Abstract

Fourier data-analysis represents data as a linear combination of sinusoidal waves with different frequencies. Each wave has a corresponding coefficient, calculated using the discrete Fourier transform (DFT). Algorithms play a major role in Fourier analysis, in particular the Fast Fourier Transform (FFT). The FFT reduces the number of operations for calculating a length $n$ DFT from $O(n^2)$ to $O(n \log n)$. Now, taking a 1D FFT of a length $n$ time-series assumes stationarity. For non-stationary time-series, practitioners analyze how frequencies change over time using time-frequency representations. A time-frequency representation calculates successive DFTs along a sliding window of the data, called a Sliding Window Fourier Transform (SWFT). Like the DFT, algorithms for calculating the SWFT play a major role. Notably, two $O(nm)$ algorithms exist for the SWFT, where $n$ is the data-length and $m$ is the window size, faster than taking an FFT in each window. For reasons articulated here, we focus on the second algorithm, called the Fast Sliding Window Fourier Transform. Today, many applications of Fourier analysis extend beyond 1D. Like 1D, we get around the stationarity assumption with a $k$-dimensional SWFT, the focus of this thesis. As preliminary work, we have extended the 1D Fast SWFT algorithm to 2D. We propose further extending the algorithm to $k$D, and give a detailed plan for this extension here. In addition, Okamura (2011) recently derived statistical properties for the 1D SWFT, which we will extend to $k$D. Finally, we identify various applications either using the $k$D SWFT explicitly, or could easily use it. We will use the $k$D SWFT in these applications, showing that it increases both the computational and statistical efficiency.
1 Introduction

The Fourier Transform dates back to the early 19th century (Fourier (1822)), when Joseph Fourier discovered how to decompose a function into sine and cosine waves with different frequencies. Analysis in the “frequency domain” has a long history (Schuster (1898)), and is widely used across science and engineering (Bracewell (1986)). In addition, various mathematical and computational advantages come from this frequency representation (e.g Cooley and Tukey (1965)). So, from a scientific, mathematical, and computational perspective, the Fourier transform is an indispensable tool.

Different types of Fourier transforms exist for the discrete and continuous settings (Chapter 2.4 of Vetterli and Kovačević (2007)). In data-analysis, we observe a discrete set of observations, and can only estimate a discrete set of frequencies. Therefore, our primary tool for Fourier data-analysis is the discrete Fourier transform (DFT). Quickly explained, the DFT calculates weights for a discrete set of sine and cosine waves with different frequencies. A linear combination of these waves with their corresponding weights gives an equivalent representation of the data.

A landmark moment in Fourier analysis occurred in 1965, when Cooley and Tukey (1965) published a fast algorithm for calculating the DFT, the Fast Fourier Transform (FFT). The FFT calculates the DFT of a length $n$ dataset with $O(n \log n)$ operations, a large improvement over a straightforward $O(n^2)$ calculation. The history of the FFT is fascinating, James et al. (1967) provided an initial survey, and Heideman et al. (1985) later pointed out the idea goes all the way back to Gauss. The FFT algorithm cleared the way for Fourier analysis of large datasets, and SIAM ranks the FFT as a top ten algorithm of the 20th century (Cipra (2000)).

From a statistical perspective, taking the DFT of a 1D time-series assumes stationarity. With non-stationary data, Gabor (1946) first introduced a time-frequency representation. A time-frequency representation shows how frequency components change over the course of a time-series, by trading frequency resolution for time resolution. Some example time-series and corresponding time-frequency representations are shown in Figure 1.

Time-frequency representations are constructed in different ways, one of the most important being the Sliding Window Fourier Transform (SWFT). In the 1D case, the SWFT takes repeated DFTs of a time-series multiplied by a window function, letting the window slide along the time-axis. The window function is typically rectangular, giving a subset of points in a contiguous window. So, the SWFT simply takes a sequence of DFTs in a sliding window. Of course, replacing the DFT in each window with an FFT gives a large computational advantage.

Beyond taking FFTs in each window, researchers began noticing faster algorithms for the SWFT. The first proposal was Aravena (1990), who gave a recursive, linear-time algorithm explained in Section 2.1.4. Around the same time, Covell and Richardson (1991) proposed a non-recursive algorithm using repeated FFT calculations in overlapping windows. Recently, Wang et al. (2009) independently discovered a fast, non-recursive algorithm similar to Covell and Richardson (1991). This thesis focuses on the non-recursive algorithm from Wang et al. (2009), called the Fast Sliding Window Fourier Transform (FSWFT).

Today, many applications of Fourier data-analysis extend beyond 1D. In particular, Fourier methods are used for fast convolutions (Mathieu et al. (2013)), signal and image processing (Lim (1990)), forensics (Vorburger et al. (2007)), and more (Kurnaz et al. (2007); Meraoumia et al. (2010)). Since Fourier transforms in higher dimensions involve more data,
fast algorithms are key to their success.

This thesis proposes extending the FSWFT algorithm to $k$-dimensions. In addition, we will explore the $kD$ SWFT as data-analysis tool, by deriving its statistical properties (Okamura (2011)) and using it in applications.

The rest of the document is organized as follows. Section 2 introduces notation and key ideas for the $kD$ algorithm. In addition, applications of the 1D SWFT and statistical properties are discussed. Section 3 introduces the 2D FSWFT algorithm, our primary result so far. Finally, Section 4 provides research objectives for this thesis. Specifically, we discuss the $kD$ algorithm, its software implementation, and the $kD$ SWFT for data-analysis.

2 Background and Notation

This section sets notation for the thesis, and introduces key ideas for the new algorithm. Section 2.1 defines the 1D Sliding Window Fourier Transform (SWFT). In addition, four al-
algorithms for calculating the 1D SWFT are discussed: taking a DFT in each window, taking an FFT in each window, the Recursive SWFT, and the Fast SWFT. Next, a motivating application of the 1D SWFT for detecting local periodic signals is described. Finally, statistical properties and methods for the 1D SWFT described in Okamura (2011) are summarized.

2.1 Sliding Window Fourier Transform

This section defines the 1D Sliding Window Fourier Transform (SWFT), and gives four algorithms for calculating it. We start by defining the discrete Fourier transform (DFT), then the Fast Fourier Transform (FFT) algorithm for calculating the DFT. Section 2.1.2 discusses a key concept: representing the FFT using a tree data-structure. Finally, we define the SWFT and detail the two fastest algorithms for calculating it.

2.1.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) converts a real or complex valued, length $n$ input signal into a set of $n$ complex coefficients. Each coefficient represents a sine and cosine wave for a specific frequency. Let $x = (x_0, x_1, \ldots, x_{n-1})$ be a length $n$ complex-valued signal. The DFT of $x$ is:

$$ a_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega_n^{-jk} $$

For $k = 0, 1, \ldots, n-1$. $\omega_n$ is the principal $n^{th}$ root of unity of Euler’s formula:

$$ \omega_n = \exp\left(\frac{i2\pi}{n}\right) = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right) $$

So, the DFT converts a length $n$ signal into $n$ complex-coefficients. These coefficients correspond to paired sine and cosine waves with frequencies $\frac{0}{n}, \frac{1}{n}, \ldots, \frac{n-1}{n}$. These frequencies are called “Fourier Frequencies”, and form an orthogonal basis for $n$-dimensional space.

