Analysis of High-Frequency Data with Wavelet Transforms

WILLIE BEDOYA

I. Problem Formulation and Background

The objective of this project is to assess the quality of denoised price signals for financial time series. This will be done by first collecting high-frequency price data and then adding noise to the collected data to simulate market noise. Then, wavelet-based denoising techniques will be applied to the noise-added price data. This quality of the denoised signals will finally be compared to the underlying true signals and the original price data. The five stocks tested on were TSLA, WMT, JNJ, INTC, and AAPL. The sample prices begin on January 5, 2016 at 11:30am and end on April 11, 2024 at 3:30pm. This sums up to over 14,000 data points per stock.

Market price data is notoriously noisy. It contains various sources of noise, including high-frequency fluctuations, which makes it difficult to discover any sort of meaningful patterns or trends. The application of wavelet transforms to financial time series data aims to decompose signals into different time-frequency domains. In this project, the effectiveness of different wavelet families and thresholding techniques will be analyzed and evaluated.

To understand wavelet transforms, first a wavelet must be defined. A wavelet is simply a waveform that has limited duration, an average value of zero, and a nonzero norm ("Wavelet Signal Analyzer"). Figure 1 below provides a nice illustration of a sine wave versus a wavelet. Sinusoids do not have limited duration whereas wavelets do, and from the image it is clear that wavelets can be irregular and asymmetric compared to sine waves. Additionally, one considerable drawback of Fourier analysis is that it involves a tradeoff between time and frequency resolution. This is due to Heisenberg's uncertainty principle, which in physics means that the position and momentum of a particle cannot be measured simultaneously and in harmonic analysis means that "a nonzero function and its Fourier transform cannot both be sharply localized" ("The Uncertainty Principle - UGA Math"). Specifically,

$$\Delta t \Delta f \geq \frac{1}{4\pi}, \tag{1}$$

where t is time and f is ordinary frequency (Hall). When Δt is large, time resolution is poor but frequency resolution is good. Conversely, if Δt is small, time resolution is fine but frequency resolution is poor. Such is the reason why wavelet transforms are considered over Fourier transforms.

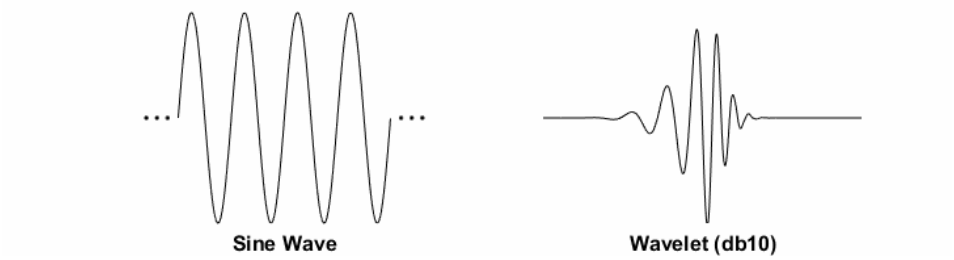

Figure 1. A Sine Wave Compared to a Wavelet ("Wavelet Signal Analyzer")

A wavelet transform allows one to decompose signals "into shifted and scaled versions of a wavelet" ("What are Wavelet Transforms?"). Similar to how Fourier analysis consists of decomposing a signal into sine waves of different frequencies, wavelet decomposition involves decomposing a signal into shifted and scaled versions of the original wavelet. The continuous wavelet transform in terms of its original signal can be defined as ("Continuous Wavelet Transform"):

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{a}}\Psi\left(\frac{t-b}{a}\right),\ a, b, t \in (-\infty, \infty). \tag{2}$$

In this formula, a is the scaling factor which adjusts the width of the wavelet and b is the time shift factor which moves the wavelet along the time axis. $\Psi\left(\frac{t-b}{a}\right)$ is the dilated/translated mother wavelet

(also called the kernel of the wavelet transform) which has to satisfy the zero mean condition stated earlier and also the admissibility condition. The zero mean condition can be expressed as

$$\int_{-\infty}^{\infty} \Psi(t)dt = 0 \tag{3}$$

and the admissibility condition, which ensures that the mother wavelet can be used to completely reconstruct the original signal, is expressed as

$$C_{\Psi} = \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega < \infty. \tag{4}$$

The term $\Psi(\omega)$ is the Fourier transform of the wavelet function $\Psi(t)$, as this function is in the time domain. To solve for the wavelet coefficient X(a, b) given a scaling factor a and time shift factor b, the continuous wavelet transform $\Psi_{a,b}(t)$ is multiplied by the original signal or function $x(t)$ being analyzed and integrated with respect to time:

$$X(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t)\Psi(\frac{t-b}{a})dt. \tag{5}$$

II. Approach and Description of Solution Methods

One method for denoising financial time series involves first decomposing the time series data into a sum of intrinsic mode function (IMF) components and a residual component. The improved complete ensemble empirical mode decomposition with adaptive noise (ICEEMDAN) is an approach outlined in a paper titled "De-noising Classification Method for Financial Time Series Based on ICEEMDAN and Wavelet Threshold, and Its Application" (Liu and Cheng). To obtain the IMF components and residual component, Gaussian noise is added to the time series

$$x_i(t) = x(t) + \beta_0 E_1(\omega^i) \tag{6}$$

where $x_i(t)$ is the new time series after i number of experiments, $x(t)$ is the original time series, β_0 is an amplitude coefficient of the added noise, $E_1(\cdot)$ is the first IMF component obtained after empirical mode decomposition (EMD), and ω^i is the i-th added Gaussian white noise. This equation represents the first application of Gaussian noise. For the purposes of the report, $i = 1$ (i.e., Gaussian noise is only being added once). Additionally, the amplitude coefficient β_k of the added noise will be the previous stock price times a factor of $\frac{1}{10000}$. This allows for dynamic adjustments of the noise given that the stock prices experience considerable change over the sample period.

After adding noise to get $x_i(t)$, the first stage residual $r_1(t)$ can be obtained by calculating the i-th local average of $x_i(t)$:

$$r_1(t) = < M(x_i(t)) > \tag{7}$$

where $< M(\cdot) >$ is the operator used to calculate the average. One commonly used operator is one involving cubic spline interpolation ("Empirical Mode Decomposition: Theory & Applications"). Upper and lower envelopes, $e_{u(t)}$ and $e_{l(t)}$ respectively, of a time series $x(t)$ are formed by applying cubic spline interpolation to the local maxima and minima of $x(t)$. Hence, the operator used to calculate the average can be expressed as

$$\frac{e_{u(t)} + e_{l(t)}}{2}. \tag{8}$$

For full details on the algorithm for empirical mode decomposition, as well as cubic spline interpolation, see the Theorems section of the Appendix.

The first IMF of ICEEMDAN via EMD is

$$IMF_1(t) = x(t) - r_1(t). \tag{9}$$

Using the first stage residual, local average of the new signal $r_1(t) + \beta_1 E_2(\omega^i)$ can be calculated to obtain the second stage residual:

$$r_2(t) = \ < M(r_1(t) + \beta_1 E_2(\omega^i)) >. \quad (10)$$

The second IMF of the original time series $x(t)$ can be obtained by taking the difference of the two residuals:

$$IMF_2(t) = r_1(t) - r_2(t). \quad (11)$$

This procedure is repeated and can be expressed recursively as

$$r_k(t) = \ < M(r_{k-1}(t) + \beta_{k-1} E_k(\omega^i)) > \quad (12)$$

and

$$IMF_k(t) = r_{k-1}(t) - r_k(t). \quad (13)$$

These steps should be repeated until the extreme points of the residual do not surpass a certain threshold; the authors of the paper designate this threshold to be 2. The final residual is therefore

$$R(t) = x(t) - \sum_{k=1}^{K} IMF_k \quad (14)$$

and can be rearranged to find the original time series in terms of IMFs and a residual:

$$x(t) = \sum_{k=1}^{K} IMF_k + R(t). \quad (15)$$

For every IMF_k component, as well as the summation $\sum_{k=1}^{i} IMF_k$, t tests and unit root tests can be carried out. The purpose of this is to conduct population mean tests and Augmented Dickey-Fuller (ADF) tests for stationarity. The IMF and residual components are then separated into two separate components/categories: a noise-containing component

$$x(t)_{noise} = \sum_{k=1}^{i} IMF_k \quad (16)$$

and a noise-removed component

$$x(t)_{non-noise} = \sum_{k=i+1}^{K} IMF_k + R(t). \quad (17)$$

When in the range of $k = 1$ to $k = i$, the overall mean of IMF_k and its summation should equal 0. Additionally, in the same range of k values IMF_k and its summation should be stationary. These are determined by the two tests previously mentioned (mean and ADF).

