2.5) implies that $w_{n+\ell,m+\ell} =$

$w_{n,m}$, thus giving us the Toeplitz property. A graph visualizing this property in effect is available in Figure 2.2. If our matrix stores the cross-correlations of functions with different scales, then it will not be Toeplitz, however, we can see a similar Toeplitz-like relationship:

$$\begin{aligned}(\theta^{(g)}_{j,n+\ell} \star \theta^{(h)}_{k,m+2^{j-k}\ell})(\tau) &= (\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(\tau + 2^j(-\ell) - 2^k(-(2^{j-k})\ell)) \\ &= (\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(\tau - 2^j\ell + 2^{k+j-k}\ell) \\ &= (\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(\tau). \end{aligned} \tag{2.6}$$

Thus for entries in the correlation matrix $W_{j,k}$, we get $w_{n+\ell,m+2^{j-k}\ell} = w_{n,m}$. This is generally not Toeplitz unless $j = k$, but still provides a useful repeating pattern for reducing the number of entries of the cross-correlation matrix that must be stored. In particular, if $j > k$, then we know row $n+\ell$ equals row n shifted $2^{j-k}\ell$ to the right. If $j < k$, then we instead get a repeating pattern for the columns of $W_{j,k}$. Although this is true, it is easier to use other properties to help represent these block matrices in a row-major format instead to keep the implementation consistent.

Property 2: Banded-Like A wavelet or scaling function $\theta^{(g)}_{j,n}$ is supported on only a finite interval, assuming we are using compact wavelets. Suppose the width of the support is length α. We know at 0 time-lag that

$$\begin{aligned}(\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(0) &= (\theta^{(g)}_{j,0} \star \theta^{(h)}_{k,m-2^{j-k}n})(0) \\ &= \int_{-\infty}^{\infty} \theta^{(g)}\left(\frac{t}{2^j}\right) \theta^{(h)}\left(\frac{t - 2^k(m - 2^{j-k}n)}{2^k}\right) dt. \end{aligned} \tag{2.7}$$

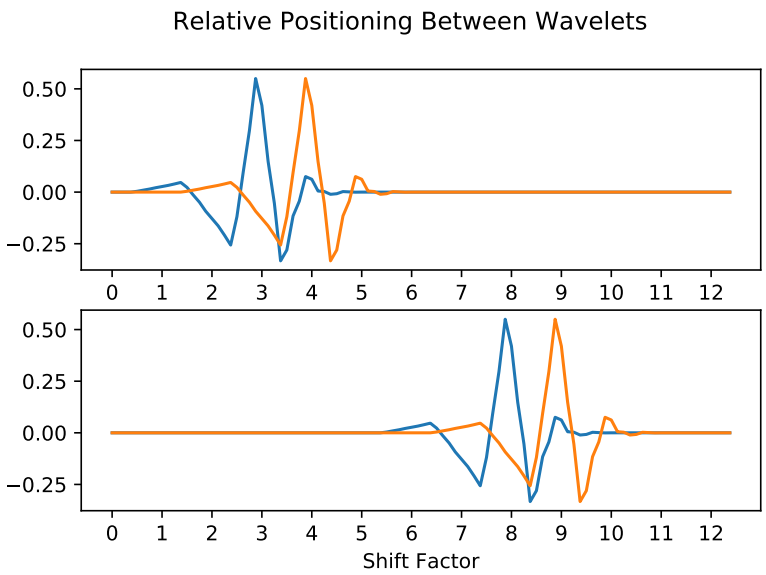

Figure 2.2: Comparison of wavelets with shift factors 0 and 1 to wavelets with shift factors 5 and 6. Since the relative difference in shift factor is the same and the functions themselves are the same except for their shift factors, the resulting cross-correlations of shift factors 0 with 1 and 5 with 6 are identical.

If $|m - 2^{j-k}n| \geq \alpha$, then $\theta^{(g)}_{j,n}$ and $\theta^{(h)}_{k,m}$ have no measurable overlap in support, which gives us

$$|m - 2^{j-k}n| \geq \alpha \implies (\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(0) = 0. \tag{2.8}$$

Since the scaling factors j and k of the wavelets in this cross-correlation are fixed for a given block matrix $W^{(0)}_{j,k}$, and the shift factors n and m determine the row and column index of $W^{(0)}_{j,k}$ corresponding to this cross-correlation, our block matrix is nonzero at $[W^{(0)}_{j,k}]_{n,m}$ only when $|m - 2^{j-k}n| < \alpha$. This holds when $\tau \neq 0$ as well:

$$\begin{aligned}(\theta^{(g)}_{j,n} \star \theta^{(h)}_{k,m})(\tau) &= (\theta^{(g)}_{j,0} \star \theta^{(h)}_{k,m-2^{j-k}n})(\tau) \\ &= \int_{-\infty}^{\infty} \theta^{(g)}\left(\frac{t}{2^j}\right) \theta^{(h)}\left(\frac{t - 2^k(m - 2^{j-k}n) + \tau}{2^k}\right) dt \end{aligned} \tag{2.9}$$

For this given value of τ, if $|m - 2^{j-k}n + \frac{\tau}{2^k}| \geq \alpha$, then $\theta_{j,0}^{(g)}$ and $\theta_{j,n_2-n_1}^{(h)}$ have no measurable overlap in support, so $(\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(\tau) = 0$. Thus we can still restrict the number of possible finite terms in each row of $W_{j,k}^{(\tau)}$ based on this inequality. However, the nonzero entries of $W_{j,k}^{(\tau)}$ may be different from $W_{j,k}^{(0)}$.

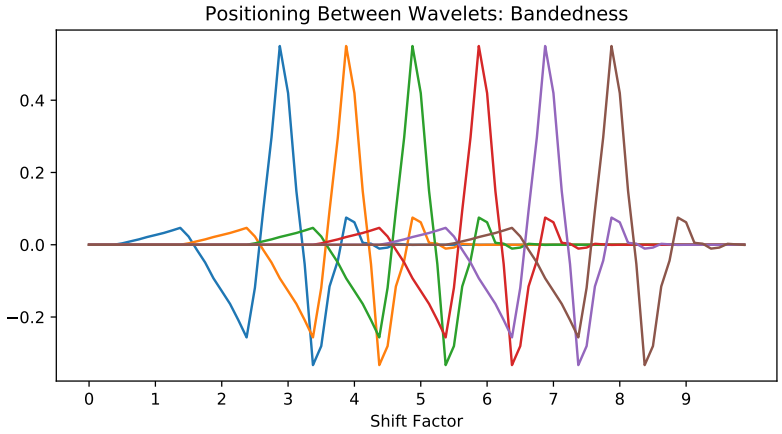

Figure 2.3: Plot of the overlap in support between wavelets with nearby shift factors. Once the relative difference in shift factor becomes too large, such as shift 0 (blue) and 5 (brown), then the overlap in support and resulting cross-correlation is always 0.

If $j = k$, then this gives us bandedness since the matrix is Toeplitz. If not, then this property still restricts the number of nonzero entries per row to a value dependant on the scaling factors only, based on how the column index relates to the row index as shown in Figure 2.3. Moreover, we know which column indices can be nonzero based on the row index.

Since the Toeplitz-like property ensures the matrix rows have repeating entries for $j \geq k$, and the Banded-like property ensures there are only finitely many nonzero entries per row, we can represent our block matrix $W_{j,k}^{(\tau)}$ using only a small number of entries, representing the nonzero entries in one row. In practice, we also need to store the cross-correlations of the first few and last few wavelets in the signal as well, since these wavelets are only partially supported on the signal's time interval and thus their cross-correlations have different values. But the number of edge wavelets is small (typically < 10) and not dependent on the signal

length. So we can represent $W_{j,k}^{(\tau)}$ using one row of interior wavelet function cross-correlations, and two small matrices of the beginning- and end-of-signal wavelet cross-correlations, with no dependency on the length of the signals being evaluated.

Property 3: Symmetry Suppose we are computing the cross-correlations for two arbitrary wavelet or scaling functions $\theta^{(g)}, \theta^{(h)}$ at time lag 0. We then know

$$\begin{aligned}(\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(0) &= \int_{-\infty}^{\infty} \theta^{(g)}\left(\frac{t-2^j n}{2^j}\right)\theta^{(h)}\left(\frac{t-2^k m}{2^k}\right)dt \\ &= \int_{-\infty}^{\infty} \theta^{(h)}\left(\frac{t-2^k m}{2^k}\right)\theta^{(g)}\left(\frac{t-2^j n}{2^j}\right)dt \\ &= (\theta_{k,m}^{(h)} \star \theta_{j,n}^{(g)})(0). \end{aligned} \tag{2.10}$$

In the case where $g = h$ and $j = k$, this gives us symmetry for $W_{j,k}^{(0)}$, reducing the number of unique entries further. If $g \neq h$ or $j \neq k$, then we do not have matrix symmetry. Symmetry is usually not possible in these cases anyway, since $W_{j,k}^{(\tau)}$ is generally not square for $j \neq k$. However, this property still lets us know that the (n, m) entry of $W_{j,k}^{(0)}$ equals the (m, n) entry of $W_{k,j}^{(0)}$. Therefore, we can substitute $\vec{x}_k^{(r)T} W_{k,j}^{(0)} \vec{x}_j^{(s)}$ for $\vec{x}_j^{(s)T} W_{j,k}^{(0)} \vec{x}_k^{(r)}$ in equation (2.3), and only need to store the block matrices $W_{j,k}^{(0)}$ for $j \geq k$. This allows us to easily use a row-major storage format for all of our zero-time-lag correlation matrices.

At nonzero time-lags, we have quasi-symmetry: $(\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(\tau) = (\theta_{k,m}^{(h)} \star \theta_{j,n}^{(g)})(-\tau)$. This does not reduce the matrix entries needed for $W_{j,j}^{(\tau)}$, but it does allow us to only store block matrices $W_{j,k}^{(\tau)}$ where $j \geq k$, provided we store them for both τ and $-\tau$.

Property 4: Cyclic This property reduces the number of time lags we need to store. First, let $\mathcal{L}$ be an operator on vectors in $\mathbb{C}^n$, that shifts each of the entries of the given vector up by one and sets the first entry equal to 0 (for example, $\mathcal{L}([1, 2, 3]^T) = [0, 1, 2]^T$). Similarly, let

$\mathcal{R}$ denote the operator that shifts the vector entries down by one, and sets the last entry to 0. These are similar to the left and right shift operators in functional analysis. Now suppose we need to store entries for a correlation block matrix $W_{j,k}^{(\tau)}$ at time lag τ. Clearly we can write τ as $2^k d + t$, where $d \in \mathbb{Z}$, and $2^k d$ is the largest multiple of 2^k such that $2^k d \leq \tau$. Then we know $t \in [0, 2^k)$. By Claim 1, we can show

$$\begin{aligned}(\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(\tau) &= (\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(2^k d + t) \\ &= (\theta_{j,n}^{(g)} \star \theta_{k,m}^{(h)})(t + 2^k(m - (m-d)) - 2^j(n-n)) \\ &= (\theta_{j,n}^{(g)} \star \theta_{k,m-d}^{(h)})(t). \end{aligned} \tag{2.11}$$

Thus, we can compute the cross-correlations of signal coefficients $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$ at time lag τ by shifting the entries of $\vec{x}_k^{(s)}$ forward by d (applying $\mathcal{L}^d$) or backwards by $-d$ if $d < 0$ (applying $\mathcal{R}^{-d}$), and then computing the cross-correlation at time lag t instead. This means we do not need to use correlation block matrices $W_{j,k}^{(\tau)}$ for every possible time-lag τ. We can apply one of our shift operators to $\vec{x}_h^{(s)}$ first, then use a block matrix $W_{j,k}^{(t)}$ where $t \in [0, 2^k)$ instead. For example, considering Equation (2.11), where $\tau = 2^k d + t$, we would get

$$\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)} = \vec{x}_j^{(r)T} W_{j,k}^{(t)} \left[\mathcal{L}^d(\vec{x}_k^{(s)})\right] \text{ or } \vec{x}_j^{(r)T} W_{j,k}^{(t)} \left[\mathcal{R}^{-d}(\vec{x}_k^{(s)})\right]. \tag{2.12}$$

We use $\mathcal{L}$ if $d \geq 0$, and $\mathcal{R}$ if $d < 0$. Moreover, our data signals are represented as discrete time series, with points sampled at small time intervals relative to the signal length. Because of this, we typically only compute cross-correlations at integer time lags in our time-series representation. Thus we only need to store correlation block matrices $W_{j,k}^{(\tau)}$ for time lag indices $\tau = 0, 1, \ldots, 2^k - 1$.

Combining the four properties together, we only need to form correlation block matrices $W_{j,k}^{(\tau)}$ for $j, k = 1, \ldots J+1$ with $j \geq k$. Moreover, we only need time lags for each of these block matrices $W_{j,k}^{(\tau)}$ in the range $\tau = 0, \ldots, 2^k - 1$. For each individual block matrix, we only need to store a small portion of the total entries: a short list of the unique nonzero entries on one row, and two small submatrices of cross-correlations for the first few and last few wavelet cross-correlations. These three data structures are a fixed size for any given block matrix $W_{j,k}^{(\tau)}$, and their specific dimensions are described in Section 2.3.

These rules allow us to use a special sparse structure to represent the cross-correlation matrix, reducing storage requirements and the number of operations needed for the cross-correlation computation. Since this implementation of $W^{(\tau)}$ is not dependent on the number of shift factors, it also works for two signals of arbitrary lengths. In particular, we expect the explicit representation of $W_{j,k}^{(\tau)}$ to be roughly of dimension $\frac{N}{2^j} \times \frac{N}{2^k}$ where N is the dimension of our coefficient vectors $\vec{x}^{(r)}, \vec{x}^{(s)}$ (which roughly equals the length of our signals d^r, d^s). Our representation of $W_{j,k}^{(\tau)}$ is a fixed size regardless of N.

2.3 Algorithm Description

Given two signals d^r, d^s whose level J discrete wavelet transform coefficients are stored in vectors $\vec{x}^{(r)}, \vec{x}^{(s)}$, we have shown in Equation (2.3) that

$$(d^r \star d^s)(\tau) = \sum_{j=1}^{J+1} \sum_{k=1}^{J+1} \vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}.$$

We also know from the quasi-symmetry property that $\vec{x}_k^{(r)T} W_{k,j}^{(\tau)} \vec{x}_j^{(s)} = \vec{x}_j^{(s)T} W_{j,k}^{(-\tau)} \vec{x}_k^{(r)}$, so we can also see

$$(d^r \star d^s)(\tau) = \sum_{j=1}^{J+1} \left[\vec{x}_j^{(r)T} W_{j,j}^{(\tau)} \vec{x}_j^{(s)} + \sum_{k=1}^{j-1} \left(\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)} + \vec{x}_j^{(s)T} W_{j,k}^{(-\tau)} \vec{x}_k^{(r)} \right) \right]. \tag{2.13}$$

This allows us to break the wavelet-domain cross-correlation into a sum of vector-matrix-vector multiplications, with matrices that possess the qualities described in section 2.4. In particular, each correlation matrix $W_{j,k}^{(\tau)}$ can be written in a row-major format, with repeating shifted rows on its interior, and two small submatrices at the top left and bottom right. We also know we only need to store $W_{j,k}^{(\tau)}$ for time-lags $\tau \in \{0, 1, \ldots, 2^k - 1\}$, and can use left and right shift operators on $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$ to calculate the cross-correlation accurately at other time-lags.

2.3.1 Computing a Single Time-Lag

Naively computing $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$ as a dense multiplication is inefficient, since most of the entries in $W_{j,k}^{(\tau)}$ are 0. However, we can use the special properties of $W_{j,k}^{(\tau)}$ to compute it much more efficiently. In the case where $j = k$, we know $W_{j,j}^{(\tau)}$ is banded and mostly Toeplitz, so most of its entries can be represented with a small number of scalars equivalent to each of its nonzero diagonals. Create a short vector consisting of the entries of each of these diagonals, and denote it $M_{j,j}$. The only entries of $W_{j,j}^{(\tau)}$ different from those in $M_{j,j}$ are associated with the first and last few wavelets of each signal: these wavelets are not fully represented in the time series of the signal, so their cross-correlations are different. We can represent these using small submatrices denoted $B_{j,j}$ and $E_{j,j}$. A diagram of these components of $W_{j,j}$ is visible in Figure 2.4.

Let $s_u(j)$ denote the number of nonzero values of the numeric representation of our given wavelet function at level j (i.e. its support), and let $s_h(j)$ denote the size of a single shift of the numeric representation of our given wavelet at level j. In general, we expect both

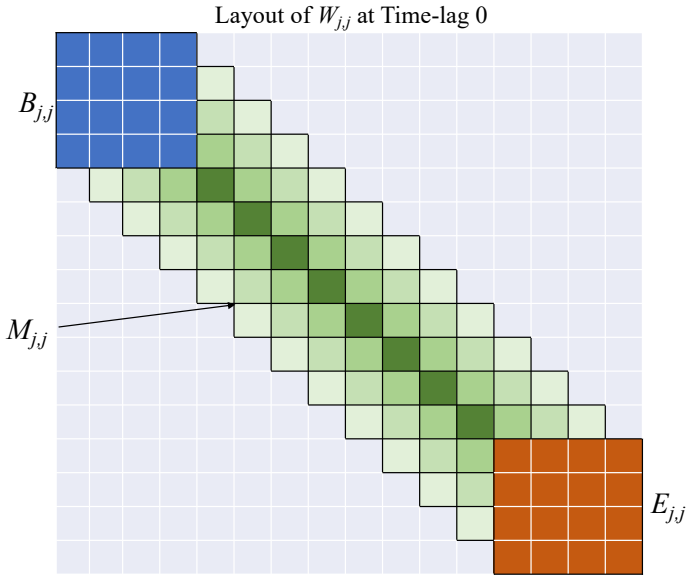

Figure 2.4: Layout of a same-level weight matrix $W_{j,j}$. Interior entries ($M_{j,j}$, colored green) repeat along each diagonal, so we only need to store one row. Entries for $B_{j,j}$ and $E_{j,j}$ are unique. If the time-lag is not zero, then $M_{j,j}$ is no longer symmetric, but is still banded and Toeplitz.