2.1.2 Fast Fourier Transform

DFT calculations are sped up using the Fast Fourier Transform (FFT) algorithm. The FFT reduces the number of operations for calculating a length $n$ DFT from $O(n^2)$ to $O(n \log n)$. The FFT algorithm was published by Cooley and Tukey (1965), and had an immediate (Cochran et al. (1967)) and lasting (Cooley (1990)) impact on science and engineering. Appendix A derives the FFT algorithm, showing how it saves computational time. This section shows how the FFT can be represented using a tree data-structure.

Before introducing the tree-structure, we point out a few specifics regarding the FFT algorithm in this thesis. First, While many different FFT algorithms have been developed (e.g. Rabiner et al. (1969); Good (1958)), this thesis focuses on the original Cooley-Tukey algorithm. This means that whenever we say FFT, we are referring to the Cooley-Tukey algorithm. In addition, different algorithms exist for different sizes of $n$. The most widely used FFT implementation comes when $n$ is a power of two, known as the Radix-2 case. In fact, the Radix-2 case was fully derived in Cooley and Tukey (1965), since it is particularly efficient for binary computers. This thesis focuses on the Radix-2 case, although the ideas can be extended to different Radices.
Figure 2: A butterfly diagram representing the FFT algorithm when $n = 8$. The circles on the far left are the input data, and the circles on the far right are the DFT coefficients. Multiplication takes place at the beginning of each arrow, and addition takes place at the end. For example, the first calculation on the bottom left means that $x_7$ is multiplied by $\omega^4$, then this quantity is added to $x_3$.

FFT algorithms have been deeply studied (Brigham et al. (1988)), and have many different representations. For example, the derivation in Appendix A gives an algebraic representation, and Van Loan (1992) makes a convincing case for always using a matrix-vector representation. For our purposes, the most important FFT representation is the butterfly diagram shown in Figure 2. The butterfly diagram was first introduced by Weinstein (1969), in order to show intermediate FFT calculations when $n = 8$. For us, the butterfly diagram is important because it can be re-drawn into our tree data-structure. Figure 3 shows the corresponding tree-data structure for the butterfly diagram when $n = 8$. It is important to stress that the calculations in Figures 2 and 3 are equivalent: both give the DFT of a length $n = 8$ input signal, with the exact same calculations.

Figure 3: The butterfly diagram re-drawn as a tree data-structure
There are two major benefits of the tree data-structure for this thesis. First, the input data is now ordered from $x_0$ on the far left of Figure 3 to $x_7$ on the far right. Second, underneath $x_7$ is a binary tree, with 3 (since $2^3 = 8$) levels. At the bottom of this binary tree are the 8 DFT coefficients. As we will show in Section 2.1.5, if we want the DFT of the next window with points $(x_1, x_2, \ldots x_8)$, the number of operations required is just the size of the binary tree. And, since the size of the tree is $O(n)$ (compared with $O(n \log n)$ for the FFT), this leads to the Fast SWFT algorithm.

### 2.1.3 Sliding Window Fourier Transform

Mentioned earlier, taking the DFT of an entire 1D time-series assumes stationarity. With non-stationary data, practitioners want to know how the frequency decomposition of the time-series changes over time, called a time-frequency representation. Gabor (1946) first introduced a method for calculating time-frequency representations, which we call the Sliding Window Fourier Transform (SWFT). Quickly explained, the SWFT takes a sequence of DFTs in a contiguous moving window of the time-series, resulting in a 2D representation of the data: one for time, the other for frequency. The SWFT is known by a variety of names: the Gabor transform, windowed Fourier transform, Short Time Fourier Transform, local Fourier transform, etc. Since all of these names correspond to the same underlying method, this thesis uses “SWFT” for clarity. Next, we define the 1D SWFT.

Let $x$ be a length $N$ complex valued signal, $(x_0, x_1, \ldots, x_{N-1})$, and let $n \leq N$ be the window size. This implies the SWFT takes the DFT in $W = N - n + 1$ total windows. Let $w$ index the window position, where $w = n - 1, n, \ldots, N - 1$. The 1D SWFT is:

$$a^w_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_{\hat{w} + j} \omega_n^{-jk}$$

where $\hat{w} = w - n + 1$, , and $k = 0, 1, \ldots, n-1$. The SWFT results in an $n \times W$ dimensional array, where the $n$-dimension corresponds to frequencies, and the $W$-dimension corresponds to time.

A straightforward calculation of Equation 1 takes $Wn^2$ operations. Replacing the DFT in each window with the FFT reduces this to $O(Wn \log n)$ operations. Since there are $Wn$ DFT coefficients, the lower computational bound is $Wn$. So, taking the FFT in each window is a log factor away from the lower bound. Next, we give two $O(Wn)$ algorithms for the SWFT, reaching the lower bound.

### 2.1.4 Recursive Sliding Window Fourier Transform

Many authors have discovered a fast, recursive algorithm for calculating the SWFT. Aravena (1990) first introduced this algorithm in a paper titled “Recursive Moving Window DFT Algorithm”. Shortly after, Lilly (1991) and Sherlock and Monro (1992) presented the same algorithm, calling it the “Moving Discrete Fourier Transform”. Since then, (Albrecht et al. (1997), Macias and Exposito (1998), Albrecht and Cumming (1999), Exposito and Macias (1999), Expósito and Macfias (2000), Albrecht (2002)) have all used or improved upon the original recursive algorithm. Jacobsen and Lyons (2003) give an overview of the recursive algorithm, in an article titled “The Sliding DFT”.

5
The two key ideas of the recursive algorithm are that data overlaps between windows, and the Fourier shift theorem. It is easy to see that data-overlaps between windows, since the SWFT takes a sequence of DFTs with inputs: \((x_0, x_1, \ldots, x_{n-1}), (x_1, x_2, \ldots, x_n), \ldots, (x_{N-n+1}, x_{N-n}, \ldots, x_{N-1})\). The Fourier shift theorem (see Appendix E) then gives a recursive formula relating the DFT coefficients for successive windows. The formula is:

\[ a_{w+1}^k = \omega_n^k (a_w^k + x_{w+1} - x_{w-n+1}) \]  

(2)

where \(k = 0, 1, \ldots n - 1\). In English, Equation 2 says that the \(k^{th}\) DFT coefficient in window position \(w+1\) equals the \(k^{th}\) coefficient in window position \(w\), plus the data-point entering the window, minus the data-point leaving the window, all multiplied by the Fourier shift factor: \(\omega_n^k\). In addition, Equation 2 can be extended for a shift of \(s > 1\):

\[ a_{w+s}^k = \omega_n^{ks}(a_w^k \sum_{j=0}^{s-1} (x_{w+1+j} - x_{w-n+1+j}) \omega_n^{-jk}) \]  

(3)

Equations 2 and 3 show that if we know the \(n\) DFT coefficients in previous windows, calculating each of the \(n\) DFT coefficients in the current window has an analytical solution. This means that each window takes \(O(n)\) operations, making the recursive algorithm \(O(Wn)\), plus the additional \(O(n \log n)\) operations required for calculating the FFT in the first window.

Unfortunately, the recursive algorithm suffers from a variety of issues. First, Equation 3 shows the algorithm is only applicable when \(s < n\). More importantly, the recursive algorithm is numerically unstable. In fact, Covell and Richardson (1991) proved that the variance of the error using the recursive algorithm is unbounded, and grows linearly with input size \(N\). Figure 4 demonstrates the numerical error of the 1D recursive algorithm by showing error as a function of the window position. We see that numerical error grows as the window position increases, implying that the numerical error problem grows worse with larger data.