Once the noise-containing component $x(t)_{noise}$ and noise-removed component $x(t)_{non-noise}$ have each been determined, wavelet threshold denoising is applied to the noise-containing component (thresholds are discussed in detail in section II of this report). This breaks the noise-containing component into a noise component $\varepsilon(t)$ and a noise-free component $x(t)_{noise-free}$. This noise-free component $x(t)_{noise-free}$ is integrated to the noise-removed component $x(t)_{non-noise}$ to arrive at the final configuration for the denoised component $x(t)_{de-noised}$:

$$x(t) = \varepsilon(t) + x(t)_{de-noised}. \quad (18)$$

To summarize, this denoising technique called ICEEMDAN involves first decomposing the time series $x(t)$ into a summation of IMF components and one residual component. Then, depending on which IMFs pass the population mean test and ADF test, $x(t)$ is separated into a noise-containing component $x(t)_{noise}$ and a noise-removed component $x(t)_{non-noise}$. The noise-containing component $x(t)_{noise}$ is then broken down into a final noise component $\varepsilon(t)$ and a noise-free component $x(t)_{noise-free}$. Lastly, the noise-removed component $x(t)_{non-noise}$ and the noise-free component $x(t)_{noise-free}$ are summed together to form $x(t)_{de-noised}$ (along with the noise $\varepsilon(t)$) to finally arrive at $x(t) = \varepsilon(t) + x(t)_{denoised}$.

A second method typically used for denoising signals is rooted in discrete wavelet transforms. Broadly speaking, there are three steps involved in wavelet-based denoising of signals. These are (1) wavelet decomposition, (2) thresholding, and (3) reconstruction. The Python library ‘pywt’ that shall be

used in approaching the wavelet-based denoising problem in section III uses discrete wavelet transforms as opposed to the continuous transforms introduced in section I.

In discrete wavelet transforms, the first step of decomposition yields the wavelet coefficients, on which thresholding can subsequently be applied and used for reconstruction. The decomposition uses orthonormal wavelet basis vectors $\psi_{a,b} = \{\psi_{a,b}(x_i),\ i = 1, \dots, N\}$ which can be grouped into sets distinguishable by a scale index a (Lindsay et al.). This scale index is akin to the scaling factor a mentioned in section I for the continuous wavelet transform case (equations 2 – 5). Additionally, every vector in the ath set can be distinguished by a location index b, which again corresponds to the time shift factor b for the continuous case. Given that wavelet coefficients are capable of being distinguished by scale index a and location index b, these coefficients provide information on the magnitude and location of values in the function or time series being denoised. The wavelet basis vectors are formed by translating two mother wavelet filters: one highpass filter and one lowpass filter. These filters correspond to the coefficients that represent scale and location, a and b, respectively. The highpass filter shall be designated by H and the lowpass designated by G.

The location index b helps indicate where the nonzero portion of each basis vector lies. Additionally, the number of nonzero elements in each basis vector is equal to 2^a. For instance, consider an $a = 1$ wavelet basis vector for a number of $N = 8$ observations using a Haar wavelet filter (Haar is a type of wavelet family discussed in section II.A). The highpass filter H is equal to

$$H = \{\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\}$$

and the lowpass filter G is equal to

$$G = \{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\}.$$

The basis vectors for scale index $a = 1$ can then be written as follows:

$$\psi_{1,1} = \{\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0, 0, 0, 0, 0\}$$
$$\psi_{1,2} = \{0, 0, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0, 0, 0\}$$
$$\psi_{1,3} = \{0, 0, 0, 0, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0\}$$
$$\psi_{1,4} = \{0, 0, 0, 0, 0, 0, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\}.$$

As can be seen, the value of location index b denotes where the nonzero elements lie; when $b = 1$ only elements 1 and 2 are nonzero, when $b = 2$ only elements 3 and 4 are nonzero, and so on. Additionally, the magnitude of each element is equal to the scale index times the elements in the scaling filter.

For a scale index of $a = 2$, the wavelet basis vectors are

$$\psi_{2,1} = \{\frac{-1}{2}, \frac{-1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0\}$$
$$\psi_{2,2} = \{0, 0, 0, 0, \frac{-1}{2}, \frac{-1}{2}, \frac{1}{2}, \frac{1}{2}\}.$$

Since $a = 2$, there are now 2^2 nonzero elements instead of 2^1. Furthermore, the magnitude of the wavelet basis vectors are equal to the elements in scaling filter G to the power of a. Once again, the location index b can be used to determine where the nonzero elements are.

Lastly, when $a = 3$ the basis vector is equal to

$$\psi_{3,1} = \{\frac{-1}{\sqrt{8}}, \frac{-1}{\sqrt{8}}, \frac{-1}{\sqrt{8}}, \frac{-1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}\}.$$

These wavelet basis vectors are necessary to decompose the original time series and find the wavelet coefficients. To solve for the wavelet coefficients $D_{a,b}$ (given the scale index a and location index b), the inner product of each observation in the time series and the wavelet basis vector is taken:

$$D_{a,b} = \sum_{i=1}^{N} x(t_i)\psi_{a,b}(t_i) = \ < x, \psi_{a,b} >. \tag{19}$$

Once the coefficients have been solved for, thresholding is applied. This involves filtering out some of the coefficients calculated and is discussed in detail in sections II.D and II.E. The final step is reconstruction. For the Haar wavelet family, the filters for reconstruction are simple to find by taking the inverse of *H* and *G* as follows:

$$H' = \{\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\}$$

$$G' = \{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\} = G.$$

The example shown above for the Haar wavelet family works fine because it is the simplest wavelet family. However, for more complex wavelets such as the Daubechies or Symlet the low-pass and high-pass filters are not as cut-and-dry. Therefore, a second common approach used to perform discrete wavelet transforms is called the orthogonal-pyramid algorithm.

This is an iterative algorithm where at each step the wavelet and scaling coefficients are computed from the scaling coefficients of the last step. The first scaling coefficients (i.e., when the scale index *a* is equal to 0) are taken to be the elements of the original time series or signal:

$$A_{a,b} = A_{0,b} = x(t_b) \tag{20}$$

$$b = \{1, 2, 3, \dots, N\} \tag{21}$$

where $A_{0,b}$ is the set of scaling coefficients at a location index of *b* and *N* is the number of points in the time series.

The pair of filters *H* and *G* are utilized in this algorithm and are considered to be quadrature mirror filters, meaning that *G* is the reverse of *H* and every second element is multiplied by a factor of -1. Each element of *G* is equal to

$$g_p = (-1)^p h_{m-p+1} \tag{22}$$

$$p = \{1, 2, 3, \dots, m\} \tag{23}$$

where *m* is the total number of elements in each filter, g_p is the *p*-th element of the lowpass filter *G*, and h_{m-p+1} is the *m-p+1*-th element of the highpass filter *H*.