$s_u(j)$ and $s_h(j)$ to increase as j increases, since wavelets with a larger scaling factor have a larger interval of support. We know $s_u(j)$ is finite for a numeric representation of a wavelet function, since wavelets are defined to only be nonzero on a finite interval. Then we can define $\sigma_j = \frac{s_u(j)}{s_h(j)}$ for a level j wavelet (a plot describing this variable is available in Figure 2.5). Then we can see $M_{j,j} \in \mathbb{R}^{2\sigma_j}$ - a diagram with an example of this is available in Figure 2.6. We can also see $B_{j,j}, E_{j,j} \in \mathbb{R}^{\sigma_j \times \sigma_j}$ because at most only σ_j wavelets will not be fully supported at the beginning or end of the signal. An example for this is available in Figure 2.7.

All nonzero entries of $W_{j,j}^{(\tau)}$ outside of the first and last submatrices of size $\sigma_j \times \sigma_j$ are confined to a small $(2\sigma_j)$ number of diagonals, and repeat because of the Toeplitz property. Thus we can compute the portion of the total cross-correlation derived from this component by calculating the cross-correlations of the coefficient vectors directly, $(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(t)$ for a

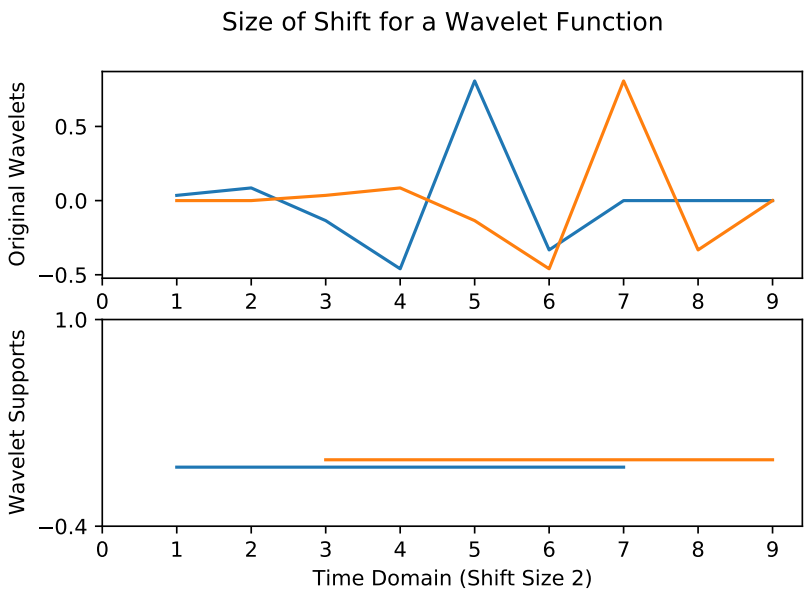

Figure 2.5: Plot of two Daubechies wavelets 1 shift apart, and their supports. The support of this wavelet is length 6, while its shift size is 2. Thus σ_j for this wavelet function is 3. Shifting this wavelet by 1 moves it two indices in the time domain.

range of time-lags t (distinct from the time-lag of the signal cross-correlation, τ), and then multiplying these inner products by entries of $M_{j,j}$. Since $M_{j,j}$ consists of at most $2\sigma_j$ entries (the maximum number of nonzero diagonals in $W_{j,j}^{(\tau)}$), we need to compute $(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(t)$ for at most $2\sigma_j$ different values of t.

The only entries in $W_{j,j}^{(\tau)}$ different from those in $M_{j,j}$ occur at the beginning and end sub-matrices, $B_{j,j}$ and $E_{j,j}$. These are due to boundary wavelet functions, of which at most σ_j can exist, so we only calculate their cross-correlations based on a subinterval of the wavelet function. To include these in the cross-correlation, we simply calculate $\vec{x}_j^{(r)}[:\sigma_j]^T B_{j,j} \vec{x}_j^{(s)}[:\sigma_j]$ and $\vec{x}_j^{(r)}[-\sigma_j:]^T E_{j,j} \vec{x}_j^{(s)}[-\sigma_j:]$, following Python indexing notation to refer to the first σ_j or last σ_j entries of vectors. σ_j is usually very small (≤ 5), so these multiplications are not computationally intensive. In addition, σ_j has no dependence on the length of our signals d^r and d^s.

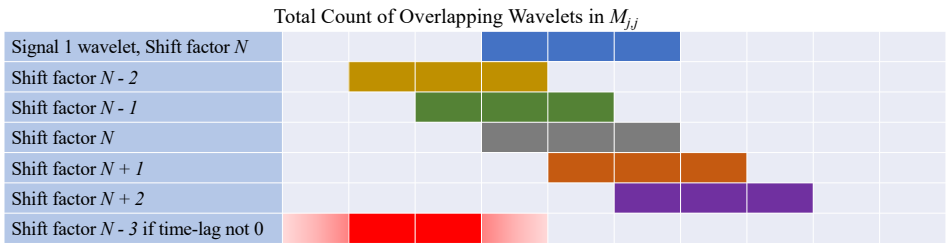

Figure 2.6: Diagram of wavelets from Figure 2.5 that overlap with a particular shift factor N. Since $\sigma_j = 3$, we know there are at most six other wavelets of this level that have overlapping support. If the time-lag is 0, then there are only five overlapping wavelets.

Since this representation of $W_{j,j}^{(\tau)}$ includes all of its nonzero entries, it calculates $\vec{x}_j^{(r)T} W_{j,j}^{(\tau)} \vec{x}_j^{(s)}$ accurately compared to a standard signal domain cross-correlation computation. Moreover, it sharply reduces the total number of required floating-point operations compared to a naive dense $W_{j,j}^{(\tau)}$ computation.

For the case where $j > k$, we no longer have true Toeplitz and banded properties, but we still have similar qualities: we know each row of $W_{j,k}^{(\tau)}$ has a small number of nonzero entries only located at certain indices, and these nonzero entries repeat in a cyclic, Toeplitz-like pattern similar to diagonals in a square matrix: $W_{j,k}^{(\tau)}[n, m] = W_{j,k}^{(\tau)}[n + \ell, m + 2^{j-k}\ell]$. Thus we can still store all unique cross-correlations of most of the wavelets with just one small vector $M_{j,k}$, consisting of these nonzero entries that repeat in the Toeplitz-like pattern. A diagram illustrating this is available in Figure 2.8.

The number of nonzero entries for $M_{j,k}$ is a function of the number of shifts for level k that overlap with the support of level j. Since the shift size of level j is $2^{(j-k)} \times$ (shift size of level k), this gives us

$$\dim(M_{j,k}) = \frac{s_u(j)}{\frac{s_h(j)}{2^{(j-k)}}} + \sigma_k = 2^{(j-k)}\sigma_j + \sigma_k. \tag{2.14}$$

A plot visualizing equation (2.14) is available in Figure 2.9. For $B_{j,k}$ and $E_{j,k}$, we can use

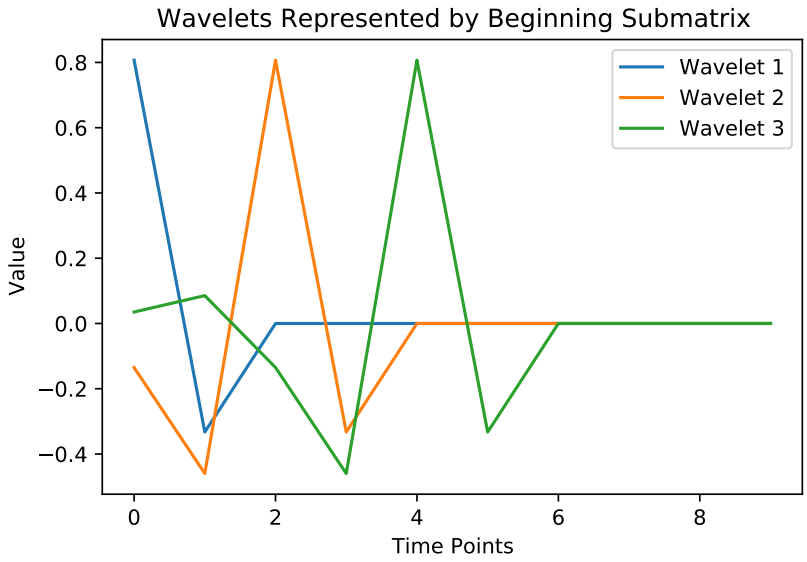

Figure 2.7: Plot of wavelet functions of the same scale at the beginning of a time series signal. Notice that the support of these functions does not fully overlap with the signal beginning at time point 0 until Wavelet 3 ($\sigma_j = 3$). Since these wavelets are only partially represented, their cross-correlation values are unique from those in $M_{j,j}$. Therefore a submatrix of size $\sigma_j \times \sigma_j$ is necessary to represent the cross-correlations of the first σ_j shifts for each signal at level j.

rectangular submatrices of size $\sigma_j \times \sigma_k$ to obtain all cross-correlations involving wavelets not fully represented on the signal. This is similar to the same-level case. Computations for $B_{j,k}$ and $E_{j,k}$ are also virtually the same as in the $j = k$ case. For computing the interior component with $M_{j,k}$, we now calculate inner products of $\vec{x}_j^{(r)}$ with slices of $\vec{x}_k^{(s)}$ taking every 2^{j-k}-th entry. This is because the entries of $W_{j,k}^{(\tau)}$ represented by $M_{j,k}$ are no longer on proper diagonals, but still have a staggered repeating pattern along each row. Once we get these inner products, we then multiply them by the entries of $M_{j,k}$ and add them to our overall cross-correlation result. Algorithm 1 outlines the process for computing the product $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$ for $j \geq k$ (a generalization of both cases described). As we calculate these level-specific cross-correlations, we can sum their values as in Equation (2.13) to get the final, overall cross-correlation of our signals d^r and d^s.

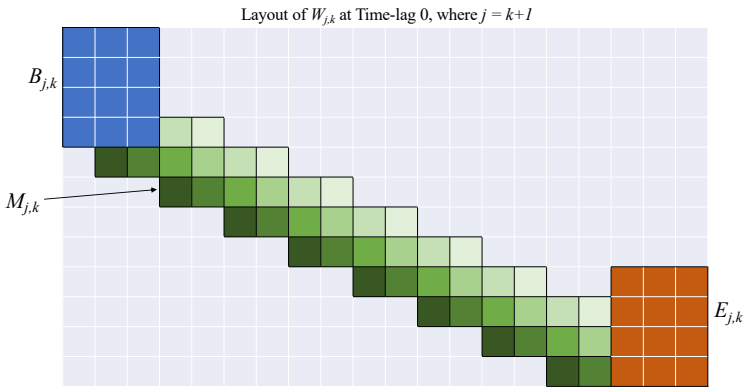

Figure 2.8: Layout of a mixed-level block matrix $W_{j,k}$. Interior entries ($M_{j,k}$, colored green) repeat, so we only need to store one row. Entries for $B_{j,k}$ and $E_{j,k}$ are unique. Note that we no longer have a true Toeplitz or banded property in $M_{j,k}$, but we still have row entries repeat at an offset.

2.3.2 Computing Across Multiple Time-lags

In most use cases, it is important to compute cross-correlations across a range of time-lags, rather than computing them for just one time-lag individually. The naive approach is to perform the entire cross-correlation computation for each required time-lag τ separately. While this method yields a correct answer, it leaves much room for improved performance. Assuming we wish to calculate $(d^r \star d^s)(\tau)$ for a continuous range of time-lags τ, we can take advantage of some redundancies in Algorithm 1 to significantly improve the runtime.

Consider a particular set of correlation block matrices $W_{j,k}^{(\tau)}$ for $\tau \in \{i\}_{i=0}^{2^k-1}$. We can store the entries of $M_{j,k}^{(0)}, \ldots, M_{j,k}^{(2^k-1)}$ together into a matrix $M_{j,k}^{(:)}$, again, using Python notation to describe this. We can then calculate the inner products of $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$ that are multiplied by $M_{j,k}$, i.e. the entries of the array $\vec{v}$ in Algorithm 1, once, then calculate the block matrix vector product

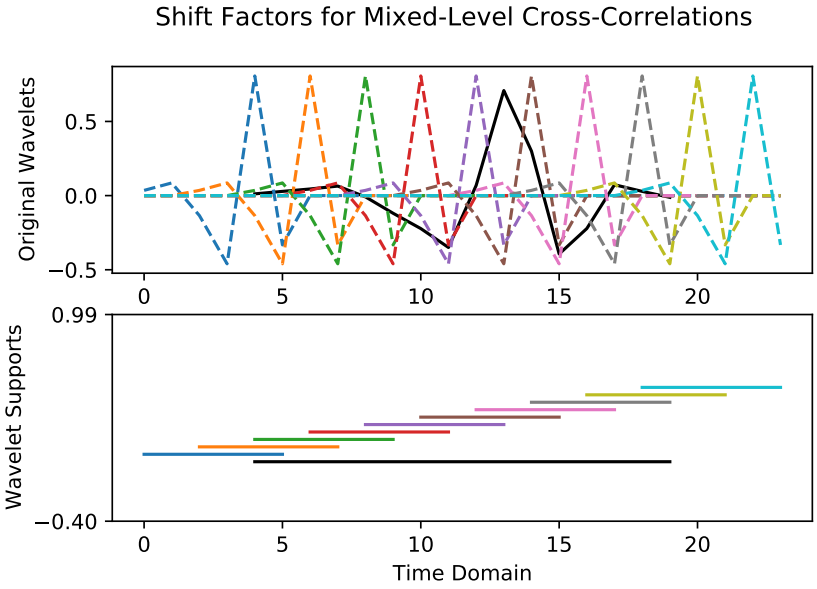

Figure 2.9: Plot of wavelet functions at two different scaling factors, showing how the smaller scale wavelet slides along the larger wavelet at each shift. In this particular case, the larger wavelet has a support of length 16 while the smaller one has a shift factor size of 2, so we have 10 shift factors that overlap with the larger wavelet here. This is consistent with Equation (2.14).

$$M_{j,k}^{(:)}\vec{v} = \begin{bmatrix} M_{j,k}^{(0)} \\ \vdots \\ M_{j,k}^{(2^k-1)} \end{bmatrix} \vec{v}. \tag{2.15}$$

This approach avoids redundantly computing the entries of $\vec{v}$, and allows us to take advantage of existing vectorized array operation libraries. If we are calculating cross-correlations for time-lags $\tau = 0, 1, \ldots, T$, then we only need to compute $\vec{v}$ for $\frac{T}{2^k}$ different shifts of $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$, and replace $T \cdot \dim(\vec{v})$ total dot products with $\frac{T}{2^k} \cdot \dim(\vec{v})$ matrix-vector multiplications (since $\vec{v}$ is a vector, its dimension is a scalar). We can employ a similar technique to improve the computational efficiency when calculating the beginning and end components: store $B_{j,k}^{(0)}, \ldots B_{j,k}^{(2^k-1)}$ together as a three-dimensional tensor $B_{j,k}^{(:)}$, and replace T total vector-matrix-vector operations $\vec{x}_j^{(r)}[:\sigma_j]^T B_{j,k} \vec{x}_k^{(s)}[:\sigma_k]$ with $\frac{T}{2^k}$ vector-tensor-vector opera-

Algorithm 1 Wavelet Cross-correlation: Single Block, $j \geq k$

$\vec{x}_j^{(r)}, \vec{x}_k^{(s)}$ = wavelet coefficients for levels j, k of the DWT for signals r, s.
τ = time-lag
$d = \text{floor}\left(\frac{\tau}{2^k}\right)$
$s_u(\ell)$ = number of nonzero values in level ℓ wavelet function
$s_h(\ell)$ = size of a single shift factor for a level ℓ wavelet function
$\sigma_j = s_u(j)/s_h(j)$
$\sigma_k = s_u(k)/s_h(k)$
$\alpha = 2^{(j-k)}\sigma_j + \sigma_k$
$W_{j,k}$ = correlation matrix mixing levels j and k at time-lag $\tau - 2^k d$
$B_{j,k}, E_{j,k}$ = beginning and end submatrices of $W_{j,k}$, each size $\sigma_j \times \sigma_k$
$M_{j,k}$ = interior values of $W_{j,k}$, size α
$\vec{v} = 0 \in \mathbb{R}^\alpha$
for i in $\{1, \ldots, \alpha\}$ **do**
    $\vec{v}[i] = \vec{x}_j^{(r)} \cdot \vec{x}_k^{(s)}[d + 2^{j-k} + i :: 2^{j-k}]$
end for
$\text{Xcorr} = \vec{v} \cdot M_{j,k} + \vec{x}_j^{(r)}[: \sigma_j]^T B_{j,k} \vec{x}_k^{(s)}[d : d + \sigma_k]$
$\text{Xcorr} = \text{Xcorr} + \vec{x}_j^{(r)}[-2^{k-j}d - \sigma_j : -2^{k-j}d]^T E_{j,k} \vec{x}_k^{(s)}[-\sigma_k :]$
return Xcorr

tions, using the same process for $E_{j,k}$.