Both Douglas and Soh (1997) and Duda (2010) proposed adaptations to the recursive algorithm, avoiding this numerical instability. In contrast, the algorithm described next (also shown in Figure 4) is fast, numerically stable and, as we will see: easier to extend to higher dimensions.

### 2.1.5 Fast Sliding Window Fourier Transform

In the early 90’s, both Covell and Richardson (1991) and Farhang and Lim (1992) proposed non-recursive, stable algorithms for the SWFT. The authors discovered their new algorithms by noticing repeated calculations in the butterfly diagrams (see Figure 2) for successive windows. Since then, Montoya-Andrade et al. (2012) proposed further refinements to this algorithm. Recently, an algorithm similar to Covell and Richardson (1991) was independently discovered by Wang et al. (2009). This thesis focuses on the algorithm discovered by Wang et al. (2009), called the Fast Sliding Window Fourier Transform (FSWFT).

The key to the FSWFT is the tree data-structure of the FFT shown in Figure 3, and introduced in Section 2.1.2. For the FSWFT, we need to calculate this FFT tree for each window of the data. The FSWFT’s speed comes from the fact that overlapping windows repeat FFT calculations. So, by storing FFT calculations for each window in binary trees, the FSWFT avoids repeated calculations by instead looking up values from previous trees.
Figure 4: The numerical error of the Recursive and Fast SWFT algorithms. Here, the data size is 10,000, and the window size is 64. We measure numerical error using the mean squared error different of the Squared Modulus statistic between each algorithm and our reference algorithm: taking the DFT in each window.

Figure 5: An illustration of how the FSWFT algorithm removes repeated calculations by storing intermediate calculations in binary trees. The green arrows correspond to taking the FFT of the first window, and the red arrows correspond to the FFT the next window. The second (red) FFT takes less operations, since it looks up values calculated in green.

An illustration of the FSWFT is given in Figure 5. Figure 5 shows how the FSWFT algorithm calculates the first two, length 8 windows. The first windows input is: \((x_0, x_1, \ldots, x_7)\), and first windows FFT calculations are shown in green. The bottom of the binary tree underneath \(x_7\) contains the DFT coefficients for the first window. The second windows input is: \((x_1, x_2, \ldots, x_8)\), since the window slides one position to the right. The FFT calculations of the second window are shown in red, and the binary tree underneath \(x_8\) contains the DFT coefficients for the second window. Figure 5 shows that calculating the FFT in the second window uses calculations from the first window, and that the number of calculations for the second window equals the binary tree. The difference in operations in the first and second windows is the log factor speed-up of the FSWFT algorithm. A complete derivation of the 1D FSWFT algorithm and implementations details are given in Appendix B.

The number of operations used by the FSWFT grows linearly with window size and signal length. To see this, say we have a length \(N\) time-series and window size \(n = 2^p\), implying
there are \( W = N - n + 1 \) windows. Next, we define an operation the same as Cooley and Tukey (1965): a complex multiplication followed by a complex addition. Appendix B shows that the calculation at each node takes one operation. Since there are \( p = \log_2(n) \) levels in the tree, and each level has \( 2^p \) nodes, the computational complexity for each window is:

\[
C_{\text{each}} = \sum_{l=1}^{p} 2^l = 2(2^p - 1) = 2(n - 1)
\] (4)

Of course, the number of windows \( (W) \) grows linearly with \( N \). Therefore, the FSWFT is \( O(W n) \), the same speed as the recursive algorithm.

The exact number of operations used by the FSWFT is more complicated, since the first \( l < n \) data-points do not require complete trees. Figures 3 and 5 show that calculating the FFT in the first window gives all the intermediate calculations required for future windows, so the exact number of operations is \( n \log_2(n) + W/2(n - 1) \). Table 1 summarizes the speed of the four different SWFT algorithms discussed in this section, showing that RSWFT and FSWFT are the fastest.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( C_{\text{extra}} )</th>
<th>( C_{\text{each}} )</th>
<th>( C_{\text{total}} )</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWDFT</td>
<td>0</td>
<td>( n^2 )</td>
<td>( W n^2 )</td>
<td>( O(W n^2) )</td>
</tr>
<tr>
<td>SWFFT</td>
<td>0</td>
<td>( n \log(n) )</td>
<td>( W n \log(n) )</td>
<td>( O(W n \log(n)) )</td>
</tr>
<tr>
<td>RSWFT</td>
<td>( n \log(n) )</td>
<td>( n \log(n) + Wn )</td>
<td>( O(Wn) )</td>
<td></td>
</tr>
<tr>
<td>FSWFT</td>
<td>( n \log(n) )</td>
<td>( 2(n - 1) )</td>
<td>( n \log(n) + W/2(n - 1) )</td>
<td>( O(Wn) )</td>
</tr>
</tbody>
</table>

Table 1: The number of operations used by each of the four 1D SWFT algorithms. \( C_{\text{extra}} \) means the number of operations before the first window, \( C_{\text{each}} \) means the number of operations per window, and \( C_{\text{total}} \) means the number of total operations. The final column gives the number of operations in Big O notation.

### 2.2 1D Application: Local Signal Detection

The FSWFT’s original motivation is summarized in a quote from Wang et al. (2009), “It is believed that immediately prior to an epileptic seizure, the brain generates short term (less than a second) high frequency oscillations (HFO’s) at hundreds of Hz”. The authors measured electrical currents in the brain using Magnetoencephalography (MEG), and needed thousands of DFTs per second for capturing these frequencies. From the authors: “we would like to compute \( 306 \times 1000 \) Fourier transforms for each second of collected data. A single 50 minute experiment requires more than a billion of such transforms”. These quotes demonstrate that a fast algorithm for the SWFT will be very useful, even necessary, and Wang et al. (2009) derived the FSWFT to solve this problem. Although HFOs were not discovered, the authors detected previously unknown signals at higher frequencies.

### 2.3 1D Statistics

The 2011 PhD thesis of Okamura (2011) worked on statistical properties of the 1D SWFT. The key insight is that a 1D SWFT gives a multivariate time-series, with one series for each frequency. Okamura’s work covered two main themes: statistical properties of the 1D SWFT,
and methods for local signal detection. For the statistical properties, Okamura derived analytical expressions such as the cross-covariance between time-series representing different frequencies. In addition, Okamura showed how the squared modulus statistic is ideal for detecting local periodicities (see bottom of Figure 1). Next, Okamura developed methods for detecting local periodic signals for time-series. In particular, Okamura computed the probability that the squared modulus statistic exceeded a threshold for consecutive observations. The Box-Cox transformation and delta method were used to approximate this probability for a Gaussian stationary, and AR(1) process. This approximation was accurate compared with a Monte-Carlo numerical simulation method.

The work of Okamura presents the 1D SWFT as a data-analysis tool, in this case used for detecting local periodic signals. We plan to extend the SWFT to higher dimensions, for use as a data-analytic tool.

3 The 2D Fast Sliding Window Fourier Transform

This thesis will extend the FSWFT to kD. So far, we have derived the 2D FSWFT, explained in this section. First, this section defines the 2D DFT, FFT, and SWFT. In addition, this section illustrates how the ideas for the 2D extension carry over to kD. Finally, two 2D applications are presented.