The scaling coefficients beyond the first ones are found by convolving the lowpass scaling filter *G* with the previous smaller-scale coefficients (meaning that the scale index is lower at this level) and subsampling by 2 (i.e., sampling only every second data point):

$$A_{a,b} = \sum_{p=1}^{m} g_p A_{(a-1),(2b+m-1-p)} \tag{24}$$

where b = 1 through $2^{-a}N$. This convolution makes it clear that the current scaling coefficients are found by considering each element in *G*, the scaling coefficients indexed by the previous scale index a - 1 and the location index $2b + m - 1 - p$. The additional $2b$ term is what induces subsampling.

To find the wavelet coefficients $D_{a,b}$ under this algorithm, a similar procedure is done except the scaling coefficients will be convolved with the highpass filter *H* instead of *G*:

$$D_{a,b} = \sum_{p=1}^{m} h_p A_{(a-1),(2b+m-1-p)}. \tag{25}$$

Once again, subsampling by 2 was applied. As before, the wavelet coefficients are passed through whichever thresholding technique is selected. Lastly, to reconstruct the time series and obtain a denoised signal one can work backwards by starting from a higher scale index *a* and working down to smaller scales. This is expressed as

$$A_{a,b} = \sum_{p=1}^{m} (D_{a+1,b} g_p + A_{a+1,b} h_p). \tag{26}$$

By working backwards from large scale index *a* values to smaller ones, one can arrive to $A_{0,b}$. Recalling from earlier, we initially set this value to equal the original time series values. However, now that wavelet coefficients were first found and thresholding applied, this final value now provides the denoised signal.

A. Haar Wavelet

There are various families of wavelets that can be used for implementing a wavelet-based denoising technique. One of the simplest is the Haar wavelet. It is a sequence of rescaled square-shaped functions that form a discrete, non-differentiable wavelet (Govindan et al.). The continuous wavelet function for the Haar wavelet is:

$$\Psi(t) = \begin{cases} 1 & 0 \le t < \frac{1}{2}, \\ -1 & \frac{1}{2} \le t < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{27}$$

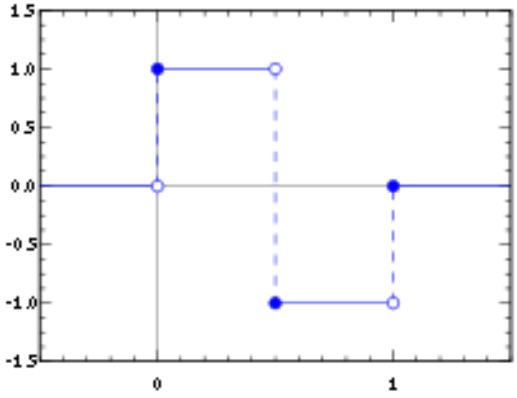

Figure 2. Haar Wavelet Function (Govindan et al.)

Figure 2 above shows what this wavelet looks like, and it's visually clear that this type of wavelet is one of the simplest ones. It is actually the simplest form of the Daubechies wavelet, which shall be explained in the following section.

B. Daubechies Wavelet

The Daubechies wavelets are a family of orthogonal wavelets (meaning that the inner product of any two wavelets or scaled versions of wavelets equals zero, unless they're the same wavelet) ("Daubechies Wavelet"). The scaling function for a Daubechies wavelet, defined by $\phi(t)$, and the wavelet function $\Psi(t)$ have a compact support length of $2n$. The support length of the wavelet function is defined as "the size of the interval on which it is non-zero" ("Haar Wavelet Filterbanks"). This implies that the function is non-zero over a finite interval, allowing for efficient computations as only a finite number of terms are present when applying the wavelets. The inner product between the original time series $x(t)$ and the wavelet function $\Psi(t)$ is equal to the integral

$$\int_{n}^{n+L} x(t)\Psi(t)dt =< x, \Psi > \tag{28}$$

where n to $n + L$ represents the domain of compact support.

Additionally, the wavelet function $\Psi(t)$ has n vanishing moments, which means that when the inner product of $\Psi(t)$ is taken with a polynomial of degree less than n, the result is 0 (i.e., it is orthogonal to all polynomials of degree less than n) (Haar Wavelet Filterbanks). This can be expressed as an integral,

$$\int_{-\infty}^{\infty} t^q \psi(t)dt = 0, \tag{29}$$

where $q = 0, 1, \ldots, n - 1$ and n is the number of vanishing moments.

A final property of Daubechies wavelets is that they are parameterized by an order, denoted by *n*. As has been stated, this order determines the number of vanishing moments *n*, as well as the smoothness of the wavelet. The higher the order, the smoother the wavelet and the more vanishing moments.

The notation commonly used is *dbN*, where *N* represents the order of the wavelet. For instance, *db*1 would represent the Haar wavelet previously mentioned, as this is actually the simplest and first member of the Daubechies wavelet family. Similarly, *db*2 would be a Daubechies wavelet of order 2 and contain 2 vanishing moments.

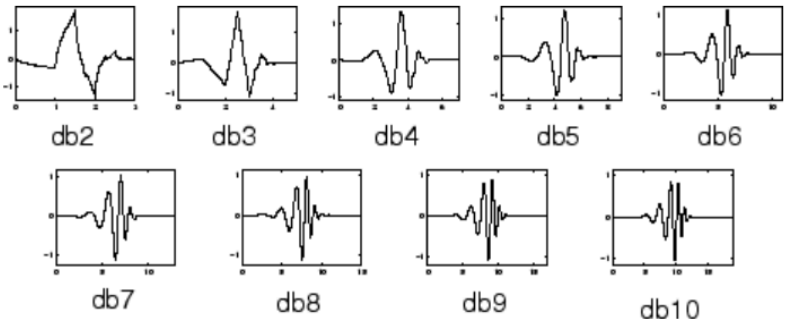

Figure 3. Daubechies Wavelets of Orders 2 through 10 ("Introduction to Wavelet Families")

Figure 3 above showcases different Daubechies wavelets at different orders. The first one, *db*2, appears quite similar to the Haar wavelet shown previously in Figure 2. This makes sense since the Haar wavelet is the same as *db*1, so they should be visually related. As the order *N* increases, there are more vanishing moments and the number of non-zero terms decreases.

C. Symlet Wavelet

Similar to Daubechies wavelets, symlet wavelets define a family of orthogonal wavelets. ("Symlet Wavelet"). As the symlet is a member of the Daubechies family of wavelets, it too has a scaling function $\phi(t)$ and wavelet function $\Psi(t)$ with a compact support length of $2n$. Once again, the scaling function has *n* vanishing moments. The main difference between Symlet wavelets and a typical Daubechies wavelet is that the wavelet coefficients are chosen specifically to optimize symmetry rather than only considering vanishing moments.

Upon visual inspection, the differences between Figure 3 and Figure 4 makes clear the objective of symmetry for symlet wavelets. Comparing *db*6 with *sym*6, for instance, indicates that the symlet wavelet is indeed being optimized for symmetry more so than *db*6 is.