Common use cases for cross-correlation in geophysical time series data require computation for a large range of time-lags, often on the order of 10^2 or 10^3. Even when reusing values of $\vec{v}$ in a matrix-vector multiplication as in (2.15), we still need to compute $\vec{v}$ for a large range of shift factors. Let $\vec{v}^{(d)}$ denote $\vec{v}$ computed for $d = \text{floor}\left(\frac{\tau}{2^j}\right)$, as described in Algorithm 1. Then, given time-lags $\tau \in [0, T]$, with $d' = \text{ceil}\left(\frac{T}{2^j}\right)$, and the block matrix operation $\vec{x}_j^{(r)T} W_{j,j}^{(\tau)} \vec{x}_j^{(s)}$, we need to compute

$$
\begin{aligned}
V &= [\vec{v}^{(0)}, \vec{v}^{(1)}, \ldots, \vec{v}^{\text{ceil}\left(\frac{T}{2^j}\right)}] \\
&= \begin{pmatrix}
(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(\sigma_j) & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(\sigma_j + 1) & \ldots & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(d' + \sigma_j) \\
(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(\sigma_j - 1) & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(\sigma_j) & \ldots & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(d' + \sigma_j - 1) \\
\vdots & & \ddots & \vdots \\
(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(-\sigma_j) & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(-\sigma_j + 1) & \ldots & (\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(d' - \sigma_j)
\end{pmatrix}.
\end{aligned}
\tag{2.16}
$$

This formulation helps reveal a useful property: due to the time-lags t for $(\vec{x}_j^{(r)} \star \vec{x}_j^{(s)})(t)$ required to compute the individual entries of $\vec{v}$ used in computing $(d^r \star d^s)(\tau)$, $\vec{v}^{(d)}[i] \approx \vec{v}^{(d+1)}[i+1]$. Therefore, we only need to calculate the sliding inner products of $\vec{x}_j^{(r)}$ and $\vec{x}_j^{(s)}$ at time-lags corresponding to the first row and first column of V, and can then use these entries to help compute the rest of V. We do not have an exact equality because the relative start and endpoints of each subdiagonal shift by 1, so we do need to apply additional operations on each subsequent row of V. However, by reusing the already calculated values from other time lags, this approach still sharply reduces the number of FLOPs needed to compute V, especially when combined with fast Fourier transform algorithms to compute the first row. We can apply similar redundancy rules to V in the $j > k$ case as well.

These modifications for computing $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$ across a range of time-lags $\tau \in [0, T]$ significantly reduce the overall runtime of the wavelet-domain cross-correlation method. An overview of this workflow is available in Algorithm 2. Once we calculate the component of the cross-correlation corresponding to levels $j \times k$ for our desired range of time-lags, we can then add them together, as shown in Equation (2.13).

Algorithm 2 Wavelet Cross-correlation: Across Multiple Time-lags

$\vec{x}_j^{(r)}, \vec{x}_k^{(s)}$ = wavelet coefficients for levels j, k of the DWT for signals r, s.
T = max time-lag
$d' = \text{ceil}\left(\frac{T}{2^k}\right)$
$s_u(\ell)$ = number of nonzero values in level ℓ wavelet function
$s_h(\ell)$ = size of a single shift factor for a level ℓ wavelet function
$\sigma_j = s_u(j)/s_h(j)$
$\sigma_k = s_u(k)/s_h(k)$
$\alpha = 2^{(j-k)}\sigma_j + \sigma_k$
$W_{j,k}^{(:)}$ = correlation tensor mixing levels j and k at time-lags $\tau = 0, 1, \ldots, 2^k - 1$
$B_{j,k}^{(:)}, E_{j,k}^{(:)}$ = beginning and end subtensors of $W_{j,k}^{(:)}$, each size $\sigma_j \times \sigma_k \times 2^k$
$M_{j,k}^{(:)}$ = interior values of $W_{j,k}^{(:)}$, size $\alpha \times 2^k$
$V = 0 \in \mathbb{R}^{\alpha \times d'}$
for i in $\{1, \ldots, \alpha\}$ **do**
    for d in $\{0, \ldots, d'\}$ **do**
        $V[i, d] = \vec{x}_j^{(r)} \cdot \vec{x}_k^{(s)}[d + 2^{j-k} + i :: 2^{j-k}]$
    end for
end for
$X_k = [\vec{x}_k^{(s)}[: d], \vec{x}_k^{(s)}[1 : 1 + d], \ldots, \vec{x}_k^{(s)}[\sigma_k : \sigma_k + d]]$
$X_j = [\vec{x}_j^{(r)}[-\sigma_j :], \vec{x}_j^{(r)}[-1 - \sigma_j : -1], \ldots, \vec{x}_j^{(r)}[-2^{k-j}d - \sigma_j : -2^{k-j}d]]$
$\text{Xcorrs} = M_{j,k}^{(:)}V + \vec{x}_j^{(r)}[: \sigma_j]^T B_{j,k}^{(:)} X_k + X_j E_{j,k}^{(:)} \vec{x}_k^{(s)}[-\sigma_k :]$
return Xcorrs

2.3.3 Use of DWT Sparsity and Overall Efficiency

A major advantage of the wavelet-domain cross-correlation algorithm is the ability to utilize sparsity in compressed DWT coefficients. Since this algorithm does not require an inverse wavelet transform, it is possible to compute cross-correlations while using relatively small volumes of data. In addition, the computations for forming V are sliding inner products of the coefficient vectors $\vec{x}^{(r)}$ and $\vec{x}^{(s)}$, which can be easily implemented as sparse operations. Software implementations of this wavelet-domain cross-correlation algorithm have been written for both dense and sparse formats of wavelet coefficients for time-series data.

To evaluate the overall efficiency using asymptotic cost, consider the block matrix multiplication $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$. Let $N = \min\{\dim(\vec{x}^{(r)}), \dim(\vec{x}^{(s)})\} \approx \min\{\dim(d^r), \dim(d^s)\}$. From Algorithm 1 we know this consists of three main computations: the formation of $\vec{v}$ which is multiplied with $M_{j,k}$, and the vector-matrix-vector products using $B_{j,k}, E_{j,k}$. $B_{j,k}$ and $E_{j,k}$ are both small (dimensions $\sigma_j \times \sigma_k$, where $\sigma_j, \sigma_k \approx 5$) matrices that are not dependent on N, so their components on the cross-correlation have a negligible impact on the computational cost as N increases. The formation of $\vec{v}$ requires the computation of $2^{(j-k)}\sigma_j + \sigma_k$ inner products of $\vec{x}_j^{(r)}$ with $\vec{x}_k^{(s)}$ or a slice of $\vec{x}_k^{(s)}$. We know $\min\{\dim(\vec{x}_j^{(r)}), \dim(\vec{x}_j^{(s)})\} \approx \frac{N}{2^j}$. Therefore, the asymptotic cost of computing $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$ (particularly $\vec{v}$) is roughly

$$\begin{aligned} \text{Number of inner products} \times \text{dim of vector} \times 2 &= (2^{(j-k)}\sigma_j + \sigma_k) \cdot \frac{N}{2^j} \cdot 2 \\ &= 2(2^{-k}\sigma_j + 2^{-j}\sigma_k)N. \end{aligned} \tag{2.17}$$

Let $\beta_{j,k} = 2(2^{-k}\sigma_j + 2^{-j}\sigma_k)$, i.e. the coefficient in Equation (2.17). For the approximation level $J+1$, this coefficient is instead $4(2^{-k}\sigma_{J+1} + 2^{-J-1}\sigma_k)$, since it has the same dimensions as

the last detail level. By applying this into Equation (2.13), we can get the overall asymptotic cost for computing the cross-correlation at one time-lag:

$$\begin{aligned} O\left(\vec{x}^{(r)} W^{(\tau)} \vec{x}^{(s)}\right) &\approx \sum_{j=1}^{J+1} \left[\beta_{j,j} N + \sum_{k=1}^{j-1} \left(\beta_{j,k} N + \beta_{j,k} N \right) \right] \\ &= \left(\sum_{j=1}^{J+1} \left[\beta_{j,j} + 2 \sum_{k=1}^{j-1} \beta_{j,k} \right] \right) N. \end{aligned} \tag{2.18}$$

Thus the asymptotic cost is $O(N)$ for a fixed DWT. However, the coefficient of our cost for a single time-lag is significantly higher than in a direct cross-correlation computation (where is it roughly 2, equivalent to an inner product). In addition, using a DWT with more levels may increase this coefficient, since it introduces more components $\beta_{j,k}$ to the summation.

There are a few mitigating factors that can reduce this asymptotic cost. If we're computing values of multiple time-lags concurrently, then we can compute the matrix V as previously described and shown in Algorithm 2, reducing the number of inner product computations required and thus reducing the coefficient associated with computing each block matrix product $\vec{x}_j^{(r)T} W_{j,k}^{(\tau)} \vec{x}_k^{(s)}$ across a range of τ. Importantly, taking advantage of sparsity in our computations of $\vec{v}$ or V (if we are using a sparse wavelet representation) can significantly reduce the number of required arithmetic operations even further.

There can be a trade-off when determining the number of levels to use in a DWT for wavelet compression. More levels in the DWT introduces more components $\beta_{j,k}$ to the coefficient of our asymptotic cost, potentially increasing it. However, higher level scaling and wavelet functions can often represent data more accurately after thresholding than lower level functions, which allows us to use higher compression factors and thus gain more benefit from sparsity in the computation of $\vec{v}$ or V. Some of the examples in Section 2.5 highlight this.

The ideal number of levels to use for accurate compression and optimal performance in computing cross-correlations is likely dependent on both the choice of wavelet family and qualities of the data itself.

2.4 Theoretical Error Analysis

2.4.1 Pointwise Error Bound for a Single Time-lag

It is useful to develop theoretical error bounds on the utilization of wavelet compression techniques, such as thresholding, when computing cross-correlations. For calculating effective error bounds, it is helpful to use the theory we have developed in previous sections. For any two real time-series signals d^r and d^s and a DWT with a given wavelet family and level J, we know

$$(d^r \star d^s)(\tau) = \vec{x}^{(r)T} W^{(\tau)} \vec{x}^{(s)}, \tag{2.19}$$

where $W^{(\tau)}$ is the correlation matrix for the given discrete wavelet transform (thus $W^{(\tau)}$ has all the properties established in section 2.2), and $\vec{x}^{(r)}, \vec{x}^{(s)}$ are vectors storing the wavelet coefficients of d^r and d^s for this particular DWT. Let $\hat{\vec{x}}^{(r)}$ be the sparse coefficient vector for $\hat{d}^r$, the compressed version of the signal, and let $\tilde{\vec{x}}^{(r)} = \vec{x}^{(r)} - \hat{\vec{x}}^{(r)}$. Suppose we are using a compression-based thresholding approach: let $c > 1$ be the compression factor, so $\hat{\vec{x}}$ stores the largest $\frac{1}{c}$ entries in magnitude, while $\tilde{\vec{x}}$ stores the remaining $1 - \frac{1}{c} = \frac{c-1}{c}$ entries. We then know

$$\begin{aligned}
\left|(d^r \star d^s)(\tau) - (\hat{d}^r \star \hat{d}^s)(\tau)\right| &= \left|\vec{x}^{(r)T} W^{(\tau)} \vec{x}^{(s)} - \hat{\vec{x}}^{(r)T} W^{(\tau)} \hat{\vec{x}}^{(s)}\right| \\
&= \left|(\tilde{\vec{x}}^{(r)T} + \hat{\vec{x}}^{(r)T}) W^{(\tau)} (\tilde{\vec{x}}^{(s)} + \hat{\vec{x}}^{(s)}) - \hat{\vec{x}}^{(r)T} W^{(\tau)} \hat{\vec{x}}^{(s)}\right| \\
&= \left|\tilde{\vec{x}}^{(r)T} W^{(\tau)} \tilde{\vec{x}}^{(s)} + \tilde{\vec{x}}^{(r)T} W^{(\tau)} \hat{\vec{x}}^{(s)} + \hat{\vec{x}}^{(r)T} W^{(\tau)} \tilde{\vec{x}}^{(s)}\right| \\
&\leq \left|\tilde{\vec{x}}^{(r)T} W^{(\tau)} \tilde{\vec{x}}^{(s)}\right| + \left|\tilde{\vec{x}}^{(r)T} W^{(\tau)} \hat{\vec{x}}^{(s)}\right| + \left|\hat{\vec{x}}^{(r)T} W^{(\tau)} \tilde{\vec{x}}^{(s)}\right| \\
&\leq \|\tilde{\vec{x}}^{(r)}\|_2 \|W^{(\tau)} \tilde{\vec{x}}^{(s)}\|_2 + \|\tilde{\vec{x}}^{(r)}\|_2 \|W^{(\tau)} \hat{\vec{x}}^{(s)}\|_2 + \|\hat{\vec{x}}^{(r)}\|_2 \|W^{(\tau)} \tilde{\vec{x}}^{(s)}\|_2 \\
&\leq \|W^{(\tau)}\|_2 \left(\|\tilde{\vec{x}}^{(r)}\|_2 \|\tilde{\vec{x}}^{(s)}\|_2 + \|\tilde{\vec{x}}^{(r)}\|_2 \|\hat{\vec{x}}^{(s)}\|_2 + \|\hat{\vec{x}}^{(r)}\|_2 \|\tilde{\vec{x}}^{(s)}\|_2\right).
\end{aligned} \tag{2.20}$$

From here, we wish to bound the coefficient vector norms inside the parentheses. Let N be the length of $\vec{x}^{(r)}, \vec{x}^{(s)}$, $B^r = \max\{|x_i^{(r)}|\}_{i=1}^N$, and $T^r = \max\{|\tilde{x}_i^{(r)}|\}_{i=1}^N$ (with B^s and T^s defined similarly). Then we know:

$$\|\hat{\vec{x}}^{(r)}\|_2 = \left[\sum_{\substack{i=1 \\ |x_i^{(r)}|>T^r}}^{N} \left|x_i^{(r)}\right|^2\right]^{1/2} \leq \left[N \frac{1}{c} \left(B^r\right)^2\right]^{1/2} = \frac{\sqrt{N}}{\sqrt{c}} B^r \tag{2.21}$$

$$\|\tilde{\vec{x}}^{(r)}\|_2 = \left[\sum_{\substack{i=1 \\ |x_i^{(r)}|\leq T^r}}^{N} \left|x_i^{(r)}\right|^2\right]^{1/2} \leq \left[N \frac{c-1}{c} \left(T^r\right)^2\right]^{1/2} = \sqrt{N} \frac{\sqrt{c-1}}{\sqrt{c}} T^r \tag{2.22}$$

Substituting inequalities (2.21) and (2.22) into (2.20), we then get

$$\begin{aligned}
&\left|(d^r \star d^s)(\tau) - (\hat{d}^r \star \hat{d}^s)(\tau)\right| \\
\leq\ & \|W^{(\tau)}\|_2 \left(N\frac{c-1}{c}T^rT^s + N\frac{\sqrt{c-1}}{c}T^rB^s + N\frac{\sqrt{c-1}}{c}B^rT^s \right) \\
=\ & \|W^{(\tau)}\|_2 \frac{N\sqrt{c-1}}{c}\left(\sqrt{c-1}T^rT^s + T^rB^s + B^rT^s\right). \qquad (2.23)
\end{aligned}$$

This error bound scales linearly with the dimensions of the time series signals d^r and d^s, which $\approx N$. It also avoids dependency on knowing the specific indices of thresholded values in $\vec{x}^{(r)}$ and $\vec{x}^{(s)}$. However, our correlation matrix varies in dimensions based on the dimensions of the time series signals, since these determine the number of shift factors per level. Therefore, it is important to prove that $\|W^{(\tau)}\|_2$ is uniformally bounded above, regardless of the shift factor count N. Let J be the level, or total number of scale factors, in this DWT (we denote the approximation level as $J+1$). Then we know, using the triangle inequality, the property $\left\| \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} \right\|_2 = \max\left(\|A\|_2, \|B\|_2\right)$, and our property $W^{(\tau)}_{j,k} = W^{(-\tau)}_{k,j}$, that

$$\begin{aligned}
\|W^{(\tau)}\|_2 &= \left\| \begin{bmatrix} W^{(\tau)}_{1,1} & & W^{(\tau)}_{1,J} \\ \vdots & \ddots & \\ W^{(\tau)}_{J,1} & & W^{(\tau)}_{J+1,J+1} \end{bmatrix} \right\|_2 \\
&\leq \left\| \begin{bmatrix} W^{(\tau)}_{1,1} & & 0 \\ & \ddots & \\ 0 & & W^{(\tau)}_{J+1,J+1} \end{bmatrix} \right\|_2 + \sum_{j=1}^{J+1}\sum_{\substack{k=1 \\ k\neq j}}^{J+1} \|W^{(\tau)}_{j,k}\|_2 \\
&= \max_j \left(\|W^{(\tau)}_{j,j}\|_2 \right) + \sum_{j=1}^{J+1}\sum_{k=1}^{j-1} \left(\|W^{(\tau)}_{j,k}\|_2 + \|W^{(-\tau)}_{j,k}\|_2 \right). \qquad (2.24)
\end{aligned}$$

From here we can bound each of these norms separately. First consider a single same-level correlation block matrix $W_{j,j}^{(\tau)}$. We can write this in the form

$$\begin{aligned}
\|W_{j,j}^{(\tau)}\|_2 &= \left\| \begin{bmatrix} B_{j,j}^{(\tau)} & & 0 \\ & M_{j,j}^{(\tau)} & \\ 0 & & E_{j,j}^{(\tau)} \end{bmatrix} \right\|_2 \\
&\leq \left\| \begin{bmatrix} 0 & & \\ & M_{j,j}^{(\tau)} & \\ & & 0 \end{bmatrix} \right\|_2 + \left\| \begin{bmatrix} B_{j,j}^{(\tau)} & & 0 \\ & 0 & \\ 0 & & E_{j,j}^{(\tau)} \end{bmatrix} \right\|_2 \\
&= \|M_{j,j}^{(\tau)}\|_2 + \max\left(\|B_{j,j}^{(\tau)}\|_2, \|E_{j,j}^{(\tau)}\|_2 \right). \qquad (2.25)
\end{aligned}$$