3.1 The 2D DFT and FFT

The 2D DFT is a straightforward extension of the 1D DFT. Let \( x \) be an \( N_0 \times N_1 \) dimensional array. Then, the 2D DFT is:

\[
a_{k_0,k_1} = \frac{1}{\sqrt{N_0N_1}} \sum_{j_0=0}^{N_0-1} \sum_{j_1=0}^{N_1-1} x_{j_0,j_1} \omega_{N_0}^{-j_0k_0} \omega_{N_1}^{-j_1k_1}
\]

where \( k_0 = 0,1,\ldots,N_0 - 1 \) and \( k_1 = 0,1,\ldots,N_1 - 1 \).

Since there are \( N_0N_1 \) coefficients, and each coefficient takes \( N_0N_1 \) operations, Equation 5 takes \( N_0^2N_1^2 \) operations. Fortunately, the same idea underlying the FFT described in Appendix A leads to a 2D FFT. Re-writing Equation 5:

\[
a_{k_0,k_1} = \frac{1}{\sqrt{N_0N_1}} \sum_{j_0=0}^{N_0-1} \omega_{N_0}^{-j_0k_0} \sum_{j_1=0}^{N_1-1} x_{j_0,j_1} \omega_{N_1}^{-j_1k_1}
\]

The inner sum is a length \( N_1 \) DFT over rows of the 2D array. Re-writing this inner sum as \( z_{j_0,k_1} \) gives:

\[
a_{k_0,k_1} = \frac{1}{\sqrt{N_0N_1}} \sum_{j_0=0}^{N_0-1} z_{j_0,k_1} \omega_{N_0}^{-j_0k_0}
\]

which is a length \( N_0 \) DFT over the columns of \( z_{j_0,k_1} \). Of course, both 1D DFTs can be calculated using the FFT. So, the 2D FFT takes \( N_0 \) length \( N_1 \) FFTs of each row, followed by \( N_1 \) length \( N_0 \) FFTs of the corresponding columns. Overall, the 2D FFT takes \( N_0N_1 \log(N_1) + N_1N_0 \log(N_0) = N_0N_1 \log(N_0N_1) \) operations, a substantial improvement over the 2D DFT.
Figure 6: An example of the 2D SWFT, where $N_0 = N_1 = 8$, $n_0 = n_1 = 4$, and $W_0 = W_1 = 5$. The red points indicate the bottom-right point of all rectangular windows. The upper left window outlined in blue has window position $(w_0, w_1) = (3, 3)$, whereas the bottom-right window has window position $(7, 7)$.

### 3.2 The 2D Sliding Window Fourier Transform

This section defines the 2D SWFT. To set notation, let $x$ be a $N_0 \times N_1$ dimensional array, and we want the 2D DFT of each $n_0 \times n_1$ dimensional window, where $n_0 \leq N_0$ and $n_1 \leq N_1$. Let $W_0$ be the number of windows in the $x$-direction, and $W_1$ the number of window in the $y$-direction. Then, $W_0 = N_0 - n_0 + 1$ and $W_1 = N_1 - n_1 + 1$. Finally, let $(w_0, w_1)$ index the window position, indicating the bottom-right points, where $w_0 = n_0 - 1, n_0, \ldots, N_0 - 1; w_1 = n_1 - 1, n_1, \ldots, N_1 - 1$. For example, Figure 6 shows a 2D array where $N_0 = N_1 = 8$, $n_0 = n_1 = 4$, and $W_0 = W_1 = 5$. The upper-left window outlined in blue is indexed by $(w_0, w_1) = (3, 3)$.

With this notation, the 2D SWFT definition is:

$$a_{k_0, k_1}^{w_0, w_1} = \frac{1}{\sqrt{n_0 n_1}} \sum_{j_0 = 0}^{n_0 - 1} \sum_{j_1 = 0}^{n_1 - 1} x_{\hat{w}_0, \hat{w}_1} e^{-j\hat{w}_0 k_0 n_0^{-1} \omega_n - j\hat{w}_1 k_1 n_1^{-1}}$$

(6)

$$\hat{w}_0 = w_0 - n_0 + 1; \hat{w}_1 = w_1 - n_1 + 1$$

$$k_0 = 0, 1, \ldots, n_0 - 1; k_1 = 0, 1, \ldots, n_1 - 1$$

(7)

The output of the 2D SWFT is a 4D array, with dimensions $n_0 \times n_1 \times W_0 \times W_1$. Since there are $W_0 W_1$ windows, a straightforward calculation of Equation 6 takes $W_0 W_1 n_0^2 n_1^2$ operations. Replacing the 2D DFT with the 2D FFT in each window reduces the number of operations to $W_0 W_1 n_0 n_1 \log(n_0 n_1)$, a large improvement. Since the output-array is size $n_0 \times n_1 \times W_0 \times W_1$, the lower bound for the 2D SWFT is $n_0 n_1 W_0 W_1$ operations.
Recently, two $O(n_0 n_1 W_0 W_1)$ algorithms have been proposed for the 2D SWFT. For the first, Park (2015) extends the 1D recursive SWFT algorithm to 2D, derived in Appendix C. Park’s 2D algorithm is fast, but suffers from the same numerical error problem as the 1D recursive algorithm (see Figure 9). Second, Byun et al. (2016) propose an algorithm avoiding numerical error based on the Vector Radix (Rivard (1977); Harris et al. (1977)) algorithm. Byun’s algorithm is both fast and avoids numerical error, but is only applicable for square (not rectangular) windows. Next, we extend the 1D FSWFT algorithm to 2D, which is fast, numerically stable, and works for rectangular window sizes.

3.3 The 2D Fast Sliding Window Fourier Transform

The key to the 2D FSWFT is the tree data-structure of the 2D FFT (shown in Figure 7), and the separability of the 2D FFT by dimension (e.g. rows and columns). Figure 7 illustrates how the 2D FFT works: first take 1D FFTs of each row, then take 1D FFTs of the corresponding columns. Separating the 2D FFT into 1D row and column FFTs is a key idea for extending algorithm to 2D.

Like 1D, the 2D FSWFT algorithm gains its speed by storing intermediate 2D FFT calculations in previous windows, and looking up the calculations instead of re-computing them. Figure 8 shows how the 2D FSWFT algorithm works for two successive windows. The green lines show the calculations used to compute the 2D FFT in the first window, and the red lines show the calculations in the second window. The number of red lines equals the size of the binary tree, since all other calculations are available from the previous window. The difference between the number of green and red arrows corresponds is the speed-up.

Several factors must be considered for extending the FSWFT algorithm to 2D. First, previous calculations can come from either the row or column dimension. Specifically, the first $p_1 = \log_2(n_1)$ levels look-up values from the same row, called row-levels. The final $(p_1 + 1)$ to $p_1 + p_0$ levels look-up values from the same column, called column levels. Another key difference from 1D is tracking two complex-exponential vectors: $\Omega^0 = (\omega_{n_0}^0, \omega_{n_0}^{-1}, \ldots, \omega_{n_0}^{-(n_0-1)})$, and $\Omega^1 = (\omega_{n_1}^0, \omega_{n_1}^{-1}, \ldots, \omega_{n_1}^{(n_1-1)})$. $\Omega^1$ works just like the 1D case, but $\Omega^0$ needs to be adjusted. The easiest way to see this is by considering the $2 \times 4$ example in Figure 7. Here, $\Omega_1 = (\omega_1^0, \omega_1^{-1}, \omega_1^{-2}, \omega_1^{-3})$ and $\Omega^0 = (\omega_2^0, \omega_2^{-1})$. To calculate the third level of the binary tree, we need 8 values from the $\Omega_0$, which is only length 2. This is solved by expanding $\Omega^0$ into $\Omega^0 \otimes 1_{n_1}$, where $1_{n_1}$ is an $n_1$-vector of ones. With these differences in mind, we derive the
Figure 8: The 2D FSWFT algorithm for a window moving one column to the right. The green arrows show the calculations made in the previous window, and the red arrows show the calculations made in the current window. This shows that the 2D FSWFT achieves its speed-up by removing repeated calculations, and instead looking them up from previous trees.