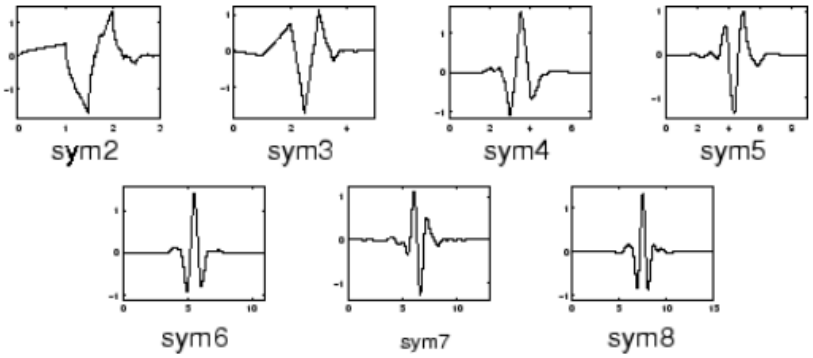

Figure 4. Symlet Wavelets of Orders 2 through 8 ("Introduction to Wavelet Families")

D. Hard Thresholding

Thresholding is a method used to de-noise wavelet coefficients after the noise-containing component x(t) has already undergone wavelet decomposition. There are two kinds of thresholding: hard thresholding and soft thresholding. Hard thresholding is a simple and direct approach, where wavelet coefficients that are smaller in magnitude than some specified threshold are set equal to zero and coefficients exceeding the threshold are kept unchanged. The following formula represents the hard threshold function, where $\omega_{j,k}$ is the wavelet coefficient, λ is the threshold, and $\widehat{\omega}_{j,k}$ is the quantized wavelet coefficient (Liu and Cheng):

$$\hat{\omega}_{j,k} = \begin{cases} \omega_{j,k}, & |\omega_{j,k}| \geq \lambda \\ 0, & |\omega_{j,k}| < \lambda \end{cases} \tag{30}$$

E. Soft Thresholding

Soft thresholding is more subtle than hard thresholding. Similar to hard thresholding, any wavelet coefficients that are smaller than a threshold are set to zero. However, the difference is that coefficients larger than the threshold are not only kept unchanged but are also shrunk towards zero by the threshold value. This allows for a smoother output compared to the hard thresholding approach. The following formula represents the soft threshold function, where $\omega_{j,k}$ is the wavelet coefficient, λ is the threshold, *sgn* is the sign function, and $\widehat{\omega}_{j,k}$ is the quantized wavelet coefficient (Liu and Cheng):

$$\hat{\omega}_{j,k} = \begin{cases} \mathrm{sgn}(\omega_{j,k})\left(|\omega_{j,k}| - \lambda\right), & |\omega_{j,k}| \geq \lambda \\ 0, & |\omega_{j,k}| < \lambda \end{cases} . \tag{31}$$

III. Results

A. Applying Noise to the Data

The approach I take in my results is to apply discrete wavelet transforms to denoise the data. This involves usage of the 'pywt' library in Python. Before denoising the data to see if it assisted in identifying any underlying patterns, Gaussian noise was applied to the data. To ensure that the stock prices were scaled appropriately, the data was multiplied by a factor of $(1 + price_{last} \cdot \frac{noise}{10000})$, where *noise* represents a Gaussian random variable with mean of 0 and standard deviation of 1. The noise was multiplied by such a small factor because the prices being used are updated every hour, and typically the logarithmic returns for one hour are very small. Hence, a small factor of $\frac{noise}{10000}$ was used. For the five stocks selected (TSLA, WMT, JNJ, INTC, AAPL), adding noise resulted in the following charts:

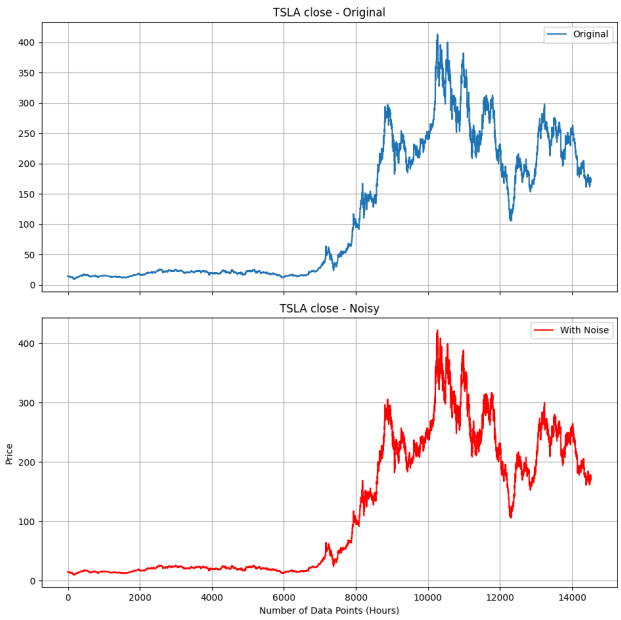

Figure 5. TSLA Original and with Noise

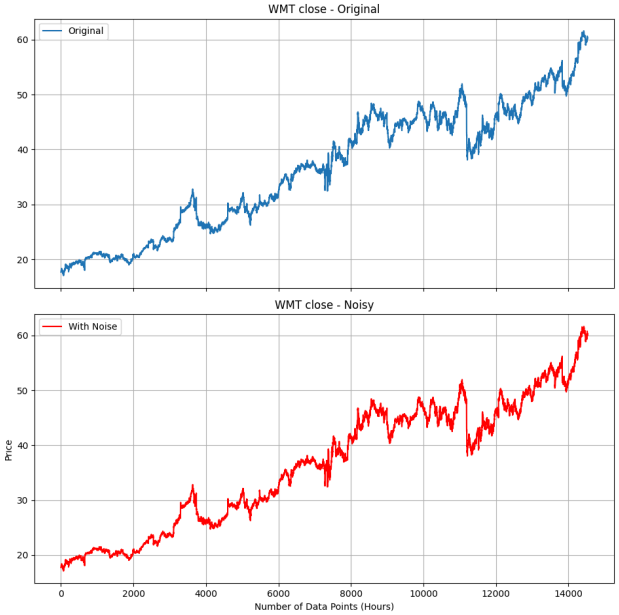

Figure 6. WMT Original and with Noise

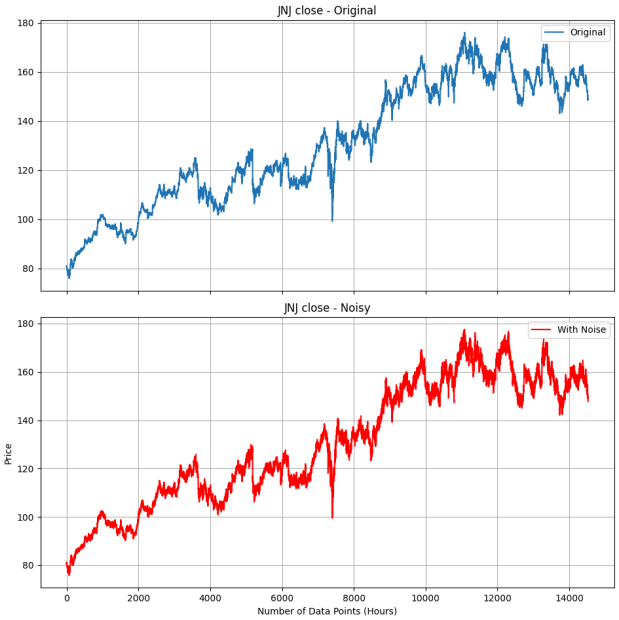

Figure 7. JNJ Original and with Noise

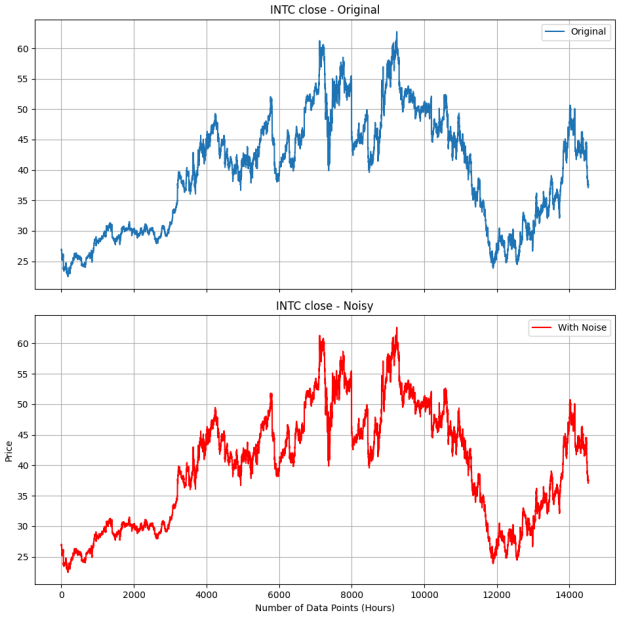

Figure 8. INTC Original and with Noise

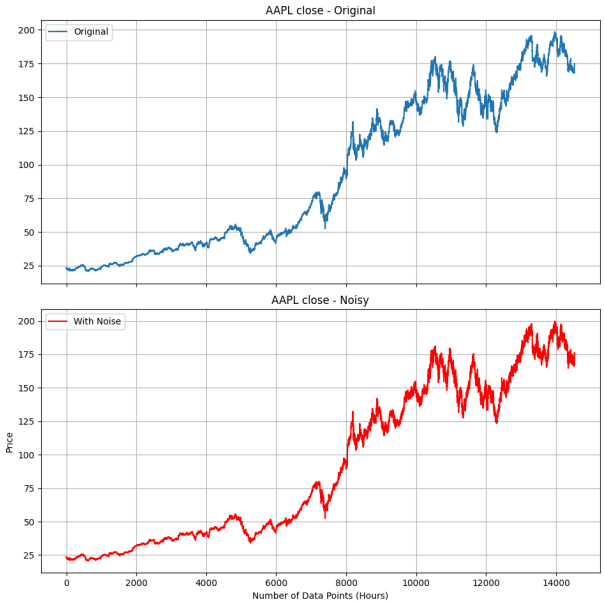

Figure 9. AAPL Original and with Noise

B. Denoising the Data Under Different Wavelet-Based Techniques

To denoise the noisy data, the 'pywt' Python library was used. This library utilizes discrete wavelet transforms to solve the denoising problem. After applying noise to the high-frequency price data, it was removed using different wavelet-based techniques. In total, 6 different possibilities were tested: Haar with soft and hard thresholding, Daubechies with soft and hard thresholding, and Symlet with soft and hard thresholding. For illustrative purposes, Figure 10 below shows how the noisy WMT data appeared after applying a 5-order symlet wavelet with a soft threshold of 10.

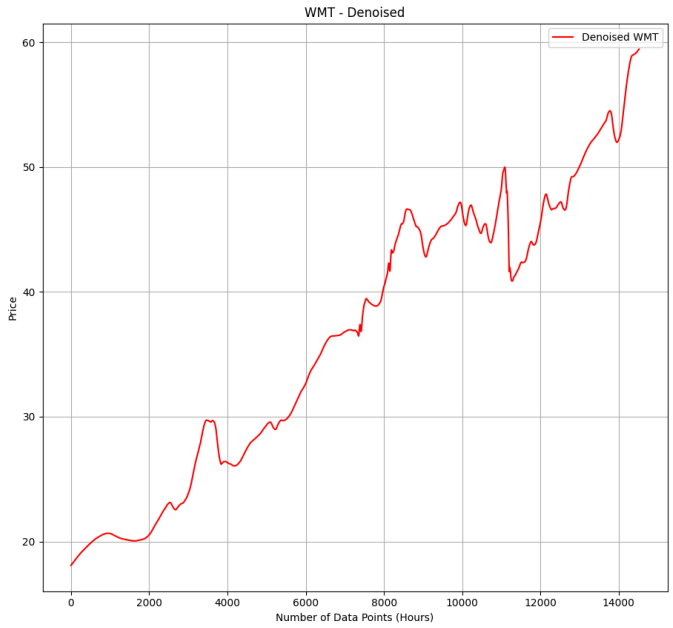

Figure 10. Example of Applying a 5-Order Symlet with a Soft Threshold of 10 to Noisy WMT Data

While it may appear that the ideal post-wavelet denoising price would appear as it does in Figure 10, this is not the case. While applying such a high-order high-threshold wavelet allows for most of the

noise to visually subside, this consequently means that a lot of the signal has also gone. For this reason, there is a necessary balance required between signal and noise and one of the key metrics for evaluating wavelet denoising is the signal-to-noise ratio. This metric and an additional one will be discussed in the next section to fine-tune the denoising process.

C. Evaluation of Results and Fine-Tuning

To evaluate the denoising results, two metrics were used: mean-squared error (MSE) and signal-to-noise ratio (SNR) (Liu and Cheng). Mean-squared error is calculated by taking the squared difference between two measurements (i.e., the denoised signal and original signal) and then taking the average of these squared differences:

$$MSE = \frac{1}{n}\sum_{k=1}^{n}(denoised_k - original_k)^2. \quad (32)$$

The SNR can be found by taking the logarithm of the ratio between the sum of the original signal values squared and the squared differences between the original signal and denoised signal values. This logarithm is then multiplied by a factor of 10:

$$SNR = 10 \cdot log_{10}\left(\frac{\sum_{k=1}^{n} original_k^2}{\sum_{k=1}^{n}(denoised_k - original_k)^2}\right). \quad (33)$$

These metrics are useful for fine-tuning the parameters of the denoising algorithms. A nested for loop was used in Python to test each wavelet (Haar, Daubechies, and Symlet) and threshold (hard and soft) combination for a range of orders and thresholds. Since Haar is just a Daubechies wavelet of order 1, any order above 1 was tested only on Daubechies and Symlet wavelets. The maximum order tested on was 20 and the threshold range varied from 1 to 20. For a set of parameters to be considered optimized, the denoised wavelet had to have both the best SNR and MSE metrics (a higher SNR is better; a lower MSE is better) compared to other previously tested wavelets. In other words, if the SNR of a wavelet tested was the best so far but its MSE was not then it was not considered to be optimized and was not selected to be the best wavelet. Fine-tuning of parameters was conducted on each stock.

For each stock, the parameters reported in Table 1 were found to be the best.

Table 1. Optimized Parameters for Wavelet Denoising

	TSLA	WMT	JNJ	INTC	AAPL
Wavelet Family and Order	Symlet 10	Symlet 11	Symlet 16	Symlet 17	Daubechies 4
Thresholding Type	Soft	Hard	Hard	Hard	Hard
Thresholding Value	8.0	2.0	8.0	2.0	8.0

After finding the parameters for each stock, the SNR and MSE metrics were found. These are reported in Table 2 directly below.

Table 2. Wavelet Denoising SNR and MSE Metrics

	TSLA	WMT	JNJ	INTC	AAPL
SNR	34.79 dB	42.84 dB	41.62 dB	40.85 dB	39.04 dB

MSE	8.68	0.08	1.22	0.14	1.47

As can be seen from these results, only wavelets from the Daubechies and symlet families were viewed as being optimal in terms of SNR and MSE metrics, and symlet was non-optimal just once. When I was selecting which parameters to use, I weighed having a high SNR as being a more important factor than having a low MSE. In other words, if for one type of wavelet family the MSE was very low but the SNR was noticeably higher, I chose not to opt for these parameters and went instead with the one with a slightly higher MSE but superior SNR.

D. Identifying Underlying Patterns or Trends in Data

To visually assess the impact of denoising on identifying patterns or trends in the original price data, I overlaid both the denoised and original time series for each stock.