Since $M_{j,j}^{(\tau)}$ is a banded, Toeplitz matrix with $2\sigma_j$ nonzero diagonals, we can bound its 2-norm by $\sum_{i=-\sigma_j}^{\sigma_j} \left| \left[M_{j,j}^{(\tau)} \right]_{\sigma_j, \sigma_j + i} \right|$. Note that σ_j is small, typically ≤ 5 for most wavelet functions and levels, and each of the entries of $M_{j,j}^{(\tau)}$ are ≤ 1 in magnitude. Meanwhile, $B_{j,j}^{(\tau)}$ and $E_{j,j}^{(\tau)}$ are each small matrices of size $\sigma_j \times \sigma_j$, so their 2-norms can be easily precomputed in advance. We expect these matrices to have reasonably small norms as well, since each of their entries are ≤ 1 in magnitude (because our wavelet and scaling functions are normalized, so by the Cauchy-Schwarz inequality $(\theta^{(g)} \star \theta^{(h)})(\tau) \leq 1$). Combining these two components, we get a reasonable bound:

$$\max_j \left(\|W_{j,j}^{(\tau)}\|_2 \right) = \max_j \left(\sum_{i=-\sigma_j}^{\sigma_j} \left| \left[M_{j,j}^{(\tau)} \right]_{\sigma_j, \sigma_j + i} \right| + \max\left(\|B_{j,j}^{(\tau)}\|_2, \|E_{j,j}^{(\tau)}\|_2 \right) \right), \qquad (2.26)$$

where the indexing outside of the square brackets refers to the entry indexing for these matrices. Note that this bound has no dependence on the dimensions of the original time

series d^r and d^s. It is only dependent on the choice of wavelet function and the number of levels. Thus the bound can be determined prior to data collection, based on the choice of DWT parameters. Now consider the mixed-level correlation block matrices $W_{j,k}^{(\tau)}$, with $j > k$. We can bound their 2-norms in a similar way to the same-level case:

$$\|W_{j,k}^{(\tau)}\|_2 \leq \|M_{j,k}^{(\tau)}\|_2 + \left\| \begin{bmatrix} B_{j,k}^{(\tau)} & 0 \\ 0 & E_{j,k}^{(\tau)} \end{bmatrix} \right\|_2 = \|M_{j,k}^{(\tau)}\|_2 + \max\left(\|B_{j,k}^{(\tau)}\|_2, \|E_{j,k}^{(\tau)}\|_2 \right). \tag{2.27}$$

Computing $\|B_{j,k}^{(\tau)}\|_2$, $\|E_{j,k}^{(\tau)}\|_2$ is straightforward. Bounding $\|M_{j,k}^{(\tau)}\|_2$, however, is somewhat more complicated than bounding $\|M_{j,j}^{(\tau)}\|_2$, since we no longer have the banded or Toeplitz properties. However, we do know only a small, constant number of entries in each row can be nonzero (equivalent to $2^{j-k}\sigma_j + \sigma_k$), and these row entries repeat in a Toeplitz-like pattern. Let $\tilde{M}_{j,k}^{(\tau)}$ be the matrix $M_{j,k}^{(\tau)}$ padded with additional rows so that it is Toeplitz. In particular, after every row of $M_{j,k}^{(\tau)}$ we insert $2^{j-k} - 1$ additional rows of the same nonzero entries, where the i'-th row is shifted right i' times ($i' \in \{1, \ldots 2^{j-k} - 1\}$). Denote this $[M_{j,k}^{(\tau)}]_{i_{i'},:}$ for the i-th row of $M_{j,k}^{(\tau)}$. Let m, n denote the number of rows and columns of $M_{j,k}^{(\tau)}$. Then we have

$$\tilde{M}_{j,k}^{(\tau)} = \begin{bmatrix} [M_{j,k}^{(\tau)}]_{1,:} \\ [M_{j,k}^{(\tau)}]_{1_{i'},:} \\ [M_{j,k}^{(\tau)}]_{2,:} \\ \vdots \\ [M_{j,k}^{(\tau)}]_{m_{i'},:} \end{bmatrix}. \tag{2.28}$$

We can show for any $\vec{x} \in \mathbb{C}^n$ that

$$\begin{aligned}
\|\tilde{M}_{j,k}^{(\tau)}\vec{x}\|_2^2 &= \left\| \begin{bmatrix} [M_{j,k}^{(\tau)}]_{1,:} \\ [M_{j,k}^{(\tau)}]_{1_{i'},:} \\ [M_{j,k}^{(\tau)}]_{2,:} \\ \vdots \\ [M_{j,k}^{(\tau)}]_{m_{i'},:} \end{bmatrix} \vec{x} \right\|_2^2 \\
&= |[M_{j,k}^{(\tau)}]_{1,:} \cdot \vec{x}|^2 + \|[M_{j,k}^{(\tau)}]_{1_{i'},:}\vec{x}\|^2 + \cdots + |[M_{j,k}^{(\tau)}]_{2,:} \cdot \vec{x}|^2 + \|[M_{j,k}^{(\tau)}]_{2_{i'},:}\vec{x}\|^2 + \dots \\
&\geq |[M_{j,k}^{(\tau)}]_{1,:} \cdot \vec{x}|^2 + |[M_{j,k}^{(\tau)}]_{2,:} \cdot \vec{x}|^2 + \cdots = \|M_{j,k}^{(\tau)}\vec{x}\|_2^2; \qquad (2.29)
\end{aligned}$$

note that $[M_{j,k}^{(\tau)}]_{i,:}$ is a vector, so $|[M_{j,k}^{(\tau)}]_{i,:} \cdot \vec{x}|^2$ is a scalar. Therefore we know $\|M_{j,k}^{(\tau)}\|_2 \leq \|\tilde{M}_{j,k}^{(\tau)}\|_2$, and that $\|\tilde{M}_{j,k}^{(\tau)}\|_2$ is Toeplitz and banded, thus we can get a bound for $\|M_{j,k}^{(\tau)}\|_2$ based on its limited nonzero entries:

$$\|M_{j,k}^{(\tau)}\|_2 \leq \|\tilde{M}_{j,k}^{(\tau)}\|_2 = \sum_{i=1}^{2^{j-k}\sigma_j+\sigma_k} \left| \left[M_{j,k}^{(\tau)} \right]_i \right|. \qquad (2.30)$$

Combining these together, we now have:

$$\begin{aligned}
\|W^{(\tau)}\|_2 &\leq \max_j \left(\|W_{j,j}^{(\tau)}\|_2 \right) + \sum_{j=1}^{J+1} \sum_{k=1}^{j-1} \left(\|W_{j,k}^{(\tau)}\|_2 + \|W_{j,k}^{(-\tau)}\|_2 \right) \\
&\leq \max_j \left(\sum_{i=-\sigma_j}^{\sigma_j} \left| \left[M_{j,j}^{(\tau)} \right]_{\sigma_j, \sigma_j + i} \right| + \max \left(\|B_{j,j}^{(\tau)}\|_2, \|E_{j,j}^{(\tau)}\|_2 \right) \right) \\
&+ \sum_{j=1}^{J+1} \sum_{k=1}^{j-1} \sum_{i=1}^{2^{j-k}\sigma_j + \sigma_k} \left(\left| \left[M_{j,k}^{(\tau)} \right]_i \right| + \left| \left[M_{j,k}^{(-\tau)} \right]_i \right| \right) \\
&+ \sum_{j=1}^{J+1} \sum_{k=1}^{j-1} \left(\max \left(\|B_{j,k}^{(\tau)}\|_2, \|E_{j,k}^{(\tau)}\|_2 \right) + \max \left(\|B_{j,k}^{(-\tau)}\|_2, \|E_{j,k}^{(-\tau)}\|_2 \right) \right). \quad (2.31)
\end{aligned}$$

If we know in advance that we will compute cross-correlations on signals of a certain length, then we know how many shift factors will be present at each level and can precompute the norm of $W^{(\tau)}$ as an explicitly formed matrix, to develop a more rigorous bound on the error of thresholding. However, Equation (2.31) gives us a uniform bound for $W^{(\tau)}$ independent of the DWT shift factors, confirming the error of our compressed cross-correlation scales linearly with the length of our signals d^r, d^s. It can also be useful for creating more generalized cross-correlation error bounds.

Also note that for any correlation block matrix $W_{j,k}^{(\tau)}$, we only store and use the matrix entries for $\tau = 0, 1, \ldots, 2^k - 1$. For larger time-lags, we make use of the cyclic property regarding correlation matrix time-lags, shown in section 2.2, by applying a left or right shift operator, as shown in Equation (2.12). Since left and right shift operators do not increase operator norms, we know this bound on $\|W^{(\tau)}\|_2$ can be developed for any time-lag τ. In addition, we can also determine $\max\{W_{j,k}^{(i)}\}_{i=0}^{2^k-1}$ for each block matrix $W_{j,k}^{(\tau)}$ to develop a bound for $\|W^{(\tau)}\|_2$ independent of τ.

2.4.2 Bound on the Cross-Correlation 1-Norm

It is also useful to develop an error bound for cross-correlations across a continuous range of time-lags, as opposed to one time-lag individually. Consider the 1-norm of cross-correlations over a range of time-lags $[\tau_1, \tau_2]$:

$$\|(d^r \star d^s)\|_1 = \int_{\tau_1}^{\tau_2} |(d^r \star d^s)(t)|\, dt. \tag{2.32}$$

We can make use of our correlation matrix implementation, along with the Cauchy-Schwarz inequality and matrix norms, to see

$$\begin{aligned} \int_{\tau_1}^{\tau_2} |(d^r \star d^s)(t)|\, dt &= \int_{\tau_1}^{\tau_2} \left|\vec{x}^{(r)T} W^{(t)} \vec{x}^{(s)}\right| dt \\ &\leq \int_{\tau_1}^{\tau_2} \|\vec{x}^{(r)}\|_2 \|W^{(t)}\|_2 \|\vec{x}^{(s)}\|_2 dt \\ &= \|\vec{x}^{(r)}\|_2 \|\vec{x}^{(s)}\|_2 \int_{\tau_1}^{\tau_2} \|W^{(t)}\|_2 dt. \end{aligned} \tag{2.33}$$

2.4.3 Applying Error Bounds for Coefficient Vectors

We have developed a basic bound on the error of wavelet compressed cross-correlations via thresholding using the correlation matrix structure we developed in section 2.2, which scales linearly with the length of the signals being used. It is useful to use these bounds on some examples of real data to develop intuition for their accuracy. To test our proposed bounds for $\|\hat{\vec{x}}\|_2$ and $\|\tilde{\vec{x}}\|_2$, we have computed the 2-norms of time series vibration data from a multichannel sensor array, and their bounds using inequalities (2.21) and (2.22).

A plot of the norms of $\|\hat{\vec{x}}\|_2$ at compression factor 10 and their computed bounds via in-

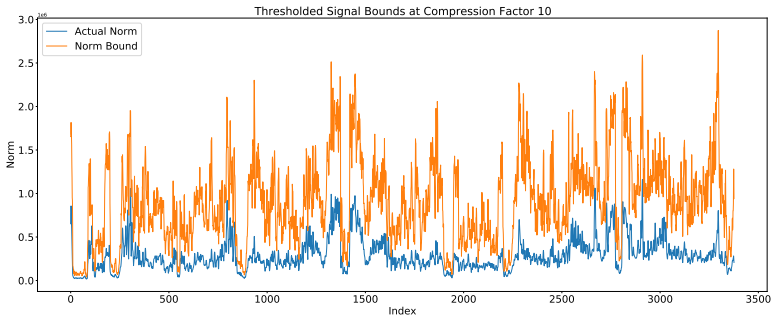

Figure 2.10: Comparison of values of $\|\hat{\vec{x}}\|_2$ and their bounds using (2.21).

equality (2.21) is shown in Figure 2.10. While correct, it is obvious the bounds for these norms are not particularly tight, especially at certain channels in the sensor array. A side by side comparison of the norms and bounds as shown here makes it difficult to asses the relative differences between the actual norms and upper bounds, however.

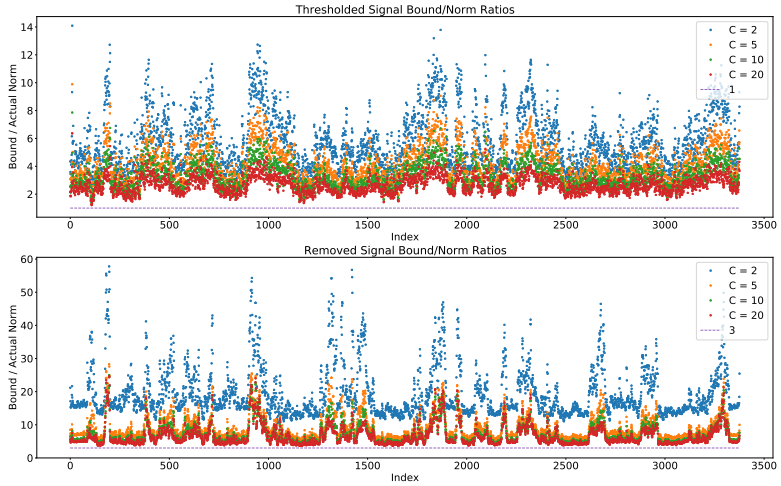

Figure 2.11: Ratios of $\|\hat{\vec{x}}\|_2$ and $\|\tilde{\vec{x}}\|_2$ compared to their computed upper bounds at compression factors C.

To better visualize the comparison of the vector norms of the thresholded signal $\hat{\vec{x}}$ and the removed portion $\tilde{\vec{x}}$, we have plotted the ratios of our bounds for these norms via (2.21), (2.22), and the actual norm values, shown in Figure 2.11. In general, we see that the bounds become tighter for larger compression factors in both cases. However, there is substantial variation in the ratios even within the same compression factor, suggesting that differences

in the variability and level of noise at individual sensors likely affects the overall accuracy. In addition, the bounds for $\tilde{\vec{x}}$ are significantly less tight than the bounds for $\hat{\vec{x}}$. This is likely because $\tilde{\vec{x}}$ consists of the majority of the DWT coefficients which possess a large amount of variance in their magnitudes, ranging from 0 to our thresholding value T, while $\hat{\vec{x}}$ consists of relatively few coefficients (especially at higher compression factors) that tend to be larger in magnitude. If thresholding values for multiple compression factors are stored for each time series signal as metadata, it may be possible to develop tighter bounds on $\tilde{\vec{x}}$, since we then no longer have to uniformally bound its entries with a constant maximum value T.

2.5 Test Cases

2.5.1 Basic Sine Curve

As a simple test to evaluate the correctness and usefulness of this wavelet-domain cross-correlation algorithm, we'll consider the autocorrelation of a basic sine curve. Consider the functions

$$f(t) = \begin{cases} 10\sin(2t), & x \in [0, 1000]; \\ 0, & \text{otherwise}; \end{cases} \quad g(t) = \begin{cases} 10\sin(2t), & x \in [0, 500]; \\ 0, & \text{otherwise}. \end{cases}$$

Note that f can be thought of as a time series signal defined over a compact interval, with g as a window of f. The cross-correlation $(f \star g)(\tau)$ resembles a cosine function for $\tau \in [0, 400]$ (this can be proven analytically). The plot in Figure 2.12 supports this, and displays the relative errors of the cross-correlation of our wavelet-domain algorithm using the Daubechies 3 DWT at levels 1, 2, and 3. We can see that our new cross-correlation method computes the expected results with reasonable accuracy, and that the largest relative errors occur where

the cross-correlation is very close to 0.

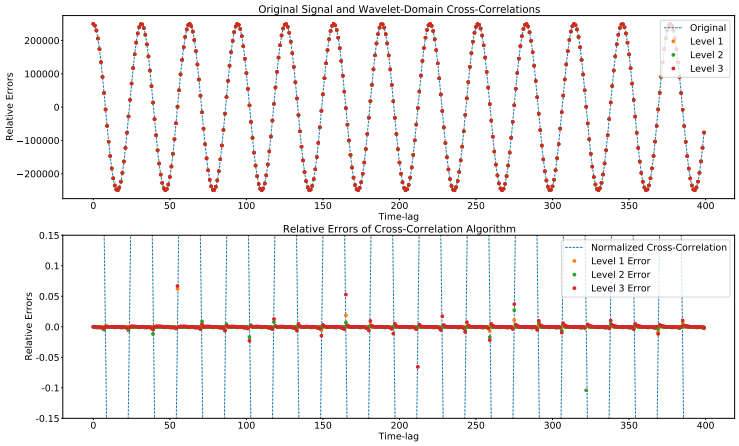

Figure 2.12: Plot of autocorrelations from a sine curve, computed from the original signal and from wavelet coefficients using our wavelet-domain cross-correlation algorithm. Note that the relative errors of the new method are small, and most significant where the original cross-correlation is close to 0.

The main advantage to the wavelet-domain cross-correlation algorithm is the ability to operate on a subset of the DWT coefficients, without the need to reconstruct the data with an inverse wavelet transform. We expect some loss of accuracy when doing this, however. Consider a form of compression where only the approximation coefficients, corresponding to the scaling function in the DWT, are preserved. Figure 2.13 is a plot of $(f \star g)(\tau)$ using only the approximation coefficients of the Daubechies 3 DWT, at levels 1, 2, and 3. We can see that the approximations at levels 1 and 2 are virtually identical to the original cross-correlation, while the approximation at level 3 only noticeably differs at the peaks and troughs of the cosine-like oscillations. However, approximation coefficients make up roughly a $\frac{1}{2^J}$ sized portion of the coefficients in a level J DWT. Therefore, the level 3 approximation-only cross-correlation requires about $\frac{1}{8}$ as much data to be stored as the traditional method.

Another common form of compression is thresholding. In Figure 2.14, $(f \star g)(\tau)$ computed using only the largest 10% of Daubechies 3 DWT coefficients in magnitude is plotted, at levels 1, 2, and 3. While the level 1 and 2 approximation coefficients more accurately

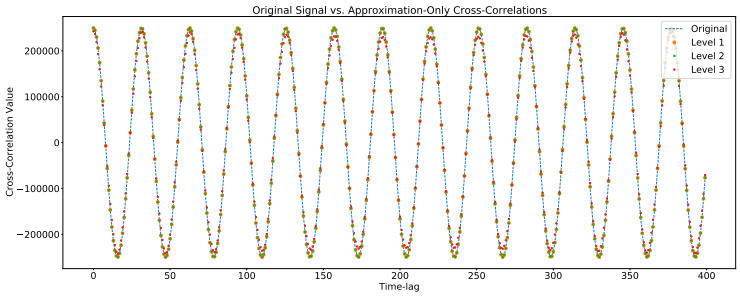

Figure 2.13: Plot of wavelet-domain cross-correlations computed using only the approximation coefficients at levels 1, 2, and 3. The number of stored approximation coefficients is roughly halved for each level.

preserve the cross-correlation values, the level 3 approximation does it with significantly fewer coefficients, allowing it to better preserve the values at a constant compression factor.