2D FSWFT algorithm next.

Say we have a length $N_0 \times N_1$ array, with window sizes $n_0 = 2^{p_0}$, $n_1 = 2^{p_1}$. This means we have $p_0$ row-levels, and $p_1$ column-levels, and $p_0 + p_1$ total levels in the binary trees. Let $\Omega^0 = (\omega_{n_0}^0, \omega_{n_0}^{-1}, \ldots, \omega_{n_0}^{(n_0-1)})$ and $\Omega^1 = (\omega_{n_1}^0, \omega_{n_1}^{-1}, \ldots, \omega_{n_1}^{(n_1-1)})$ be the complex exponential vectors. Finally, let $T$ be our tree data-structure. We access elements of $T$ by window position, level, and node. For example, $T_{w_0, w_1, l, i}$ corresponds to the $i^{th}$ node in the $l^{th}$ level of the tree at position $(w_0, w_1)$.

We derive the calculations required to calculate the binary tree for an arbitrary window position $(w_0, w_1)$. Level zero of the tree is the data, so $T_{w_0, w_1, 0, 0} = x_{w_0, w_1}$. For the remaining levels, we use two additional pieces of information. First, let $q_l$ be the number of nodes at level $l$ of the tree, so $q_l = 2^l$ for all levels. Next, let $s_l^d$ be the “shift”, representing the distance between the current tree at position $(w_0, w_1)$ and the tree with the repeated calculations. The value of $s_l^d$ depends on both the level $l$ and the dimension $d$. For row levels $s_l^1 = \frac{n_1}{q_l}$, and for column levels $s_l^0 = \frac{n_0}{q_l - p_1}$.

For level one, $q_1 = 2^1 = 2$, so $s_1^1 = \frac{n_1}{2}$. This means we need to calculate two nodes, and the repeated calculations are contained in the tree shifted in the row-direction by $\frac{n_1}{2}$. The calculations for level one are:

\[
T_{w_0, w_1, 1, 0} = T_{w_0, w_1, 1, 0} + s_0^l T_{w_0, w_1, 0, 0} \quad \text{mod } q_0
\]

\[
T_{w_0, w_1, 1, 1} = T_{w_0, w_1, 1, 1} + s_1^l T_{w_0, w_1, 0, 1} \quad \text{mod } q_0
\]

Now, consider an arbitrary row-level $t < p_1$ of the tree, so $q_t = 2^t$, and $s_t^1 = \frac{n_1}{q_t}$. The calculation for node $i$ is:

\[
T_{w_0, w_1, t, i} = T_{w_0, w_1, t, 1, i} + s_t^0 T_{w_0, w_1, t, 0, 1} \quad \text{mod } q_t
\]

Next, we give the calculations for an arbitrary column-level $v > p_1$. For level $v$, $q_v = 2^v$ and $s_v^0 = \frac{n_0}{q_v - p_1}$. The calculation for node $i$ is:

\[
T_{w_0, w_1, v, i} = T_{w_0, w_1, v, v, 1, i} + s_v^0 T_{w_0, w_1, v, v, 0, 1} \quad \text{mod } q_v
\]

where we could written $[\Omega^0 \otimes 1_{n_1}]_{i \times s}$ instead of $\Omega^0_{\frac{l-1}{n_1} \times s}$ for the complex exponential vector. The final level $p_0 + p_1$ contains the 2D DFT coefficients for the window position $(w_0, w_1)$. After all the trees are calculated, all that remains is sub-setting the final level of each complete binary tree, and the algorithm is complete.
Like 1D, the 2D FSWFT algorithm grows linearly in window and array size. To show this, we first point out that calculating each node requires a complex multiplication followed by a complex addition, which we define as an operation (this is the same definition used in Cooley and Tukey (1965)). From this, each level \( l \) of the binary tree takes \( q_l = 2^l \) operations. So, the computational complexity per-window is:

\[
C_{each} = \sum_{l=1}^{p_0+p_1} q_l = (2 + 4 + \ldots + 2^{p_0+p_1}) = 2(2^{p_0+p_1} - 1) = 2(n_0n_1 - 1) \tag{10}
\]

The exact run-time is a bit more complicated, since data-points with indices either less than \( n_0 \) in the row direction or less than \( n_1 \) in the column direction do not require complete trees. An upper bound comes from noticing that that we need \( N_0n_1 \log_2(n_1) \) pre-operations for the row-levels, and at least \( (N_1 - n_1)n_0n_1 \log_2(n_0n_1) \) pre-operations for column levels. So, the 2D FSWFT is upper bounded by \( N_0n_1 \log_2(n_1) + (N_1 - n_1)n_0n_1 \log_2(n_0n_1) + W_0W_12(n_0n_1 - 1) \) operations. For large arrays, the first two terms are much smaller than the third, so the 2D FSWFT takes \( O(n_0n_1W_0W_1) \) operations. Table 2 summarizes the computational complexity of the four 2D SWFT algorithms given in this section. Out of all the 2D SWFT algorithms, ours reaches the top speed, is numerically stable, and works for non-square windows.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Big O</th>
<th>Stable</th>
<th>Non-Square Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D SWDFT</td>
<td>( O(n_0^2n_1W_0W_1) )</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2D SWFFT</td>
<td>( O(n_1n_0 \log(n_0n_1)W_0W_1) )</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2D RSWFT</td>
<td>( O(n_0n_1W_0W_1) )</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2D VR-SWFT</td>
<td>( O(n_0n_1W_0W_1) )</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2D FSWFT</td>
<td>( O(n_0n_1W_0W_1) )</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2: Computational Complexity 2D SWFT algorithms. The first column gives the speed in Big O notation, the second indicates numerical stability, and the third indicates where the algorithm works for non-square windows.

### 3.4 2D Applications: Bullet, Palmprint, and Array Matching

Like 1D, a motivation for the 2D FSWFT comes from a scientific problem. Here, the problem is identifying the type of gun used at the scene of a crime, or “Bullet Matching”. Bullet matching takes an image of a bullet (or, some characteristic of the bullet) found at the scene of a crime, and compares it with a database of bullet images. The goal is matching a crime-scene bullet with a bullet in the database, allowing investigators to infer the type of gun used. Tai (2017) presents a six-step methodology for Bullet Matching. Steps 1-4 pre-process the image, and extract a “Region of Interest”. Step 5 calculates one or multiple statistics measuring the similarity between two images. Finally, Step 6 computes the uncertainty of the similarity statistics. The 2D FSWFT applies to step 5: calculating similarity statistics between images. Song and Vorburger (2000); Vorburger et al. (2007) give a baseline statistic for bullet matching: the global cross-correlation function (Global CCF). The Global CCF uses Fourier methods for efficiently calculating the Cross-Correlation between two images.
Since then, Chu et al. (2013) introduced another method, called “Congruent Matching Cells” (CMC). CMC partitions an image, then calculates the cross-correlation in each partitioned window. Of course, the 2D FSWFT algorithm allows for a higher resolution comparison: instead of partitioning the image, the CCF would be calculated in sliding windows. With more windows, we hypothesize that the 2D FSWFT will increase the matching accuracy without sacrificing computational efficiency. Finally, while Tai (2017) used 2D images, a newly released database (Song (2013)) provides 3D ballistic images. So, 3D FSWFTs will be needed, making the extension of the FSWFT to $kD$ very relevant.