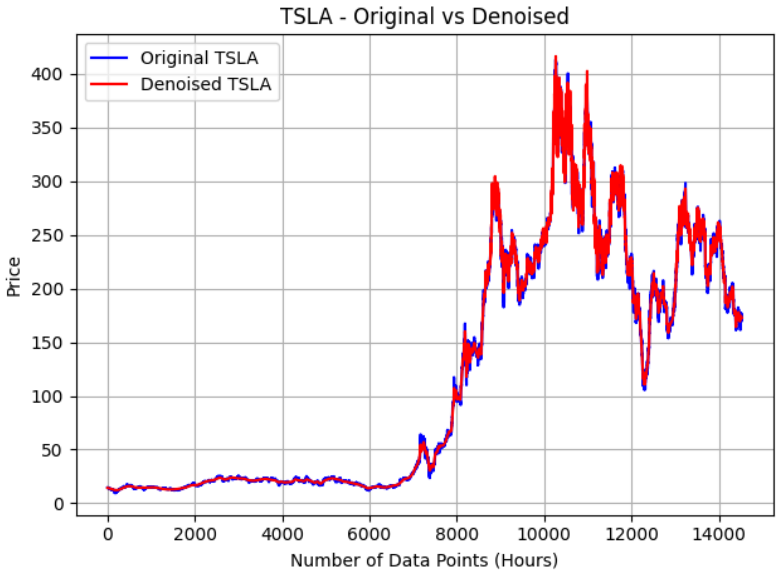

Figure 11. Overlay of TSLA Original and Fine-Tuned Denoised

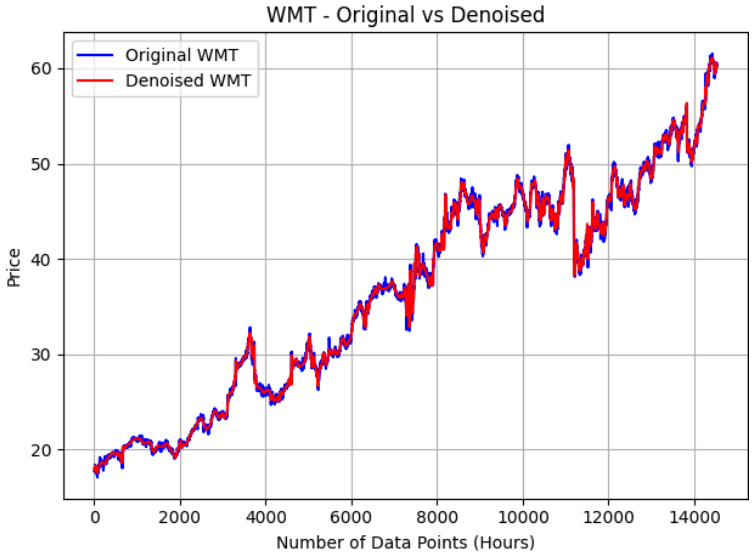

Figure 12. Overlay of WMT Original and Fine-Tuned Denoised

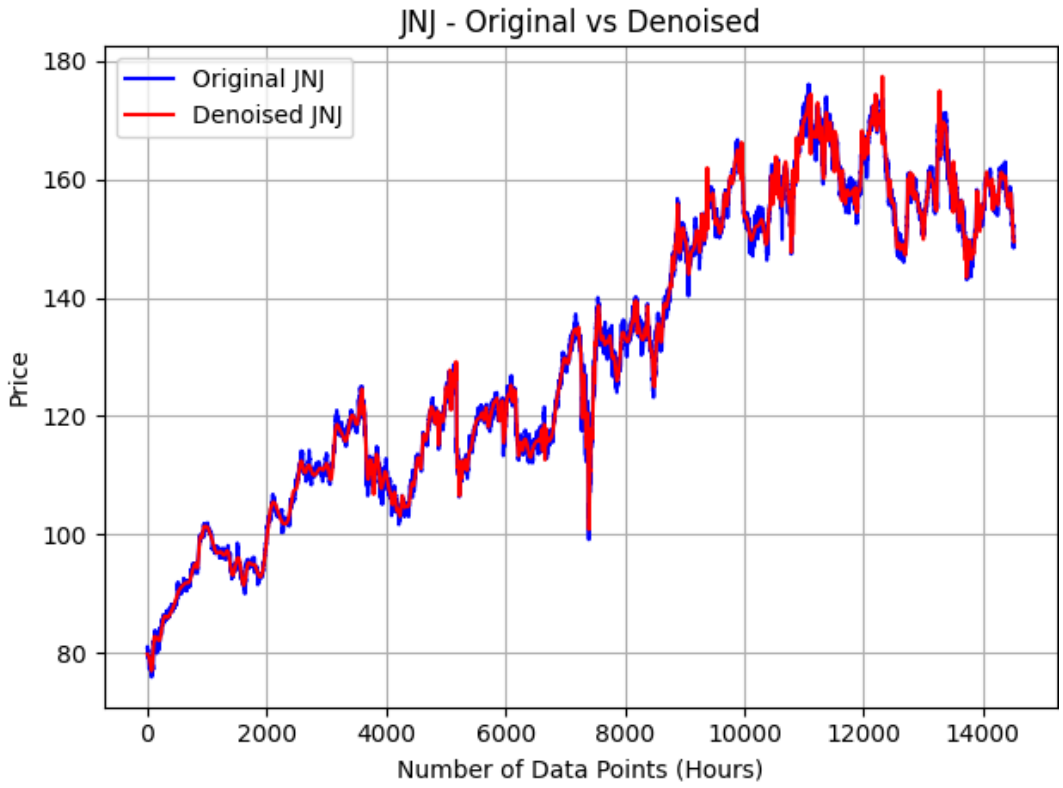

Figure 13. Overlay of JNJ Original and Fine-Tuned Denoised

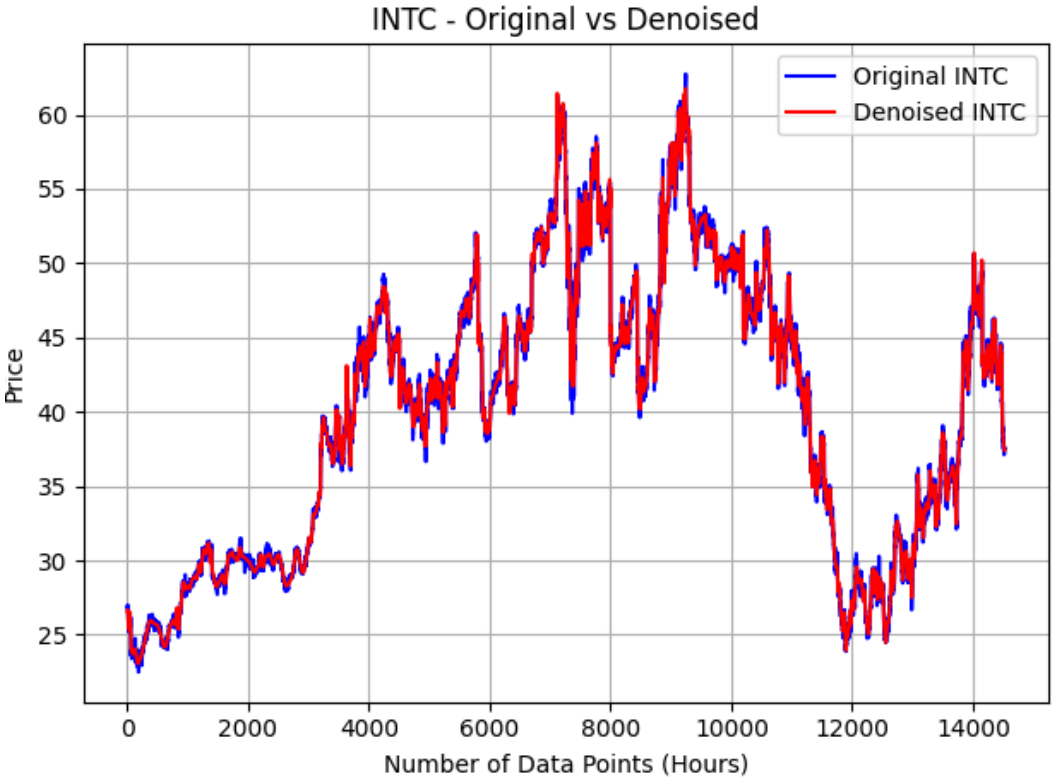

Figure 14. Overlay of INTC Original and Fine-Tuned Denoised

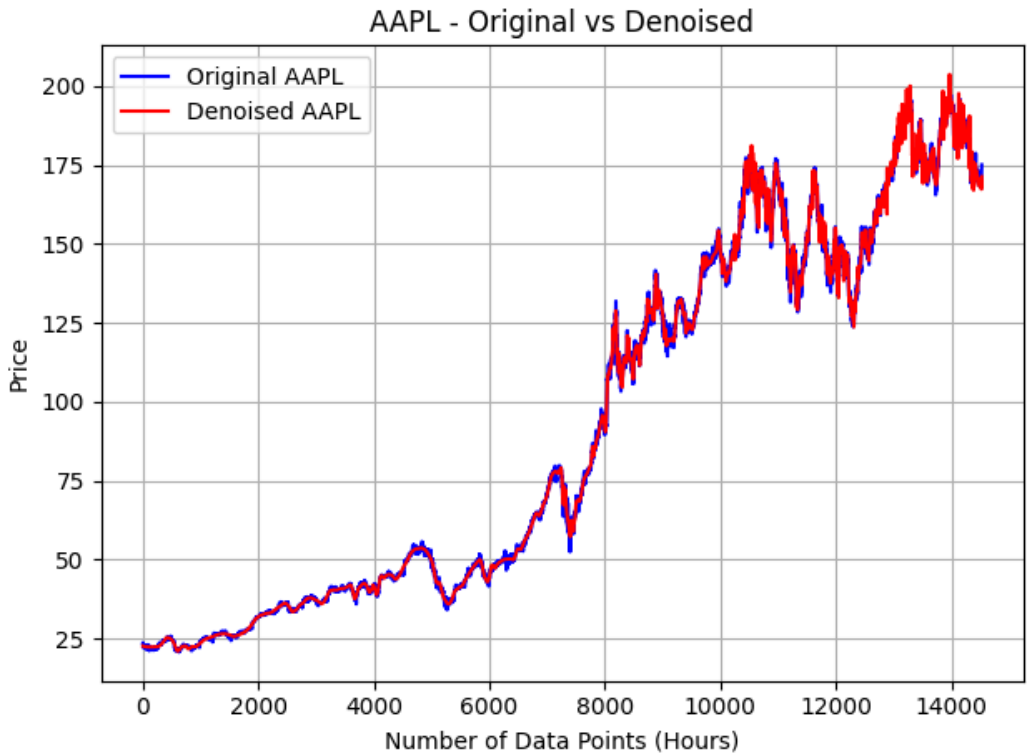

Figure 15. Overlay of AAPL Original and Fine-Tuned Denoised

It is noticeable in Figures 11 – 15 that the denoised stock price generally has fewer extrema compared to the original stock data. The best example of this seems to be in the WMT stock, where the denoised stock price is clearly enclosed on either side by the greater volatility of the actual stock price for most of the 14,000 hours. These results are similar to what was observed in a paper titled "Cross-Correlation Multifractal Analysis of Technological Innovation, Financial Market and Real Economy Indices", where the magnitude of logarithmic returns was generally lower for the denoised returns compared to the original returns (Ke et al.). Of course, there are times where this is not true since markets are still very volatile at times and the goal is to drive out noise while still retaining some kind of signal. If the price series was to be completely smoothed over, there would be very little noise but also likely not much signal. This is an ever-present tradeoff and explains why there are still notable spikes even after the data is denoised.

Analyzing these visuals in Figures 11 – 15 is a helpful way for determining if the denoised data is capable of identifying any trends. An additional way to determine if denoising the data reveals any underlying patterns or trends is by calculating the volatility (i.e., standard deviation) of both the original and denoised logarithmic returns. As was expected, the volatility of the denoised logarithmic returns was always lower than the original logarithmic returns, as shown below in Table 3.

Table 3. Annualized Volatility of Logarithmic Returns – Original vs Denoised

	TSLA	WMT	JNJ	INTC	AAPL
Original	52.55%	20.41%	18.15%	32.57%	26.92%
Denoised	38.73%	6.32%	8.87%	14.01%	17.77%
SNR	34.79 dB	42.84 dB	41.62 dB	40.85 dB	39.04 dB

To test whether wavelet denoising performed better on low volatility stocks relative to high volatility stocks, the original volatility for each stock was compared to its SNR. It appears that there is some sort of relationship since the two lowest volatility stocks, WMT and JNJ, also had the two highest SNRs. Similarly, TSLA had the highest volatility and the lowest SNR. This seems to suggest that wavelet denoising may perform better on stocks with lower volatility. This conclusion is further supported upon reevaluation of Figures 11 – 15, where the denoised versions of WMT and JNJ appear smoother relative to its original price compared to denoised TSLA relative to original TSLA. Both AAPL and INTC fall in the middle range of original volatilities, and similarly in the middle of SNRs. Nevertheless, across all 5 stocks the volatility saw a noticeable reduction and it can therefore be concluded that wavelet denoising did help in identifying trends. This is also visually clear from Figures 11 – 15.