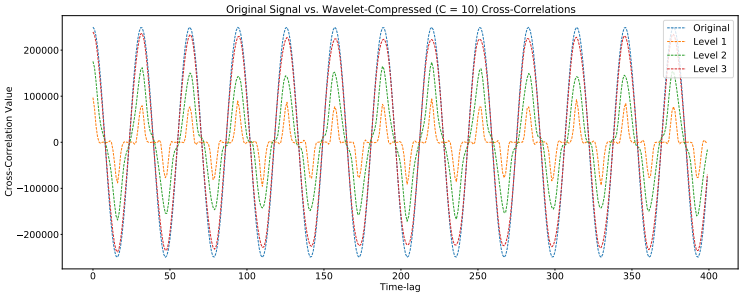

Figure 2.14: Plot of wavelet-domain cross-correlations computed using the top 10% of coefficients in magnitude at levels 1, 2, and 3.

Wavelet compression is considered a useful denoising technique [1], but it is still useful to test how well wavelet-compressed cross-correlations respond to noise. To evaluate this, we added random Gaussian noise ($\mu = 0, \sigma = 4$) to our numeric representations of f and g, seeding the random generator differently before each call. We then calculated the wavelet-domain cross-correlation values on this noisy data at compression factor 10 (using a level 3 Daubechies 3 DWT), and compared it to both the original signal and the wavelet compressed cross-correlations. A plot of these results is shown in Figure 2.15. In this case, there is minimal difference between the noisy and noiseless wavelet compressed results, and only a modest difference between the these and the original signal-derived values, supporting the notion

Table 2.1: Counts of coefficients that are preserved in a level 3 Daubechies 3 DWT, after thresholding with a compression factor of 10. "Total" refers to the number of coefficients stored in the wavelet transformed data without compression.

Coefficient	Total	Preserved (No Noise)	Preserved (Noise)
Approximation	1254	1002	921
Detail 3	1254	0	34
Detail 2	2503	0	13
Detail 1	5002	0	34

that wavelet compressed cross-correlations are robust to noise.

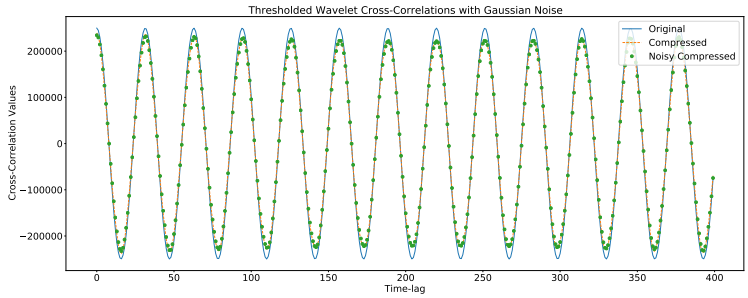

Figure 2.15: Plot of wavelet compressed cross-correlations of two sine curve signals with random noise added, at compression factor $c = 10$. The cross-correlations of the original noiseless signals, computed directly and with the wavelet-domain algorithm at $c = 10$, are added for comparison.

One final observation for this simple test case is the relative breakdown of preserved coefficients in the wavelet compressed representation of f, shown in Table 2.1. In the noiseless case, where $f(t) = 10\sin(2t)$ for $t \in [0, 1000]$, we see that all preserved coefficients are in the approximation level, while all detail coefficients are thresholded. Once Gaussian noise is added, a small number of detail coefficients at each level are preserved, but the majority of preserved coefficients are still from the approximation level. Since f and g both represent a signal active at only one frequency, it appears the scaling function in this DWT best represents data of that frequency.

2.5.2 Signal from Real Data

The previous test case provides a useful example of the effectiveness of our wavelet-domain algorithm and its ability to calculate reasonably accurate cross-correlations while using a relatively small portion of the original data volume. However, real life signal data in most applications is active on a range of frequencies, and generally does not resemble a basic mathematical function such as a sine curve. For a more realistic but still relatively simple test case, we took time series vibration data from a single channel in a multichannel sensor array, sampled over one minute (2 ms per sample). A plot of this signal and its frequency components, calculated using a Fourier transform, are available in Figure 2.16. A heatmap displaying the Fourier transforms of 10 second windows is also available in Figure 2.17. We can see that most of the signal is represented by frequency components below 10 Hz, with additional significant components at roughly 30 Hz and 120 Hz. The 120 Hz component is constant across the entire time sample, while the other major frequency components are somewhat less consistent.

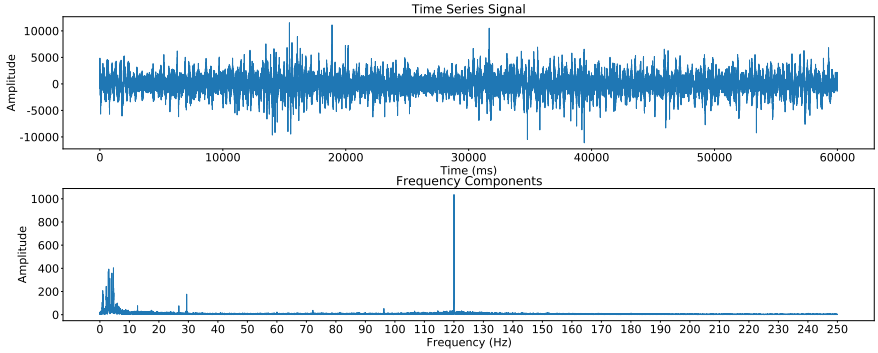

Figure 2.16: Plot of one minute of vibration data from a multichannel sensor array, and its Fourier transform. Note the strongest activity occurs at ≤ 30 Hz and ≈ 120 Hz.

We can calculate the autocorrelations of this signal with a subset of itself - a plot of this is available in Figure 2.18. Note that the cross-correlation values peak at 400 ms (time-lag 200) due to the indexing of the subset used. There are both rapid oscillations within the cross-

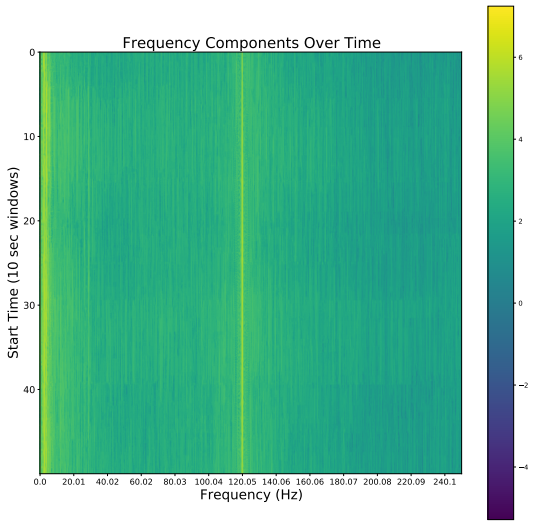

Figure 2.17: A heatmap of 10 second windows of FTs of the data in Figure 2.16.

correlation values, indicative of the relative phases of the signal at the particular time-lags, as well as slower, more gradual changes in these values. The cross-correlations calculated using the traditional signal based method and our wavelet-domain algorithm are virtually identical, since this example uses the full DWT (level 3, Daubechies 3 wavelets).

The time-lags for this cross-correlation test case correspond to intervals of 2 milliseconds each, due to the sampling frequency of this signal. A more detailed visualization of the oscillations, such as in Figure 2.19, shows that roughly six cycles occur every 50 milliseconds, which corresponds to a frequency of 120 Hz. Thus the rapid oscillations in our cross-correlations are almost certainly related to the 120 Hz component within this signal.

For the next part of this test, we calculated the wavelet-domain autocorrelations of this signal two more times, each using only certain subsets of the DWT: just the approximation coefficients for one, and thresholded coefficients at a compression factor of 10 for the other. Plots of results from both of these compression methods are available in Figure 2.19. Notice that the approximation-only autocorrelation does not preserve the rapid oscillations that likely correspond to the 120 Hz component, but otherwise closely follow the original

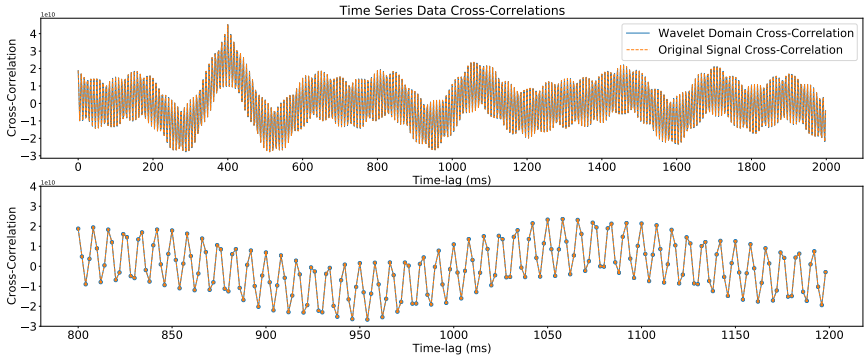

Figure 2.18: Autocorrelations of our signal in Figure 2.16 with a subset of itself. Part of the time-lag range has been zoomed in for clarity.

Table 2.2: A table showing the number of wavelets in a time series signal from a multichannel sensor array that are preserved by thresholding (compression factor 10).

Coefficient	Total	Preserved	Percent Preserved
Approximation	3754	1843	49.09 %
Detail 3	3754	32	0.85 %
Detail 2	7503	871	11.61 %
Detail 1	15002	256	1.71 %
Total	30013	3002	10.002 %

cross-correlation values. Meanwhile, the thresholded autocorrelation preserves slight 120 Hz oscillations, noticeable in the more detailed subsection of the plot, but they also primarily resemble the lower frequency features of the original autocorrelation. Both of these computations on reduced portions of the transformed signal are still valuable for the typical purposes of cross-correlations, such as detecting similar events, but they preserve or emphasize specific features.

Table 2.2 displays the total coefficient counts associated with the DWT of this signal, and the number of coefficients in each level preserved by thresholding to a compression factor of 10. Similar to the tests with a sine function, certain levels of the transform seem to have larger coefficients. In this case, the approximation coefficients are still heavily prioritized, though not to the same degree. Interestingly, the level 2 detail coefficients are also preserved by

more than the overall compression factor, while only a small portion of detail 1 and virtually no detail 3 coefficients remain. This suggests that the level 3 scaling function and level 2 wavelet function best represent the original time series signal, or at least certain components of the signal.

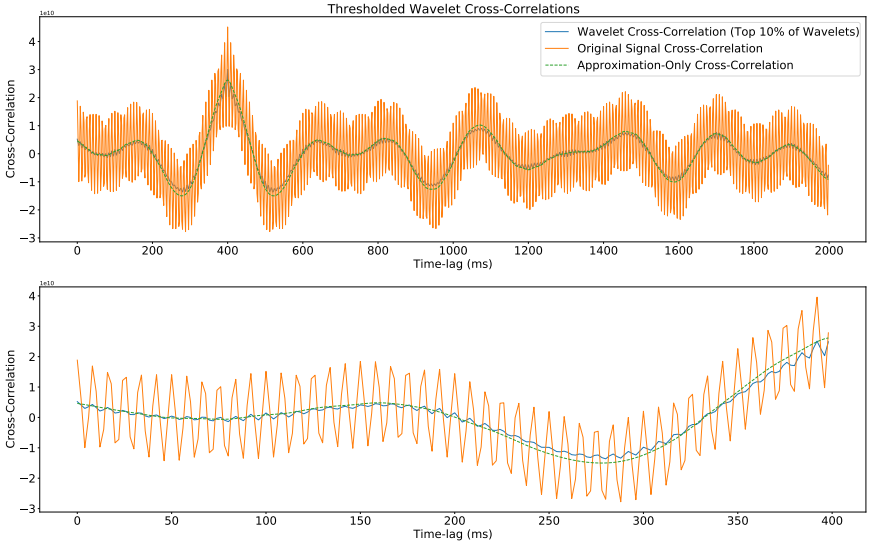

Figure 2.19: Autocorrelations of the signal in Figure 2.16, calculated by the wavelet-domain algorithm using two separate forms of compression (approximation coefficients and thresholding). A closer view of a subset of time-lags is also provided.

To better understand the relationship between the time-domain frequency spectrum and the corresponding wavelet representation, we computed the Fourier transforms of the wavelet and scaling functions present in a level 3 Daubechies 3 DWT, and plotted the results in Figure 2.20. The frequencies have been calculated assuming these functions' supports are defined as they are used in the transform of our vibration data signal, on a scale of 2 ms per sample. We can see that the scaling function's (approximation's) frequency components are strongest below 20 Hz and weaker, but still present, below 50 Hz and at small pockets of higher frequencies. Since most of the stronger frequency components in this example signal are below 30 Hz (especially in the 0-10 Hz range), it seems reasonable that the majority of the preserved coefficients in thresholding are approximation coefficients. Meanwhile, the

level 2 wavelet function (detail 2) possesses the strongest 120 Hz component in this DWT. Given the prominence of that frequency in the signal, it also seems reasonable that a significant number of the preserved coefficients are in detail 2. Neither detail 1 nor detail 3 are active at high amplitude frequencies from the signal in Figure 2.16, especially detail 3, potentially explaining why a large portion of those coefficients are thresholded relative to the compression factor.

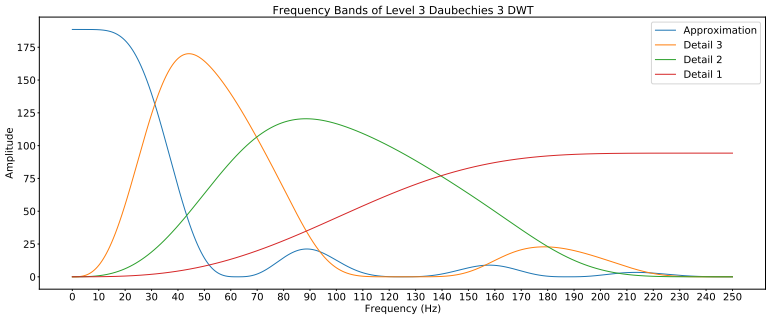

Figure 2.20: The frequency bands of scaling and wavelet functions (approximation and detail levels) in a level 3 Daubechies 3 discrete wavelet transform.

Plots of the Fourier transforms of the wavelet compressed signals, both by approximation-only preservation and thresholding, are available in Figure 2.21. These support our notions developed from observations of the functions' frequency bands. The approximation-only compressed signal is active primarily at frequencies represented by the scaling function: 0-50 Hz, and to a much lesser extent at roughly 90-100 and 150-160 Hz. The strongest activity is below 10 Hz, aligning with both the vibration signal and the scaling function, and other significant activity occurs at roughly 30 Hz. The thresholded signal frequency components are very similar below 50 Hz, which seems reasonable as the majority of preserved coefficients are in the approximation level, with the additional presence of frequencies around and especially at 120 Hz. A few frequencies seem to be more prominent (at least relatively) here than in the original signal in Figure 2.16, primarily at roughly 33 Hz and 130 Hz. However, the strongest frequency activity is consistent with the original signal.

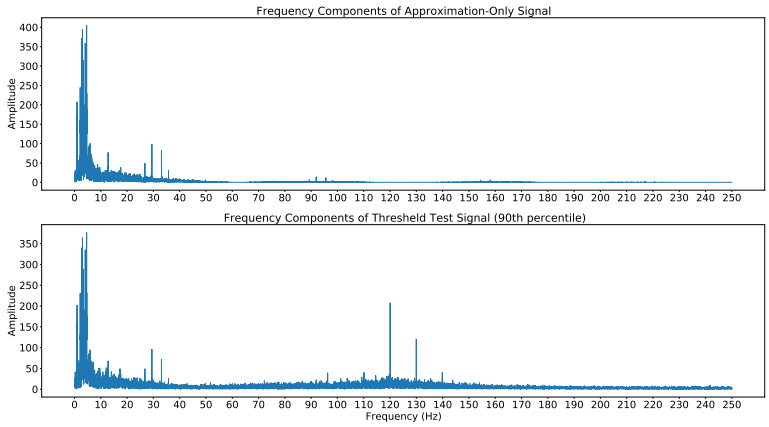

Figure 2.21: Plots of the frequency components of our wavelet compressed signal. The first one uses only the approximation coefficients, while the second one thresholds with compression factor 10.

The most significant change in the thresholded vibration data signal is the reduction in energy at 120 Hz. While still prominent, it has a lower amplitude than in the original data. This is likely related to the 120 Hz oscillations in the wavelet compressed cross-correlations being far less pronounced than those computed from the original signal. Since activity at 120 Hz is consistent across the entire minute of data, it is probably not related to temporal changes within the signal.

Compressing by only using one or more specific levels of the transformed signal can effectively mimic a band-pass filter for specific frequencies (depending on the wavelet and level used), providing a means to highlight important information within certain frequency bands. Meanwhile, using a compression technique such as thresholding can preserve all of the important frequency components of a signal and relevant features within the signal cross-correlations. However, it does not necessarily preserve the relative differences between the most prominent frequencies, which may emphasize or de-emphasize certain behaviors in the resulting cross-correlations. Because of this, wavelet-compressed cross-correlations may have a high entrywise error compared to those from the original data, but can still provide some of the same useful information to indicate time-lags at which there is high similarity.

Table 2.3: A table showing the runtime of the wavelet-domain cross-correlation algorithm on two fixed signals, with key enhancements implemented.

Method	Computing full V	Redundancy Rules	With FFTs
Runtime	231.0 ms	77.6 ms	10.6 ms

2.5.3 Algorithm Speed and Performance

The primary benefit of the wavelet-domain cross-correlation algorithm's ability to operate directly on the DWT coefficients is that there is no longer a need to reconstruct the original data (or an approximation of it) using an inverse wavelet transform. If our transformed data is stored in a sparse, wavelet compressed format, than this sharply reduces the total memory costs of computing large numbers of cross-correlations (though our algorithm does still produce a modest overhead, with the formations of V, X^r, and X^s in Algorithm 2). However, it is also important to evaluate the speed of the new method, and compare it to traditional cross-correlations computed directly on the reconstructed signal data.

The original multi-time-lag implementation of our wavelet-domain algorithm was rather slow. This is primarily because the computation of the interior coefficient cross-correlation values (represented by $\vec{v}$ in Algorithm 1, and V in Algorithm 2) requires us to compute the cross-correlations of the coefficient vectors $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$, at several offsets. Employing the redundancy pattern described at the end of section 2.3 allows us to only compute the cross-correlations for the first row of V, then use simple rules to calculate subsequent rows, which sharply reduces the total number of floating-point operations required to compute V. Moreover, using FFTs with the cross-correlation theorem to compute the first row improves performance further. A relative comparison of the wavelet-domain algorithm with these changes, employed on the same two signals, is available in Table 2.3.