A related problem to bullet matching is “Palmprint Matching”. While the applications are different, the underlying methodology and utility of the FSWFT remain. Meraoumia et al. (2010) uses palmprints as a biometric marker for identification. Like bullet matching, palmprint matching is a multi-step process: a sensor takes a palmprint image, the image is processed, and a Region of Interest (ROI) is extracted. Finally, the processed ROI is compared with a database of palmprint images using similarity statistics. Meraoumia et al. (2010) compares images using “feature extraction”, which represents images with summary statistics, in this case the phase of the 2D DFT and the 2D Block discrete Cosine transform (DCT). Meraoumia et al. (2010) also uses a Phase Correlation Function (PCF), which is remarkably similar to the Global CCF statistic from Bullet Matching. In addition, the PCF function is used for fingerprint matching (Koichi et al. (2004)), suggesting that the 2D FSWFT is applicable to a broad class of “Image Matching”, or more generally “Array Matching” applications. Of course, the utility of the $kD$ SWFT extends to other application classes, for instance “Image Segmentation” Kurnaz et al. (2007).

4 Future Plans

So far, this thesis has extended the FSWFT algorithm to 2D. We propose further extending the FSWFT to $kD$, with details of this extension provided in Section 4.1. In addition, we plan on implementing an efficient version of the algorithm in an easy-to-use software package, with these plans detailed in Section 4.1.2.

In addition to the algorithm, we propose investigating the $kD$ SWFT as a data-analysis tool. Section 4.2 provides details on two components of the SWFT for data-analysis. First, we propose building on Okamura (2011) by deriving the corresponding $kD$ statistical properties. Second, we propose using the SWFT in applications listed in Section 3.4.

4.1 The $kD$ Sliding Window Fourier Transform

A primary goal for this thesis is extending the FSWFT algorithm to $kD$. We propose two specific steps towards reaching this goal. First, we will mathematically derive the algorithm. Second, we plan on developing an efficient, easy-to-use software package for users interested in the $kD$ FSWFT for their own applications. This section discusses these two steps.

4.1.1 $kD$ Algorithm

Section 3.3 derives the 2D FSWFT, and the road-map for the $kD$ will be similar. In particular, we need to understand how both the complex exponential vector and directional shifts work in $kD$. For the complex exponential vectors, we are fairly certain that as $k$ increases, vectors
are built using outer products of the individual $1, 2, \ldots, k-1$ dimensional vectors with vectors of ones, just like 2D. For the directional shifts, we believe the direction changes when we reach levels of the tree corresponding to completion of a 1D FFT over a particular dimension. For instance, say we have an array with dimensions $n_0 \times n_1 \times \ldots \times n_k$, and let $p_0 = \log_2(n_0), p_1 = \log_2(n_1), \ldots, p_k = \log_2(k)$. Then, we hypothesize that levels $1 - p_0$ of the binary trees correspond to shifts across dimension $n_k$, levels $(p_0 + 1) : p_1$ correspond to shifts across dimension $n_{k-1}$, etc.

In addition to extending the $k$D FSWFT, another future direction is extending the algorithm beyond Radix-2. Since the derivation in this thesis used Binary trees, one could image using non-binary trees in future extensions. Finally, we propose investigating whether the computational savings from the FSWFT carry over to other orthogonal bases, such as the discrete Sine, Cosine, and Hartley transforms.

### 4.1.2 Software Implementation

We plan one developing an easy-to-use software package with an efficient implementation of the $k$D FSWFT. We believe that the likelihood $k$D FSWFT adoption increases substantially if a software implementation is available. In addition, available software makes it possible for users to apply the FSWFT algorithm in ways we have not thought of.

So far, all derivations in this thesis are implemented as the R package, called dft. In particular, a C program with an R-interface implements the FSWFT in 1 and 2D, following Algorithms 1 and 2 in the Appendices. R users can currently implement the 1D SWFT using packages e1071 and seewave Meyer et al. (2015); Sueur et al. (2008). However, these packages are application specific, not designed around the DFT.

Of course, many efficient software packages exist implementing the DFT. The fastest and most widely used implementation is called the “Fastest Fourier Transform in the West” (FFTW) Frigo and Johnson (2005). FFTW uses highly sophisticated programs for implementing DFTs of any dimension and input size in $O(n \log(n))$ time. A major FFTW feature is adapting the algorithm to the specific computer architecture. For this, the FFTW consists of three major parts: The executor, planner, and codelet generator. The codelet generator uses a custom compiler, producing highly optimized C code for small base-case FFTs, called “codelets”. The planner takes a “problem” as input, specifying both the dimension and size of the DFT to be calculated. With this problem, the planner tests various factorizations of DFT using the codelets, saving the most efficient plan. Finally, this plan is passed to the executor, which implements the DFT. Benchmarked against approximately 50 DFT implementations, the FFTW comes out as not only the most efficient, but also the most flexible.

We plan on studying the FFTW for improving the efficiency of our own software. For instance, one example of this could be writing several different FSWFT algorithms, and using the appropriate algorithm based on input size and dimension.

Another idea is adapting the implementation discussed at the end of Section 3.3. Here, moving from 1D to 2D increased the number of loops from 3 to 6. One could easily write a separate algorithm for each dimension $k$, each using $3k$ loops. However, representing a $k$D array with a single index is possible using an algebraic expression for array indexing. These details make it possible to implement the FSWFT algorithm in any dimension, with just three loops.
4.2 Data Analysis of the $kD$ SWFT

Section 4.1 focused computational aspects of this thesis, and this section focuses on using the SWFT as a data-analysis tool. This section discusses two components of the SWFT for data-analysis: statistical properties and utility in scientific applications.

4.2.1 Statistics

Section 2.3 summarizes the work of Okamura (2011), who derived statistical properties of the 1D SWFT. Okamura shows that the squared modulus statistic is ideal of analyzing SWFT output (see Figure 1). Okamura also derived analytical expressions for the cross-covariance function between two frequencies in a SWFT output. In addition, Okamura derived the bivariate distribution of the squared modulus statistic for the same frequencies at different time lags. Finally, Okamura used these tools to develop methods for local signal detection.

This thesis proposes extending Okamura's work by deriving $kD$ statistical properties. Specifically, we plan on deriving cross-covariance function and bivariate distributions in $kD$. Time permitting, we will explore other formal statistical frameworks, such as hypothesis tests for periodicity tests (e.g. Fisher (1929)), and model selection for the time-frequency trade-off.

4.2.2 Applications

Ultimately, the value of the $kD$ SWFT depends on its utility in a scientific or engineering context. Section 3.4 discussed several applications which use, or could easily use, a SWFT in one or higher dimensions. We propose investigating these applications, using the $kD$ FSWFT algorithm and statistical properties.