Regarding the interpretation of SNR values, a "good" or "bad" value can differ widely across applications. Figures 11 – 15 above and Table 3 provide evidence that denoising was accomplished, as there is less noise relative to signal in the Figures and volatility decreased across all 5 assets. For financial time series, these values for SNR may be lower compared to other denoised time series. I do not know for certain why this may be the case, but my suspicion is that different levels of data frequency will provide different SNRs. For instance, minutely or hourly data may be much noisier compared to daily or monthly data, hence a comparatively lower SNR. Therefore, one may reasonably expect lower SNRs for higher-frequency, intraday data compared to a frequency like daily or monthly data.

IV. Conclusion

The main objective of this report was to determine if wavelet denoising could be applied to hourly historical price data to identify underlying patterns or trends. The data for 5 stocks were collected between 2016 and 2024. Prior to denoising the data, Gaussian noise was applied to it. Then, denoising was accomplished in three main steps: (1) deconstructing the time series into wavelets and their wavelet coefficients, (2) applying thresholding techniques (e.g., hard and soft) to set any coefficients that don't meet the threshold requirements equal to 0, and (3) reconstructing the time series using the thresholded wavelet coefficients. Furthermore, an additional step was added where the parameters for the wavelet were optimized with respect to SNR and MSE metrics.

It was found that the symlet wavelet was most effective for these 5 stocks, as TSLA, WMT, JNJ, and INTC used this family after being optimized while only AAPL used the Daubechies wavelet. Similarly, graphs of the denoised vs original prices were plotted and a comparison of volatility before and after denoising was conducted. It was found that lower volatility in the original price data generally meant improved SNRs, while higher volatility stocks saw lower SNRs. The range of SNRs found was roughly between 35 and 40. Overall, the denoised prices exhibited lower volatility compared to the original prices. This indicates that denoising assisted in identifying underlying patterns and trends in the time series data, and this claim is further supported by figures of denoised prices appearing smoother relative to their original counterparts.

For future research, it would be interesting to see how the optimized parameters change over time. Somewhat stable or stationary optimal parameters may be useful in future modeling of time series data via wavelet denoising, or other forms of signal processing. Additionally, variations of this project could be conducted on higher-frequency price data and a larger pool of assets. Lastly, traders may be interested in conducting out-of-sample studies on wavelet denoising and seeing how a strategy may perform by attempting to predict future asset prices based on current denoised prices.