In Figure 2.22, we plot a comparison of the runtimes of standard signal cross-correlations and our dense wavelet-domain algorithm, on a range of time-series signal lengths. The

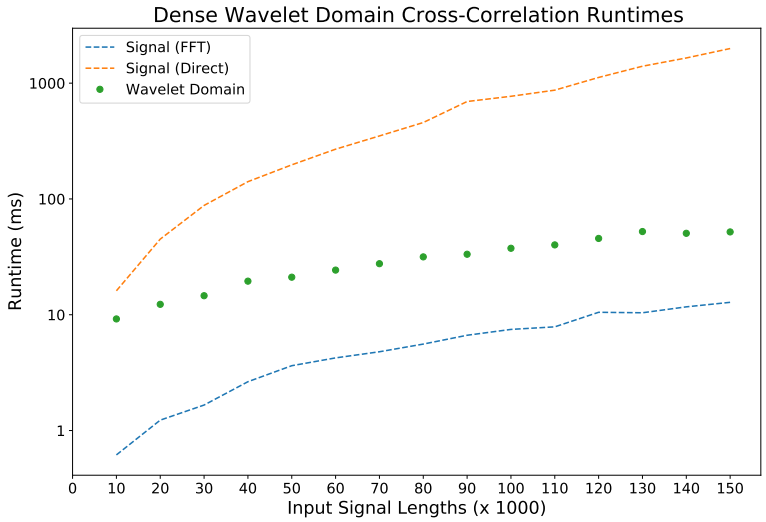

Figure 2.22: Plot comparing the runtimes of signal-based cross-correlations and our wavelet-domain algorithm on dense coefficients. Displayed runtime is logarithmic.

number of time-lags in this computation is fixed at 1000. We see that the wavelet-domain implementation is much faster than the direct cross-correlation method, and scales much better with longer signals: it is only 1.75 times faster at signal length 10000, but 38.3 times faster at signal length 150000. However, our algorithm is slower than signal based cross-correlations computed with FFTs, though its relative scaling improves with longer signal lengths - it is 14.9 times slower at length 10000, but only 4.05 times slower at length 150000.

A plot comparing FFT signal cross-correlations to sparse wavelet-domain cross-correlations at compression factors 5, 10, and 20 is available in Figure 2.23. The main performance improvement of the sparse algorithm is by computing the entries of V at each level with a sparse cross-correlation function on $\vec{x}_j^{(r)}$ and $\vec{x}_k^{(s)}$. Although we can no longer use FFTs with the cross-correlation theorem to compute the entries of V, sparse cross-correlations are much faster than dense cross-correlations at sufficient high sparsity levels. At compression factor $c = 5$ (where the coefficients vectors are 80% sparse), the wavelet-domain implementation is still significantly slower than signal FFTs, although it is faster than the dense algorithm. At $c = 10$, we see comparable performance to signal FFTs at longer lengths, and even better

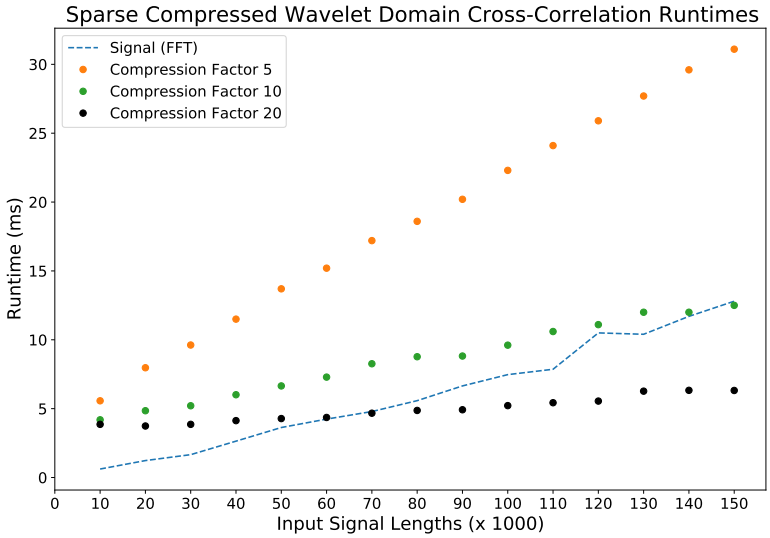

Figure 2.23: Plot comparing the runtimes of FFT signal cross-correlations and our wavelet-domain algorithm on sparse compressed coefficients, at different compression factors.

performance at length 150000 (the upper end of most typical use cases in geophysical data analytics). We see significant improvements in speed at $c = 20$, though we have not found this level of compression to be reliably accurate in all use cases. Still, this test suggests the sparse wavelet-domain algorithm has good performance scaling compared to typical FFT-based signal cross-correlations.

As a final performance test, we evaluated the runtime of various signal and wavelet-domain methods on larger problems: the cross-correlations of two signals of lengths 5×10^5 through 2×10^6, with 1000 time-lags. The results are shown in Figure 2.24. The dense algorithm (not plotted) is roughly 4.4 times slower than the FFT based signal cross-correlation on signals of length 10^6, which is consistent with the differences noticed in tests on smaller problems. However, the sparse algorithm scales well on the larger problems: at compression factor $c = 10$ (90% sparse), it is roughly 1.8 times faster at signal length 10^6 and over 2 times faster at length 2×10^6; at compression factor 20 (95% sparse), it is over 4.8 times faster at length 10^6 and over 5.8 times faster at length 2×10^6. This further supports the sparse

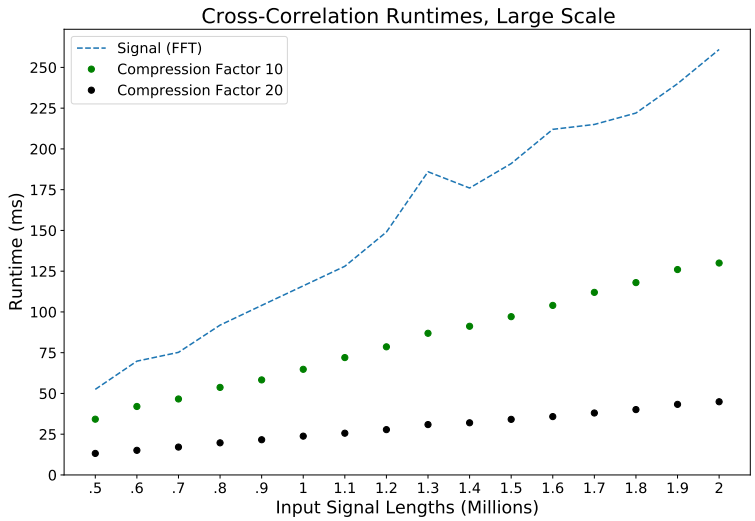

Figure 2.24: Plot comparing the runtimes of FFT signal cross-correlations and our wavelet-domain algorithm on sparse compressed coefficients, at longer signal lengths.

wavelet-domain cross-correlation algorithm scaling well on larger problem sizes. Further enhancements to the sparse algorithm, perhaps by implementing X^r and X^s as described in Algorithm 2 as sparse matrices, or by identifying wavelet families that allow for higher compression factors while preserving accuracy, will likely improve this scalability further. Many current applications of cross-correlations on vibration data sets still feature signals of smaller lengths, roughly $10^4 - 10^5$. However, as higher resolution datasets become more common in geophysical analysis, demanding cross-correlations of longer signals, the improved scaling of the sparse wavelet-domain algorithm may become more valuable.

Chapter 3

Accelerated Multichannel Analysis of Surface Waves

This chapter mostly consists of material already submitted for publication. [9]

3.1 Overview of Problem

Multichannel Analysis of Surface Waves (MASW) consists of two main processes, MASW dispersion and MASW inversion. Dispersion requires collecting near-surface wave data, primarily ground roll, from an array of sensors, and using spectral analysis to compute the dispersion curve of the ground roll. This process is only done once per site, and typically does not lend itself well to parallelization (it often requires some judgement and manual input by the user to "trace" the ideal curve in the dispersion plot). MASW inversion is better suited to parallelization, and there is a much greater need for it since it is run once for each proposed near-surface model of the site, and there can be hundreds of such models. Our algorithms are for MASW inversion.

A typical use case for MASW inversion has two "loops", regarding potential optimizations. Optimizing the "inner loop" consists of further enhancements to the MASW inversion algorithm itself, improving its performance when computing the theoretical dispersion curve of a near-surface model. As one computes MASW inversion for more models, improvements

to the inner loop become more valuable. Optimizing the "outer loop" means reducing the number of models that must be assessed to find an accurate prediction for this near-surface. This in turn reduces the total number of necessary iterations of MASW inversion.

Existing optimizations to MSAW inversion focus on improving the outer loop. Various algorithms have been employed to optimize the choice of near-surface models to test, including pattern matching and bee swarm algorithms [15, 16]. However, there were no previous open-source MASW implementations that optimize the inner loop through parallelization or use of GPUs. Our MASW inversion implementations do this. These inner loop enhancements can improve performance for both individual runs and large quantities of runs of MASW. Moreover, they can be combined with the outer loop optimizations to further improve performance.

We review the a standard serial algorithm for MASW inversion from MASWaves, an open source implementation written in MATLAB [12], and note optimizations we have made in our C implementation. We then propose new algorithms that parallelize MASW inversion using MPI, and that perform MASW inversion on GPUs.

3.2 Serial Implementation

Before parallelizing the algorithm for MASW inversion, we need to implement a C serial version of the code which can then be modified with MPI and CUDA. We used MASWaves, an existing implementation written in MATLAB as our reference to compare against [12]. Initially our C serial implementation was a simple port of MASWaves to C, but we later made a number of changes to the algorithm to improve its efficiency.

The model evaluation process in MASW consists of two main algorithms. First, the model

parameter inputs are used to compute the most likely velocities for each wavelength in the experimentally derived dispersion curve. This is done in the function MASW Theoretical Dispersion Curve, illustrated in Algorithm 3. The model parameter inputs (M) are the number of finite thickness layers, the thickness and density of each layer, and the compressional (P) and shear (S) wave velocities through each layer.

Algorithm 3 MASW Theoretical Dispersion Curve

```
M = Guessed model parameters
W = Wavelength values from experimental dispersion curve
C_t = Velocity corresponding to each wavelength given M
V = Range of potential velocity values to test
d_old = Determinant value for previous entry in V
d_new = Determinant value for current entry in V
for w in W do
    d_old = stiffness_matrix(M, V[0])
    d_new = stiffness_matrix(M, V[1])
    n = 1
    while sign(d_old) == sign(d_new) do
        n = n + 1
        d_old = d_new
        d_new = stiffness_matrix(M, V[n])
    end while
    C_t[w] = V[n]
end for
```

MASWaves utilizes the stiffness matrix method [6] to compute the theoretical dispersion curve. Given a wavelength, sparse stiffness matrices are computed for each test velocity (in V) in increasing order, until one has a determinant (d_{new}) with a different sign than its predecessor, suggesting the matrix generated at this test velocity is near a singular matrix. This test velocity ($V[n]$) is then stored as the theoretical velocity corresponding to that particular wavelength, i.e. it is stored as $C_t[w]$. The process is repeated for each wavelength in the experimental dispersion curve to generate a theoretical dispersion curve of Rayleigh wave velocities and their wavelengths.

The entries of each matrix are a function of its corresponding test velocity in V and the model parameters M: the number of finite-thickness layers, layer densities, layer thicknesses, layer shear wave velocities, and layer compressional wave velocities, which are all constant for a single run of MASW inversion. The matrices are always symmetric and heptadiagonal, and their size is dependent on the number of layers in M.

A stiffness matrix may be generated for each wavelength and each test velocity. Realistically a dispersion curve may have up to 100 wavelengths and 1,000 test velocities, requiring a total of 100,000 stiffness matrices to be generated (in practice this number is often lower, since the ideal test velocity is usually found before all velocities are checked for a given wavelength). These stiffness matrices also have a sparse banded structure, which is not clearly taken advantage of in MASWaves but is in our algorithms.

The algorithm also requires the determinants for each of these matrices to be computed. The stiffness matrices are of size $2(N+1) \times 2(N+1)$, where N is the number of finite-thickness layers in M. Since they have a symmetric heptadiagonal structure, computing the entries of a stiffness matrix and its determinant requires $O(N)$ operations.

Algorithm 4 MASW Misfit

$C_t =$ `Velocity corresponding to each wavelength given` M
$C_e =$ `Experimentally derived velocities`
$e =$ `Relative errors`
$l =$ `Length of` W`,` C_t`, and` C_e
$m =$ the average misfit
for $i = 1$ to l **do**
    e = e + $\frac{|C_t[i]-C_e[i]|}{C_e[i]}$
end for
$m = \frac{e}{l}$

Once the theoretical dispersion curve C_t is computed, the second part of the algorithm compares its velocities to the velocities experimentally derived from the data, labelled C_e. The average relative error is labelled the misfit, and indicates how accurate M serves as a

model of the ground structure. Essentially the misfit behaves as a loss function for MASW.

The model parameters M, along with the given test velocity and wavelength, are used to compute each stiffness matrix. If the theoretical velocity for each wavelength is close to the experimental velocity (i.e. the misfit is small), then the model parameters are more likely to be a good approximation of the ground truth.

One downside to MASW is that it does not have any form of backpropagation to accompany its loss function. Therefore the only way to minimize the misfit is to compute theoretical dispersion curves for a large quantity of plausible model parameters, each of which require forming and finding the determinants of up to 100,000 small matrices. While methods have been used to more efficiently select which models should be tested, such as the pattern search and bee swarm algorithms, these still require a large number of model theoretical dispersion curves to be computed [15, 16]. Since each of these curves and their misfits can be computed independently, this algorithm can benefit from parallelization, both between different test models and within individual models.

3.3 MPI Parallelism

It is possible to visualize MASW as a mesh computation, although it is not a literal ground mesh. Stiffness matrices must be computed with several different values of wavelengths and test velocities and the same values of model parameters. If the wavelength and velocity values are thought of as the x and y axis, then `MASW_Theoretical_dispersion_curve` can be viewed as computing data points along a two dimensional "grid". It is reasonable to partition this grid of computations into multiple processes in an MPI implementation.

There are two obvious ways to partition this grid. The stiffness matrix method heavily uti-

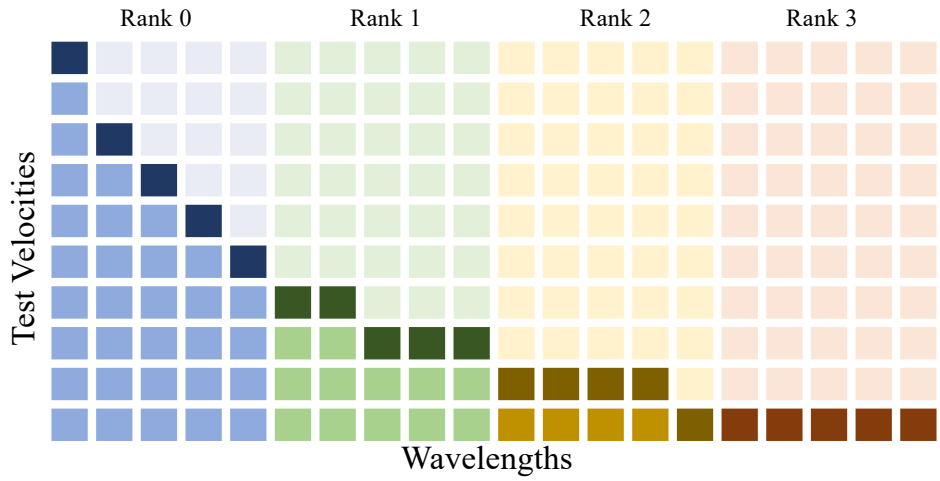

Figure 3.1: Visualization of original MPI partition. Darker colors correspond to determinants that were computed, with the darkest highlighting the theoretical curve. Each rank has its own color scheme. Note the number of computed determinants varies heavily for each rank.

lizes the test velocities in computing the individual matrices. Therefore, one could partition the grid along the velocity axis: given S processes and V test velocities, compute stiffness matrix determinants for all wavelengths and the first $\frac{V}{S}$ velocities in process 0, then the next $\frac{V}{S}$ velocities in process 1, and so on. This approach would allow components of the stiffness matrix dependent on the test velocity to be pre-computed, reducing the number of repetitive computations.

Partitioning along the velocity axis would have significant drawbacks, however. Once the determinants are computed, a linear search must be performed along each wavelength to find the first sign change. Since the determinants for each wavelength are split along multiple processes, a significant amount of communication would be required between each process to find the first sign change. In addition, determinants for higher test velocities might not even need to be computed if the first sign change (and thus correct test velocity) has already been found. Thus this approach may lead to several unnecessary computations.

Because of these difficulties, the method used was partitioning along the wavelength axis.

The stiffness matrix computations do not feature the wavelength input as much as the test velocity, so there is little potential to pre-compute components of the matrices for each wavelength. But finding the ideal test velocity for a given wavelength has no dependence on other wavelength values, so no communication between processes is required for the dispersion curve. In addition, once the ideal test velocity is found for a particular wavelength, the process can begin computing matrix determinants for the next wavelength. This preserves the serial implementation's advantage of avoiding unnecessary matrix and determinant computations. Partitioning along the wavelengths also makes computing the misfit parallel as well, with only one reduction operation required to combine the errors from each process.