Another goal of using the SWFT in applications is understanding how it works in general classes of applications. For example, the same general structure emerged in multiple “Array Matching” problems: pre-process the image, calculate similarity statistics against a database, determine if there is a match. So, we could move from specific applications (e.g. bullet, palmprint, and fingerprint matching) to the more general problem of “Array Matching”. Another goal could be developing a “Common Task” framework (as discussed in Donoho (2015)) for “Array Matching”, or other application classes. With this, we could analyze the pro’s and con’s of the FSWFT approach systematically, across more that one application.
References


A  Fast Fourier Transform Derivation

Since the FFT calculates the DFT, we start with the definition of the DFT:

\[ a_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \]

\[ k = 0, 1, \ldots, n - 1 \]

Assuming \( n \) is a composite number, we can re-write the indices as:

\[ n = r \times c \]
\[ k = ar + b \]
\[ j = sc + t \]

(11)

With these indices, the DFT is now:

\[ a_{a,b} = \frac{1}{\sqrt{n}} \sum_{s=0}^{c-1} \sum_{t=0}^{r-1} x_{s,t} \omega_n^{-(sc+t)(ar+b)} \]

(12)

Using properties of \( \omega_n \), namely that \( \omega_n^n = 1 \), \( \omega_n^c = \omega_r \), and \( \omega_n^r = c \), we can re-write Equation 12 as:

\[ a_{a,b} = \frac{1}{\sqrt{n}} \sum_{t=0}^{c-1} \omega_c^{-at} \omega_n^{-bt} \left( \sum_{s=1}^{r-1} x_{s,t} \omega_r^{-bs} \right) \]

(13)

The expression inside the parenthesis is just a length-\( r \) DFT. Re-writing this as \( z_{b,t} \), we have:

\[ a_{a,b} = \frac{1}{\sqrt{n}} (\sum_{t=0}^{c-1} z_{b,t} \omega_c^{-at}) \omega_n^{-bt} \]

(14)

And the expression inside the parenthesis is a length-\( c \) DFT. So, this derivation shows how to re-write a length \( n \) DFT into \( c \) length \( r \) DFTs, followed by \( r \) length \( c \) DFTs. The term outside the parenthesis is known as the "Twiddle Factor", a very important quantity.

Since a straightforward DFT using Equation 11 takes \( n^2 \) operations, this derivation takes \( cr^2 + rc^2 = n(r + c) \) operations. So, we see that the speed of the FFT comes from combining two smaller DFTs. These smaller DFTs can be further divided, resulting in large speed-ups for highly composite values of \( n \).

B  1D Fast Sliding Window Fourier Transform

This section mathematically derives the 1D FSWFT algorithm, then details its implementation.
B.1 1D FSWFT Algorithm

Let \( x = (x_0, x_1, \ldots, x_{N-1}) \) be a length \( N \) vector, and let \( n = 2^p \leq N \) be the window size. Then, there are \( W = N - n + 1 \) complete binary trees, indexed by \( w \). Each tree has \( p = \log_2(n) \) levels, and \( 2^p \) nodes at each level. The calculation at each node is a value of a shifted tree (S), plus a value of the current tree (C) multiplied by a twiddle factor (T): \( S + TC \).

To derive the FSWFT, consider calculating the DFT for the \( w \)th window. Level zero of the tree is just \( w \)th data point, \( x_w \). Level one is built using level zero of the current tree (\( x_w \)), level zero of the tree shifted by \( \frac{n}{2} \) (\( x_w - \frac{n}{2} \)), and a two-vector of twiddle factors (\( \omega_0^n, \omega_1^n \)). Notice that the exponents in the twiddle factor correspond with the tree-shift. With these three components, the level one calculations are:

\[
x_w - \frac{n}{2} + \omega_0^n x_w \\
\]

(15)

Level two is built using level one of the current tree (the above calculations), level one of the tree shifted by \( \frac{n}{4} \) (the above calculations for the \( w - \frac{n}{4} \) tree), and a 4-vector of twiddle factors (\( \omega_0^n, \omega_1^n, \omega_2^n, \omega_3^n \)). The level two calculations are:

\[
x_w - \frac{3n}{4} + \omega_0^n x_w - \frac{3n}{4} + \omega_0^n (x_w - \frac{n}{2} + \omega_0^n x_w) \\
x_w - \frac{3n}{4} + \omega_1^n x_w - \frac{3n}{4} + \omega_1^n (x_w - \frac{n}{2} + \omega_1^n x_w) \\
x_w - \frac{3n}{4} + \omega_2^n x_w - \frac{3n}{4} + \omega_2^n (x_w - \frac{n}{2} + \omega_2^n x_w) \\
x_w - \frac{3n}{4} + \omega_3^n x_w - \frac{3n}{4} + \omega_3^n (x_w - \frac{n}{2} + \omega_3^n x_w)
\]

(16)

The remaining levels are calculated the same way. Level \( p \) is the base of the tree, an \( n \)-vector build from level \( p - 1 \) of the current tree, level \( p - 1 \) of the tree shifted by \( \frac{n}{2^p} = 1 \), and an \( n \)-vector of twiddle factors (\( \omega_0^n, \omega_1^n, \omega_2^n, \ldots, \omega_1^{(n-1)n} \)). The \( 2^p \) nodes at level \( p \) of the tree are DFT in \( w \)th window.

B.2 1D FSWFT Implementation

Implementing the FSWFT can be done in several ways, but the core of the algorithm is three nested loops:

- A loop over trees
- A loop over levels of the tree
- A loop over nodes in a level of a tree
Inside the three loops, the calculation is simply:

\[ X_a = X_b + T_c \times X_d \] (17)

Where \( a, b, c, \) and \( d \) are indices to the arrays where the intermediate data and twiddle factors are stored. This matches our derivation, as \( X_b \) is the value of the shifted tree, \( T_c \) is the value of the twiddle factor vector, and \( X_d \) is the value of a current tree. Complete pseudo-code for the FSWFT is given in Algorithm 1.

---

**Algorithm 1: 1D Fast Sliding Window Fourier Transform**

<table>
<thead>
<tr>
<th>input</th>
<th>( X ): Length N-complex-vector, ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( W = N - n + 1 )</td>
</tr>
<tr>
<td></td>
<td>( p = \log_2(n) )</td>
</tr>
<tr>
<td></td>
<td>( T = T_{N, \text{level}, \text{node}} ) Tree data-structure</td>
</tr>
<tr>
<td>for ( w ) in ( 0 : (N - 1) ) do</td>
<td></td>
</tr>
<tr>
<td>for ( l ) in ( 1 : p ) do</td>
<td></td>
</tr>
<tr>
<td>( q_l = 2^l )</td>
<td></td>
</tr>
<tr>
<td>( s_l = \frac{n}{q_l} )</td>
<td></td>
</tr>
<tr>
<td>Skip if ( w &lt; n - s_l )</td>
<td></td>
</tr>
<tr>
<td>for ( i ) in ( 0 : (q_l - 1) ) do</td>
<td></td>
</tr>
<tr>
<td>( T_{w,l,i} = T_{w-s_l,l-1,i \text{mod} q_l-1} + \Omega_{i \times s_l} T_{w,l-1,i \text{mod} q_l-1} )</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>( A = T_{W:(N-1),p} )</td>
<td></td>
</tr>
</tbody>
</table>

**output:** \( A: n \times W \) array

---

**C 2D Recursive Sliding Window Fourier Transform**

Section 2.1.4 gives a recursive algorithm for calculating the 1D SWFT, the Recursive Sliding Window Fourier Transform (RSWFT). The RSWFT uses both the Fourier Shift Theorem and the fact that data-points change in successive windows. Recently, Park (2015) extended the RSWFT to the 2D. Park’s 2D RSWFT works like 1D, except that instead of two data-points changing in successive windows, either two rows (or columns) change. The 2D RSWFT has the same downside as in 1D: numerical error. Figure 9 shows that numerical error increases as the window position slides in both directions.

Following Park (2015), we give the 2D RSWFT. For clarity, we remove the scaling factor \( \frac{1}{\sqrt{n_0 n_1}} \). Starting with the 2D SWFT definition:

\[
 a_{k_0, k_1}^{w_0, w_1} = \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} x_{w_0+j_0, w_1+j_1} \omega_{-j_0 k_0} \omega_{-j_1 k_1} (18)
\]

Consider moving the window one column to the right (the same argument applies for moving the window one row below). This means we want \( a_{k_0, k_1}^{w_0, w_1} \) as a function of 2D DFT of the previous window, \( a_{k_0, k_1}^{w_0, w_1-1} \). To do this, we re-write Equation 6:
Now, using the fact that $\forall n, \omega_n^m = 1$, we factor $\omega_{n_1}^{k_1}$ out of each term:

$$
\begin{align*}
    a_{k_0, k_1}^{w_0, w_1} &= \omega_{n_1}^{k_1} \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} x_{\hat{w}_0 + j_0, \hat{w}_1 + j_1} - j_0 k_0 \omega_{n_1}^{-j_0 - j_1} k_1 \\
    &+ \omega_{n_1}^{k_1} \sum_{j_0=0}^{n_0-1} x_{\hat{w}_0 + j_0, \hat{w}_1 + n_1} - j_0 k_0 \omega_{n_1}^{-j_0} k_1 \\
    &- \omega_{n_1}^{k_1} \sum_{j_0=0}^{n_0-1} x_{\hat{w}_0 + j_0, \hat{w}_1} - j_0 k_0 \omega_{n_1}^{-j_0} k_1
\end{align*}
$$

The first term of the above equation is the 2D DFT of window shifted one column to the left, $a_{k_0, k_1}^{w_0, w_1-1}$. The second and third terms are 1D DFTs corresponding to the columns
$w_1$ and $w_1 - n_1$, which are exactly the columns being added and removed from the sliding window! Re-writing the 1D column DFTs as:

$$z_{k_0}^{w_0,w_1} = \sum_{j_0=0}^{n_0-1} x_{w_0+j_0,w_1} \omega_{n_0}^{-j_0k_0}$$  \hspace{1cm} (21)

The 2D RSWFT formula is:

$$a_{k_0,k_1}^{w_0,w_1} = \omega_{n_1}^{k_1} (a_{k_0,k_1}^{w_0,w_1-1} + z_{k_0}^{w_0,w_1} - z_{k_0}^{w_0,w_1-n_1})$$  \hspace{1cm} (22)

Park (2015) points out that the using the linearity property of the DFT, only one column DFT is needed per window: the DFT of the new column $w_1$ minus the column moving out of the window $w_1 - n_1$. In addition, Park (2015) points out that the 1D DFTs can be calculated recursively, using the 1D RSWFT. So, assuming that both 2D DFT of the previous window and the 1D DFTs of the new columns are calculated, each new coefficient $a_{k_0,k_1}^{w_0,w_1}$ can be calculated directly from Equation 22. Since Equation 22 is two complex additions followed by a complex multiplication, the 2D RSWFT takes $O(n_0n_1)$ operations per window. Of course, since there are $W_0W_1$ overall window locations, the full 2D RSWFT takes $O(n_0n_1W_0W_1)$ operations, reaching the computational lower bound. Of course, the numerical instability shown in Figure 9 still exist. Next, a numerically stable $O(n_0n_1W_0W_1)$ is given.

D 2D Fast Sliding Window Fourier Transform Implementation

The programming details of the 2D FSWFT algorithm more complicated, but follow the same structure as the 1D implementation. In 1D, the algorithm consists of three loops: over the trees, over the levels, and over the nodes. In 2D, the algorithm requires nested loops over trees, levels and nodes, but now each of these components requires two-loops: once again loops over trees, nodes, and levels, but in this case each of these requires two loops:

- A loop over rows
- A loop over columns
- A loop over row-levels
- A loop over nodes corresponding to row-levels
- A loop over column-levels
- A loop over nodes corresponding to column-levels

Algorithm 2 provides detailed pseudo-code of the 2D FSWFT.
Algorithm 2: 2D Fast Sliding Window Fourier Transform

**input:** $X: N_0 \times N_1$ array, $n_0, n_1$

$W_0 = N_0 - n_0 + 1$

$W_1 = N_1 - n_1 + 1$

$p_0 = \log_2(n_0)$

$p_1 = \log_2(n_1)$

$T = T_{N_0, N_1, \text{level}, \text{node}}$ Tree data-structure for each window

$\Omega^1 = [\omega^0, \omega^{n_1-1}, \ldots, \omega_{n_1-1}]$

$\Omega^0 = [\omega^0, \omega^{n_0-1}, \ldots, \omega_{n_0-1}]$

$T_{w_0, \cdot, \cdot} = X$

for $w_0$ in $0 : (N_0 - 1)$ do

for $w_1$ in $0 : (N_1 - 1)$ do

for $l$ in $1 : p_1$ do

$q_l = 2^l$

$s_l = \frac{n_l}{q_l}$

Skip if $w_1 < n_1 - s_l$

for $i$ in $0 : (q_l - 1)$ do

$T_{w_0, w_1, t, i} = T_{w_0, w_1, t-1, i \mod q_{l-1}} + \Omega^1_{i \times s_l} T_{w_0, w_1, t-1, i \mod t-1}$

end

end

for $l$ in $(p_1 + 1) : (p_1 + p_0)$ do

$q_l = 2^l$

$s_l^0 = \frac{n_l}{q_l - p_1}$

Skip if $w_0 < n_0 - s$

for $i$ in $0 : (q_l - 1)$ do

$T_{w_0, w_1, v, i} = T_{w_0, w_1, v-1, i \mod q_{v-1}} + \Omega^0_{1 \times s_l^0} T_{w_0, w_1, v-1, i \mod q_{v-1}}$

end

end

$A = T_{W_0 : (N_0 - 1), W_1 : (N_1 - 1), p_0 + p_1}$

**output:** $A$
E Fourier Shift Theorem

The Fourier shift theorem calculates a new DFT based on the Fourier coefficients of a previously calculated DFT, with overlapping data. The shift theorem is the central idea behind the Recursive Sliding Window Fourier Transform (RSWFT), where the coefficients in the next window are simply related to the coefficients in the previous window. Starting with the DFT:

\[ a_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \]  

(23)

If we shift \( x \) by some integer \( \delta \), then the corresponding coefficients are:

\[ a_{k+\delta} = \sum_{j=0}^{n-1} x_{j+\delta} \omega_n^{-jk} \]  

(24)

Next, let \( m = j + \delta \), then:

\[
a_{k+\delta} = \sum_{m=\delta}^{n-1+\delta} x_m \omega_n^{-(m-\delta)k} \\
= \sum_{m=\delta}^{n-1+\delta} x_m \omega_n^{-mk} \omega_n^{\delta k} \\
= \omega_n^{\delta k} \sum_{m=\delta}^{n-1+\delta} x_m \omega_n^{-mk} \\
= \omega_n^{\delta k} \sum_{m=0}^{n-1} x_m \omega_n^{-mk} \\
= \omega_n^{\delta k} a_k
\]  

(25)