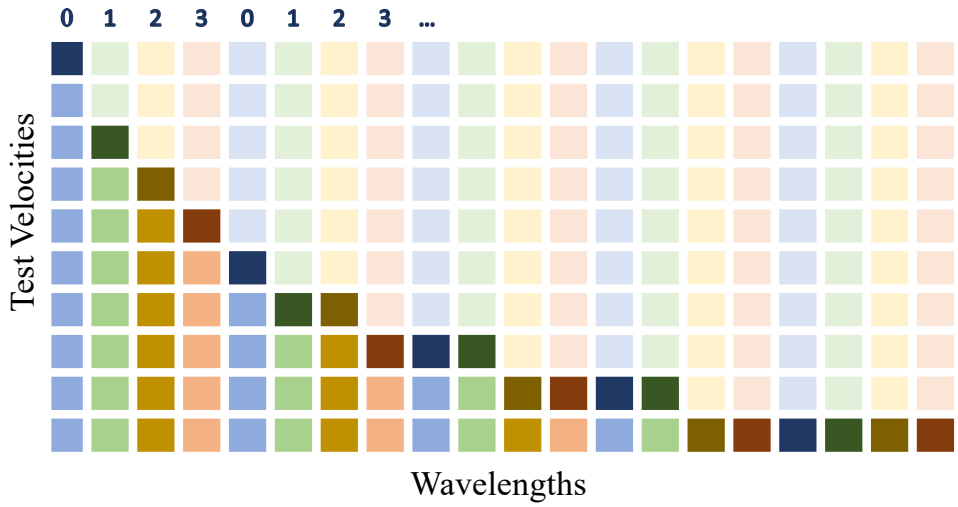

Figure 3.2: Visualization of modular MPI partition, with the same color rules in place. In this case the computed determinants are balanced more evenly.

Since the algorithm does not require communication between different wavelengths of the dispersion curve, there is freedom to choose exactly how to partition the problem along the wavelength axis. The original approach was to assign wavelengths contiguously: given s processes and W wavelengths, assign wavelengths $0, 1, \ldots, \frac{W}{s}$ to process 0, wavelengths $\frac{W}{s} + 1, \ldots, \frac{2W}{s}$ to process 1, and so on. However, dispersion curves are continuous and typically monotonic, so this leads to an unbalanced workload between processes. We determined a

more efficient approach was to partition wavelengths in a modular pattern: assign every wavelength equivalent to 0 mod s to rank 0, every wavelength equivalent to 1 mod s to rank 1, and so on. This is a simple way to improve the load balancing of the MPI implementation, by taking advantage of typical features in the dispersion curve data.

3.4 GPU Acceleration

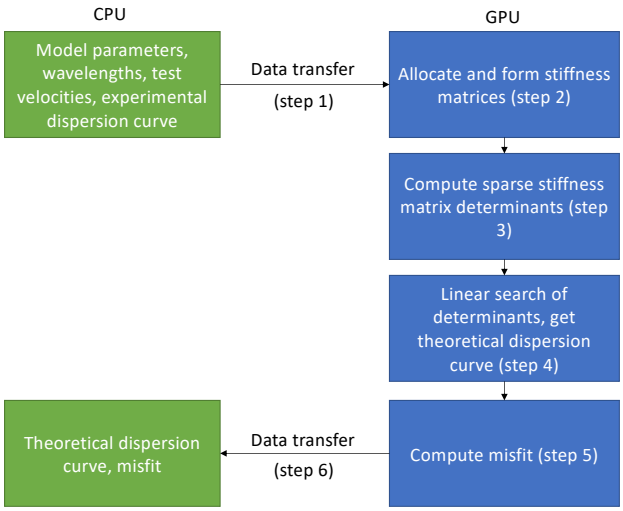

Figure 3.3: General outline of GPU implementation.

Many workstations used by geotechnical engineers contain CPUs with multiple cores as well as graphics cards. Thus, in addition to writing MASW for MPI, we have also implemented the MASW algorithm for graphics processing units (GPUs). MASW requires calculating many determinants of sparse matrices, a problem which has not previously been adapted to GPUs.

There are five main steps to the GPU implementation that are distinct from the serial and MPI versions of MASW. First, the initial input values to the algorithm - wavelengths, test velocities, experimental dispersion curve, and the ground model parameters - are transferred

from the CPU "host" to the GPU "device" as shown in step 1 of Figure 3.3. Copying memory between the host and device is costly, but the volume of data copied here is relatively small (the test velocity array may have ≈ 1000 entries, and the other inputs are scalars or much shorter arrays), so this transfer is not problematic.

Once the inputs are on the device, the stiffness matrices are allocated as global device memory and their entries are filled in based on the model parameters M, shown in step 2 of Figure 3.3. Forming the stiffness matrices is one of the most costly steps in MASW, so effective parallelization is critical. Since a GPU contains thousands of cores (as opposed to the dozens that may be available on one or more CPUs), it is feasible to partition the problem along both the wavelength and velocity axis dimensions of the grid. In fact, it is reasonable to compute all the stiffness matrices concurrently since they are mutually independent. Using CUDA, each thread is assigned to fill in the entries for one stiffness matrix.

Computing values for a stiffness matrix is $O(N)$ as described in the serial implementation, where N (the number of finite thickness layers in M) is typically small ($N \leq 10$). Entries in the stiffness matrices are incrementally increased multiple times, so trying to compute a stiffness matrix across multiple threads can lead to race conditions. Thus assigning a single thread to each stiffness matrix is reasonable.

Once the stiffness matrices are formed, Gaussian elimination is performed so their determinants can be easily computed, shown in step 3 of Figure 3.3. This is roughly equal to forming the stiffness matrices in terms of time cost. Initially, we used the function `cublasZgetrfBatched()` to perform LU factorizations on the stiffness matrices. This function took over 50% of the runtime of the GPU implementation, most likely because it did not take advantage of the stiffness matrices' banded structure and was therefore $O(N^3)$ for each factorization. Because of this, we replaced it with a kernel that assigned one thread to each stiffness matrix to perform a banded Gaussian elimination. This function was roughly

ten times faster than `cublasZgetrfBatched()`, and approximately equal in runtime to the kernel that formed the stiffness matrices.

The next step is to find the first test velocity whose corresponding determinant is the opposite sign of its predecessor for each wavelength, which is step 4 in Figure 3.3. Normally this would require a linear search along all the test velocities for each wavelength - a serial $O(N)$ process that does not lend well to GPU architecture. However, there is still some potential to partition the problem along the device cores. The search kernel breaks up the stiffness matrices along its blocks by wavelength, and along threads within each block by test velocity. This is illustrated in Figure 3.4.

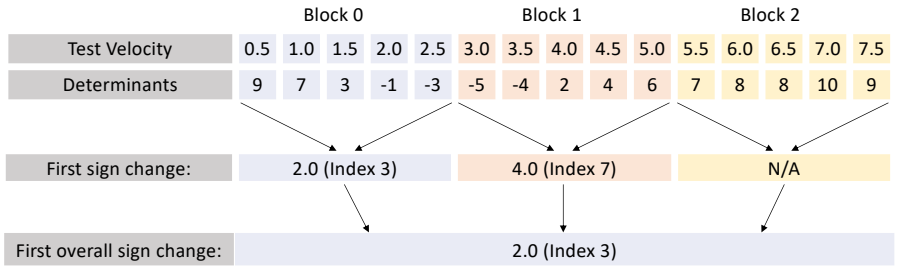

Figure 3.4: Outline of GPU determinant search along one wavelength. The first sign change within each block is found, then a search over the blocks finds the first overall sign change.

Each wavelength will be paired with multiple blocks, since the block size is assumed to be 256 (for compatibility with older GPUs) and often MASW is run with ≥ 500 test velocities. Within each block, thread i computes the determinant of its respective stiffness matrix by multiplying the diagonal entries, then compares it to the determinant of thread $i+1$. It then stores the result of the sign comparison into shared memory.

Once this is complete for all threads, the first thread of each block then performs a linear search for the first sign change in its shared memory block, and stores this as the index of the first sign change within that range of test velocities. These results are placed in the matrix

$\in \mathbb{R}^{\ell \times b}$, where ℓ is the number of wavelengths and b is the number of blocks assigned to each wavelength. The next kernel then iterates over S to find the first recorded sign change for each wavelength, which is a small linear search over approximately 4 entries. The test velocity whose stiffness matrix determinant produced the first sign change is then labelled as the velocity corresponding to that wavelength in the theoretical dispersion curve.

The final steps are to average the errors between the theoretical dispersion curve and experimental curve to get the overall misfit, and to send the theoretical dispersion curve and misfit to the host, which are steps 5 and 6 of Figure 3.3. The former is effectively equivalent to a vector dot product (in terms of number of floating-point operations), which is straightforward to implement in CUDA. The latter is simply a CUDA memory copy involving a vector of length ≤ 100 and a scalar. Steps 1 and 6 require only a small amount of data to be transferred between the host and device, while the stiffness matrices, which take up much more memory, are allocated and freed exclusively on the device.

3.5 Test Cases

We developed a variety of tests for correctness of the code (unit tests and end-to-end tests), as well as performance and scalability tests to understand code efficiency.

We used two main test cases to evaluate MASWAccelerated's performance. A synthetic dispersion curve of identical wavelength values was used to provide a "uniform" dataset. This is useful because of the design of the serial and MPI algorithms - since every test velocity is evaluated in increasing order, varying wavelengths with different theoretical velocities will have different runtimes. Thus a uniform test allows us to observe other factors that may affect how MASWAccelerated scales with more data. The second test case was a more realistic

"variable" dataset, with decreasing wavelengths, matching a typical dispersion curve. We present these test cases in the code as `testScaling` and `testProcess`, respectively. Both of these functions also have wrappers (`testScaling_full` and `testProcess_full`) that enable test cases to be run multiple times in a loop.

3.5.1 Serial Tests

The primary purpose of our serial C implementation is to enable usage of MPI and CUDA to parallelize MASW. However, it is still important to make sure the serial C version is correct, and it is useful to compare its speed to the original MATLAB version of MASWaves. We timed the speed of MASWaves on our realistic dataset, and compared it to MASWAccelerated's speed on the variable dataset when run in serial. Both of these tests were run ten times on a laptop with a 3.1 GHz Intel Core i7 CPU, and the mean results are shown in Figure 3.5, along with errors denoting one sample standard deviation.

Initially the serial C implementation, while mathematically correct, was slower than MASWaves, which is not ideal. This is likely because MASWaves made use of MATLAB's vectorized operations to compute stiffness matrix determinants, while the initial Gaussian elimination algorithm written for MASWAccelerated was not vectorized.

The stiffness matrices formed by MASW are always sparse, and moreover have a banded heptadiagonal structure (nonzero entries only on the main diagonal and the three above and below it). We used this fact to improve the Gaussian elimination algorithm to be only $O(N)$ instead of $O(N^3)$, which increased the algorithm's speed by 2.0 times, as shown in the C (sparse) column. We also noticed components of the entries in the stiffness matrices could be pre-computed before iterating over the matrix entries in a loop, reducing the number of

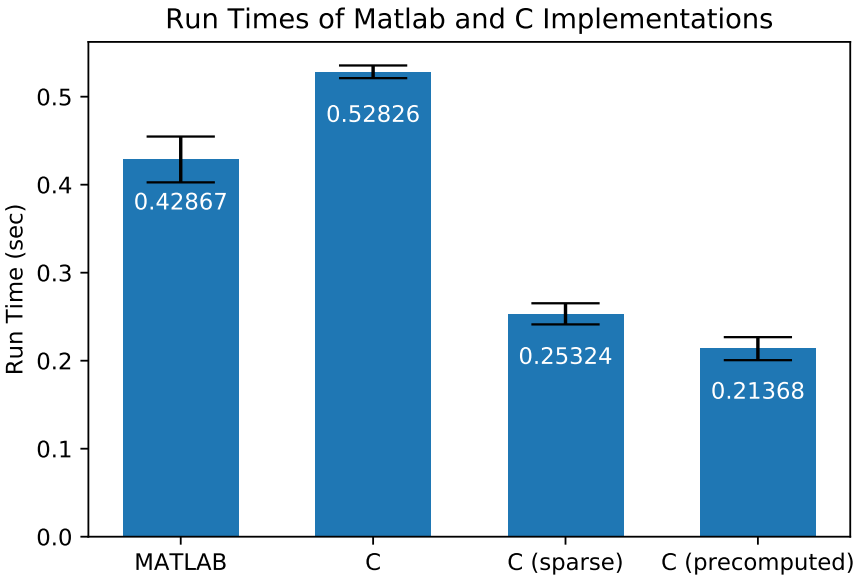

Figure 3.5: Comparison of MATLAB and C on a variable dataset.

arithmetic operations required and increasing the algorithm speed a further 18% as shown in the time for C (precomputed). These improvements enable the serial C implementation to be over twice as fast as MASWaves without any parallelization or vectorization calls. The precomputed serial C version is used as the basis for the MPI algorithm, and for performance comparison with the GPU algorithm.

It is worth noting the runtime for the first instance of MASWaves was significantly slower than subsequent runs, causing the increased variability. This may be due to some type of caching effect in MATLAB which is not present in C.

3.5.2 MPI Tests

As the MPI implementation with one process is virtually identical to the serial implementation, the main purpose of testing is to evaluate how it scales with more processes and larger

datasets. For this purpose we ran a few strong and weak scaling studies.

For the strong scaling study, we first tested the algorithm on the uniform dataset with 1000 wavelengths, allowing for distinctions in runtime to be more noticeable. The results are shown in Figure 3.6, which shows the average runtime for each process count on ten runs.

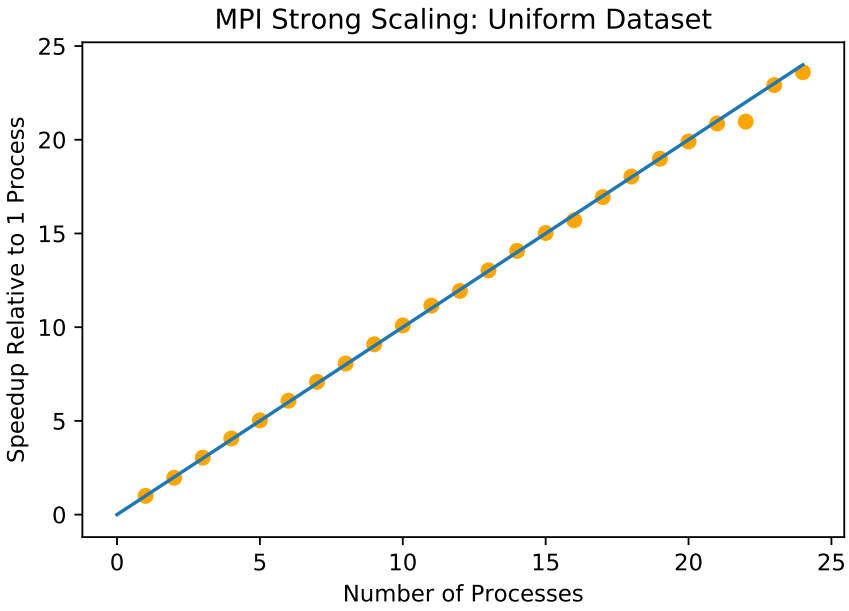

Figure 3.6: MPI strong scaling on uniform data.

This test was run on the NewRiver computing cluster at Virginia Tech, using two Haswell E5-2680v3 2.5GHz processors with 12 cores each. The algorithm scaled almost exactly linearly with more cores, which is expected given the minimal amount of communication required between processes and the highly parallel method used to partition MASW.

In practice, dispersion curves usually have a couple features that make the MPI algorithm scale less than linearly, as seen in the strong scaling test for the uniform dataset.

One problem is that dispersion curves do not have identical wavelength values throughout, but rather varying wavelength values which correspond to different velocities. Since MASW must evaluate test velocities in increasing order to identify the determinant sign change,

entries with larger velocities will require more stiffness matrix determinants to be computed, resulting in a longer run time. Because of this, the MPI algorithm is prone to load imbalancing on realistic data, even though the entries of the dispersion curve are partitioned evenly.

For near-surface imaging, most dispersion curves are decreasing in both wavelength and velocity, and usually resemble a continuous curve. A naive contiguous partition of the dispersion curve will place entries with similar wavelengths and velocities on the same process, thus resulting in a few processes receiving all of the high-velocity entries, exacerbating the load imbalance. The modular partition of the dispersion curve mitigates this problem, and improves the MPI algorithm's strong scaling as a result. The comparison of speedup for these partitions on the variable dataset can be seen in Figure 3.7. A line has been added to compare both to ideal linear scaling.

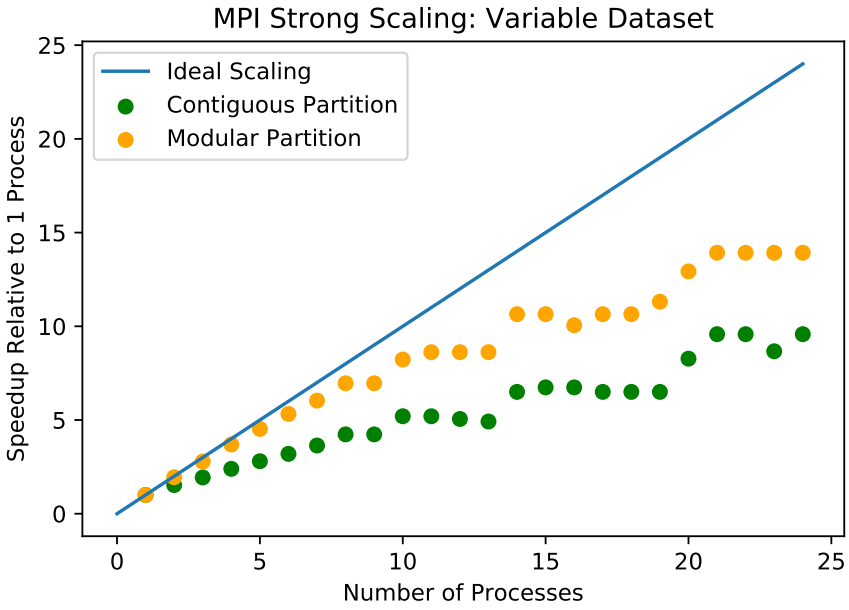

Figure 3.7: MPI strong scaling on variable data, using different partitions. The variable data highlights the two features that can make MASW problematic for MPI: large variations in wavelength values for the dispersion curve, and short dispersion curve length.

This test was run on the same hardware as the uniform strong scaling, and again the average of ten runs per size was used. Although still not linear, the modular partition scales significantly better than the original contiguous partition: with 3 processes it has a speedup of nearly 2.8 compared to 1.9 for the naive approach, and with 8 processes it has a speedup of nearly 7.0 compared to only 4.2. Overall, the MPI algorithm with a modular partition will have near-linear scaling for most datasets with relatively few processes.

The reduction in speedup with more processes is likely due to another problem - most dispersion curves have a relatively short length (the variable dispersion curve has 40 entries, which is fairly typical). When the size of the partition is small, approximately 10 or less, each additional process reduces the number of dispersion curve entries computed for all of the other processes. For example, at size 3 each process is computing velocities for 13 or 14 wavelengths, while at size 4 each process is computing velocities for only 10 wavelengths. This is a significant reduction in workload and results in major speedup as seen in Figure 3.7. But at larger sizes there is not a reduction in workload for every process. For example, at size 20 each process is computing 2 entries, while at size 24, 16 processes are still computing 2 entries and the last 8 are computing 1. Since many processes have no reduction in workload, the overall runtime of the algorithm is not reduced. Therefore, the MPI partition scales near linearly with relatively few processes (depending on the size of the dispersion curve), but experiences diminishing returns with more processes.

A weak scaling study was also performed on the NewRiver cluster using the uniform dataset, again with the same hardware and taking the average of ten runs per size. The size of the dispersion curve was 1000 $\times$ the number of processes. As seen in Figure 3.8, the MPI implementation scaled efficiently by this measure, having no significant increase in time with more data and processes. There are a few sizes with marginally higher runtimes - 7, 10, and 15 - but these are minor and likely due to external factors, such as additional processes

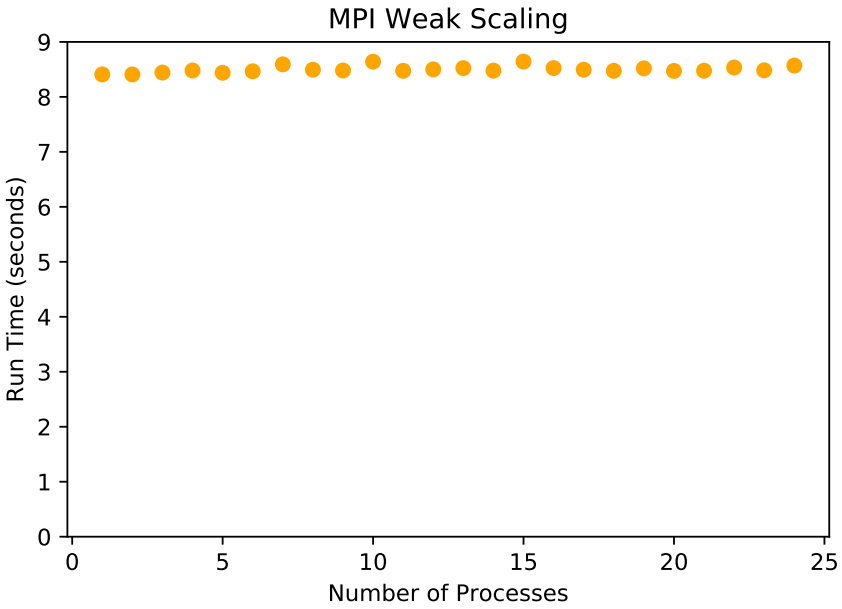

Figure 3.8: MPI weak scaling on uniform data, with a dispersion curve of length 1000 × number of processes.

being run on these resources. Like the strong scaling study with uniform data, this test highlights the minimal communication requirements for the MPI implementation.

3.5.3 GPU Tests

First we compared the CPU and GPU implementations on the variable dataset. This was done eleven times in the wrapper loop `testProcess_full` using the MPI implementation (with one process), then eleven times both in a loop and separately with the CUDA implementation. The resultant run times are shown in Figure 3.9. Note the blue bars are the first run for each method, while the orange bars are the means of subsequent runs with their sample standard deviation posted. The CPU algorithm is unchanged from the serial and MPI tests, but its runtime is different since it was run on a different machine.

All three of these implementations were run on the same desktop, using an Intel Xeon CPU

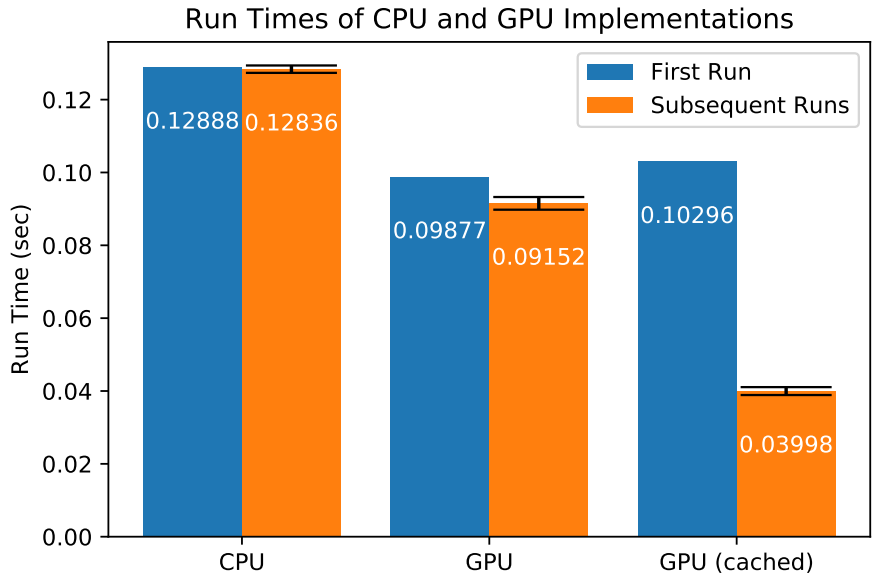

Figure 3.9: Comparison of CPU and GPU on the variable dataset.

E3-1271 v3 @ 3.60 GHz and an Nvidia Quadro K620 GPU. The first run is the run time of the first instance of `MASW_inversion`, while the subsequent runs denote the mean runtime of all other instances. The error bars denote one standard deviation for the subsequent runs. The GPU implementation of `MASW_inversion` is about 25% faster than the CPU implementation when run once, but over 3.2 times faster when it is run multiple times in a for loop. This is likely because of just-in-time compilation, used by Nvidia to allow CUDA kernels to benefit from new device architectures. When a kernel is run multiple times within a function call, it only needs to be compiled for the first kernel run while it is "cached" for subsequent runs. Since MASW inversion is often run multiple times with different test velocity models, it is reasonable to call MASWAccelerated functions repeatedly to run inversions on multiple models to take advantage of this caching effect. It is worth noting that, while not as dramatic, there is still roughly 8% speedup when running the inversion multiple times individually on the GPU, but no significant speedup for the CPU implementation.

We next use the uniform dispersion curve to evaluate how the GPU implementation performs on progressively larger datasets. This test ran the MPI (one rank) and CUDA implementations of `MASW_inversion` on dispersion curves of lengths 50 - 500. The wavelength values of these curves were designated to match test velocities of 72, 238, and 256 (each implementation and each dispersion length was run three times with three different wavelength values). The results are shown in Figure 3.10, run on the same desktop used for the previous test.

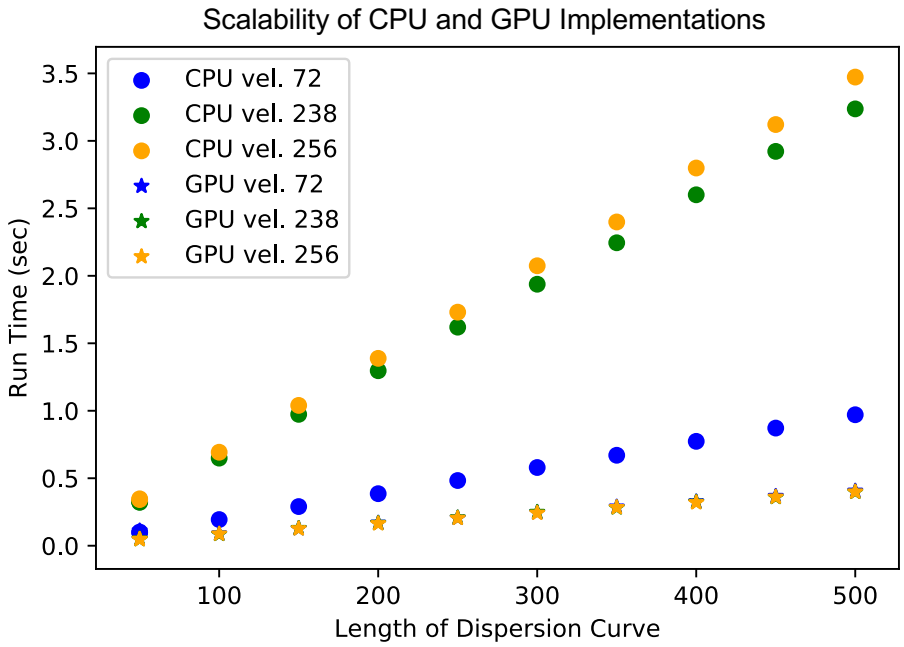

Figure 3.10: Comparison of CPU and GPU on increasing uniform datasets.

As expected, the CPU implementation scaled linearly with the length of the dispersion curve. The exact run time is highly dependent on the theoretical velocity values, as shown in the MPI scaling studies. The GPU implementation was significantly faster (since the caching effect was recognized in testing the variable data, we made use of it here), and had no dependence on the theoretical velocities since all stiffness matrices are computed regardless.

It is worth noting that too large a dispersion curve or too many test velocities can overload the GPU memory on the CUDA implementation, since all the stiffness matrices are formed

concurrently in global memory and the dispersion curve and test velocities determine the number of stiffness matrices. The exact upper limit depends on the memory space of the GPU and the number of finite thickness layers in the model M (which determines the size of the stiffness matrices), but typical problem sizes for MASW will not take up too much memory, even for older GPUs. We found dispersion curves larger than 500 typically caused memory problems for the Quadro K620 GPU, which has 2 GB of global memory. This is because each stiffness matrix has 196 entries, each of which is a CuDoubleComplex datatype that takes 16 bytes, so 500 wavelengths $\times$ 1000 test velocities $\times$ 3136 bytes $\approx$ 1.6 GB, close to the memory limit of the GPU.

Chapter 4

Conclusions and Future Work

In this thesis, we have described the design and implementation of efficient algorithms for two commonly used methods in geophysical imaging: cross-correlations on wavelet-compressed data, and Multichannel Analysis of Surface Waves (MASW). Both of these methods are frequently used on DAS data, and our algorithms address some of the challenges posed by geophysical data used in this context, although they can be used on general data collected by other sources as well.

For the first method, we have designed an algorithm for computing cross-correlations of data stored in the wavelet-domain by a discrete wavelet transform (DWT), without requiring the use of an inverse transform to reconstruct the signal. We have tested implementations that use both dense DWT coefficients, and thresholded sparse coefficients. The algorithm avoids the costs of an inverse transform, and appears to scale better than typical cross-correlation computations when making use of sparsity. In designing the algorithm, we have also developed theory that may be useful for wavelet analysis in other contexts: the wavelet properties pertaining to cross-correlations, and error bounds of wavelet thresholding on cross-correlations.

For the second method, we designed algorithms for MASW using MPI and CUDA, which can take advantage of multiple CPU cores and GPUs, respectively. The MPI implementation has been found to scale near-linearly with smaller numbers of computing cores, and requires minimal communication between processes. The GPU implementation also sees marked

improvement in performance compared to serial CPU implementations, while still possessing linear scalability on larger problem sizes. In addition, it features a linear algebra operation that is currently unavailable in existing libraries for CUDA: a determinant solver for several small, banded matrices.

4.1 Future Work

The wavelet-domain cross-correlation algorithm is currently implemented in Python, primarily utilizing NumPy for vectorized linear algebra operations, PyWavelet for DWTs, and custom-written C code for performing sparse cross-correlations. There are existing GPU algorithms that perform the DWT, so a GPU implementation of wavelet-domain cross-correlation could be part of an efficient pipeline for wavelet compression and analysis of data. However, while the development of properties for the correlation matrix $W^{(\tau)}$ is useful for this serial implementation, any representation of $W^{(\tau)}$ for use on a GPU would require additional work.

Another, less ambitious enhancement is tailoring the wavelet-domain algorithm for specific problems. Our test cases show that individual levels of wavelet or scaling functions in a DWT are ideal for representing particular frequency components of a signal. If we are interested in only a specific frequency band in the data (for example, we are looking for a particular event that is active at only certain frequencies), then we could choose an appropriate wavelet family and modify our cross-correlation algorithm so it only computes cross-correlations for the levels at which the desired frequency occur. This can reduce the total volume of necessary storage for the compressed data and further improve the algorithm's speed, but it is dependent on the particular use case and data.

Lastly, in a more theoretical direction, there is the possibility to generalize or modify the

properties used to efficiently store $W^{(\tau)}$ to work with mathematical structures similar to wavelets, such as curvelets. This may allow us to perform efficient cross-correlations on data better preserved by alternatives to wavelet compression. Similarly, some of the concepts used to prove the properties of $W^{(\tau)}$ may also inform design and implementation choices for machine learning algorithms that can take wavelet-domain data as inputs, such as hierarchical neural networks.

For the implementations of MASW, there are alternative means to asses the singularity of the matrices generated by the stiffness matrix method. The current implementation uses determinants to predict which test velocities make these matrices near-singular. Since the matrices are heptadiagonal, it is cheap to compute their determinants by Gaussian elimination. However, an implementation that uses singular values instead to determine when these matrices approach singularity may have some advantages. We know the matrices are themselves functions of the selected range of test velocities and the parameters M of the proposed near-surface model. We can analytically solve for the partial derivatives of these matrices relative to M, and then find the partial derivatives of the matrix singular values relative to M [5].

Once we have the singular value derivatives, we then know what changes to M can make our resulting singular values approach zero at test velocities closer to the velocities of the curve computed in MASW dispersion. This in effect can give MASW a form of backpropagation, allowing us to determine what modifications to M can reduce the misfit of the theoretical dispersion curve. Making effective use of the singular value derivatives presents a significant challenge, since the matrix entries are not linear functions of our model parameters. Moreover, the determinant based approach is already only $O(N)$ for each (small) matrix in the method. However, developing a potential backpropagation for MASW may help optimize the outer loop of selecting which near-surface models to evaluate.

4.2 Available Software

The MASWAccelerated software, along with examples to produce the results in this paper, are publicly available at https://github.com/jlk9/MASWA under an MIT license. The MASWAccelerated code was first made publicly available in 2020 upon submission of [9]. Hardware and software requirements, as well as other code features, are available in the repository's README file.

Software for the wavelet-domain cross-correlation algorithms will be publicly available on a GitHub repository soon.

Bibliography

[1] S Grace Chang, Bin Yu, and Martin Vetterli. Adaptive wavelet thresholding for image denoising and compression. *IEEE transactions on image processing*, 9(9):1532–1546, 2000.

[2] Chul Hwan Kim and Raj Aggarwal. Wavelet transforms in power systems. I. General introduction to the wavelet transforms. *Power Engineering Journal*, 14(2):81–87, 2000. doi: 10.1049/pe:20000210.

[3] Fergal Cotter and Nick Kingsbury. Deep Learning in the Wavelet Domain, 2018.

[4] S. Dou, N. Lindsey, A.M. Wagner, T.M. Daley, B. Freifeld, M. Robertson, J. Peterson, C. Ulrich, E.R. Martin, and J.B. Ajo-Franklin. Distributed acoustic sensing for seismic monitoring of the near surface: A traffic-noise interferometry case study. *Scientific Reports*, 7:article no. 11620, 2017.

[5] Juan-Miguel Gracia. Directional derivatives of the singular values of matrices depending on several real parameters, 2020.

[6] E. Kausel and J.M. Roësset. Stiffness matrices for layered soils. *Bulletin of the Seismological Society of America*, 71(6):1743–1761, 1981.

[7] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[8] S. Klimenko, G. Mitselmakher, and A. Sazonov. A Cross-Correlation Technique in Wavelet Domain for Detection of Stochastic Gravitational Waves, 2002.

[9] Joseph Kump and Eileen R. Martin. Multichannel Analysis of Surface Waves Accelerated (MASWAccelerated): Software for Efficient Surface Wave Inversion Using MPI and GPUs, 2020.

[10] Nathaniel J. Lindsey and Eileen R. Martin. Fiber-Optic Seismology. *Annual Review of Earth and Planetary Sciences*, 49(1), 2021. doi: 10.1146/annurev-earth-072420-065213. URL https://doi.org/10.1146/annurev-earth-072420-065213.

[11] J.N. Louie. Faster, better: Shear-wave velocity to 100 meters depth from refraction microtremor arrays. *Bulletin of the Seismological Society of America*, 91(2):347–364, 2001. doi: 10.1785/0120000098.

[12] E.Á. Ólafsdóttir, S. Erlingsson, and B. Bessason. Tool for analysis of multichannel analysis of surface waves (MASW) field data and evaluation of shear wave velocity profiles of soils. *Canadian Geotechnical Journal*, 55(2):217–233, 2018. doi: 10.1139/cgj-2016-0302.

[13] C.B. Park, R.D. Miller, and J. Xia. Multichannel analysis of surface waves. *Geophysics*, 64(3):800–808, 1999. doi: 10.1190/1.1444590.

[14] C.B. Park, R.D. Miller, J. Xia, and J. Ivanov. Multichannel analysis of surface waves (MASW)- active and passive methods. *The Leading Edge*, 26(1):60–64, 2007.

[15] X. Song, H. Gu, X. Zhang, and J. Liu. Pattern search algorithms for nonlinear inversion of high-frequency Rayleigh-wave dispersion curves. *Computers & Geosciences*, 34(6): 611–624, 2008.

[16] X. Song, H. Gu, L. Tang, S. Zhao, X. Zhang, L. Li, and J. Huang. Application of artificial bee colony algorithm on surface wave data. *Computers & Geosciences*, 83: 219–230, 2015.

[17] E.M. Stein and R. Shakarchi. *Fourier Analysis: An Introduction*. Princeton University Press, 2003. ISBN 9780691113845. URL https://books.google.com/books?id=I6CJngEACAAJ.