V. Appendices

A. Theorems

Empirical mode decomposition can be done via the following algorithm ("Empirical Mode Decomposition: Theory & Applications"):

A. Find all the local maxima and minima of the time series $x(t)$.
B. Create the upper and lower envelopes, $e_{u(t)}$ and $e_{l(t)}$ respectively, of the time series $x(t)$ by cubic spline interpolation of the local maxima and minima found in the prior step.
   a. Cubic spline interpolation involves a set of piecewise cubic functions where two points (x_i, y_i) and (x_{i+1}, y_{i+1}) are joined by a cubic polynomial $S_i(x) = a_i x^3 + b_i x^2 + c_i + d_i$ for $x_i \leq x \leq x_{i+1}$ and $i = 1, \ldots, n-1$ ("Python Numerical Methods"). Therefore, to find the interpolating function the four coefficients a_i, b_i, c_i, and d_i have to be determined for each of the cubic functions. For a set of n data points (e.x., n stock prices) there are n-1 cubic functions to solve for with each equation containing 4 unknown coefficients. So there is a total of $4(n-1)$ unknown coefficients. To obtain these values, consider the following constraints:
      i. $S_i(x_i) = y_i$
      ii. $S_i(x_{i+1}) = y_{i+1}$
      iii. $S'_i(x_{i+1}) = S'_{i+1}(x_{i+1})$
      iv. $S''_i(x_{i+1}) = S''_{i+1}(x_{i+1})$
      v. $S''_1(x_1) = 0$
      vi. $S''_{n-1}(x_n) = 0$

      For a given number of n data points, there will now be a $4(n-1)$ system of equations and the cubic spline interpolation function can be solved.
C. Once the upper and lower evolves of the time series $x(t)$ are found via cubic spline interpolation, calculate the mean function of the envelope $m_1(t) = \frac{e_{u(t)} + e_{l(t)}}{2}$.
D. Set the difference between the time series function and mean function $d_1(t) = x(t) - m_1(t)$.
E. If $d_1(t)$ is a zero-mean function, there is no iteration and $d_1(t)$ is the first IMF. If not, $d_1(t)$ replaces $x(t)$ and steps 1 through 4 are repeated until an IMF is found.

B. Data

All data for the 5 stocks used has been submitted on Brightspace with this report. The CSV file is entitled "AMS_522_Project_1hr_Stock_Data." The data was obtained from TradingView.

C. Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator
import pywt

def add_noise_to_stock_data(file_path):
```

```
    # Load the closing 1hr stock prices from the Excel file
    data = pd.read_excel(file_path, usecols=['time', 'TSLA close', 'WMT close', 'JNJ close', 'INTC close',
'AAPL close'])

    # Create a data frame to store noisy data
    noisy_data = data.copy()

    # Columns containing the stock close prices
    stock_columns = ['TSLA close', 'WMT close', 'JNJ close', 'INTC close', 'AAPL close']

    # Apply multiple of Gaussian noise to each stock column and plot
    for column in stock_columns:
        # Creating Gaussian noise
        noise = np.random.normal(0, 1, len(data))

        # Applying noise to the stock prices
        noisy_data[column] = data[column] * (1 + data[column].shift(1).fillna(data[column].iloc[0]) *
(1/10000) * noise)

        # Create figure with two subplots, one above the other
        fig, axs = plt.subplots(2, 1, figsize=(10, 10), sharex=True)

        # Plot original data on first subplot
        axs[0].plot(data[column], label='Original')
        axs[0].set_title(f"{column} - Original")
        axs[0].legend()
        axs[0].grid(True)

        # Plot noisy data on second subplot
        axs[1].plot(noisy_data[column], label='With Noise', color='red')
        axs[1].set_title(f"{column} - Noisy")
        axs[1].set_xlabel('Number of Data Points (Hours)')
        axs[1].set_ylabel('Price')
        axs[1].legend()
        axs[1].grid(True)

        plt.tight_layout()
        plt.show()

    return noisy_data, data

def haar_denoising(data, threshold, method='hard'):
    # Ensure no NaN values are present
    if data.isna().any():
```

```
        data = data.fillna(method='ffill')

    # Haar wavelet decomposition
    coeffs = pywt.wavedec(data, 'haar', mode='per')

    # Select thresholding function
    threshold_mode = 'hard' if method == 'hard' else 'soft'

    # Apply thresholding to coefficients
    coeffs[1:] = [pywt.threshold(c, value=threshold, mode=threshold_mode) for c in coeffs[1:]]

    # Reconstruct signal from thresholded coefficients
    denoised_data = pywt.waverec(coeffs, 'haar', mode='per')

    return denoised_data

def daubechies_denoising(data, threshold, order, method='soft'):
    # Ensure no NaN values are present
    if data.isna().any():
        data = data.fillna(method='ffill')

    # Specify wavelet order
    wavelet = f'db{order}'

    # Daubechies wavelet decomposition
    coeffs = pywt.wavedec(data, wavelet, mode='symmetric')

    # Select thresholding function
    threshold_mode = 'hard' if method == 'hard' else 'soft'

    # Apply thresholding to coefficients
    coeffs[1:] = [pywt.threshold(c, value=threshold, mode=threshold_mode) for c in coeffs[1:]]

    # Reconstruct signal from thresholded coefficients
    denoised_data = pywt.waverec(coeffs, wavelet, mode='symmetric')

    return denoised_data

def symlet_denoising(data, threshold, order, method='soft'):
    # Ensure no NaN values are present
    if data.isna().any():
        data = data.fillna(method='ffill')

    # Specify the wavelet order
```

```
    wavelet = f'sym{order}'

    # Symlet wavelet decomposition
    coeffs = pywt.wavedec(data, wavelet, mode='symmetric')

    # Select thresholding function
    threshold_mode = 'hard' if method == 'hard' else 'soft'

    # Apply thresholding to coefficients
    coeffs[1:] = [pywt.threshold(c, value=threshold, mode=threshold_mode) for c in coeffs[1:]]

    # Reconstruct signal from thresholded coefficients
    denoised_data = pywt.waverec(coeffs, wavelet, mode='symmetric')

    return denoised_data

def calculate_mse(original, denoised):
    # Calculate MSE between original and denoised data
    return np.mean((denoised - original) ** 2)

def calculate_snr(original, denoised):
    # Calculate signal-to-noise ratio (dB) between original and denoised data
    signal_power = np.sum(original ** 2)
    noise_power = np.sum((denoised  - original) ** 2)
    return 10 * np.log10(signal_power / noise_power)

def optimize_denoising(data, original, wavelet='haar', max_order=5, threshold_range=(1, 10),
method='soft'):
    # After inputting noisy data, original data, wavelet type, and wavelet type, this function loops through
    # different thresholds and order values to determine which parameters

    # Initialize variables
    # Negative infinity is used for initializing SNR to ensure that any real SNR calculated is larger
    # Similarly, positive infinity is used for initializing MSE to ensure that any real MSE calculated is
smaller
    best_snr = -np.inf
    best_mse = np.inf
    best_params = {}

    # Nested for loops to go through all order and threshold combinations
    for order in range(1, max_order):
        for threshold in np.linspace(*threshold_range, num=10):
```

```
        # Construct wavelet using different pre-defined functions based on type and order input into
optimize_denoising
        wavelet_name = f'{wavelet}{order}' if wavelet in ['db', 'sym'] else wavelet

        # Denoise data based on which type of wavelet is input
        if wavelet == 'haar':
            denoised_data = haar_denoising(data, threshold, method)
        elif wavelet in ['db', 'sym']:
            denoised_data = daubechies_denoising(data, threshold, order + 1, method) if wavelet == 'db'
else symlet_denoising(data, threshold, order + 1, method)

        # Calculate MSE and SNR
        mse = calculate_mse(original, denoised_data)
        snr = calculate_snr(original, denoised_data)

        # Check based on MSE and SNR combined if selected parameters are best so far
        if snr > best_snr:
            best_snr = snr
            best_mse = mse
            best_params = {'wavelet': wavelet_name, 'threshold': threshold, 'method': method, 'order':
order}

    return best_params, best_snr, best_mse

file_path = r'C:\Users\willk\OneDrive\Desktop\AMS522 HW7\AMS_522_Project_1hr_Stock_Data.xlsx'
noisy_data, data = add_noise_to_stock_data(file_path)

# Example denoising for illustrative purposes
denoised_WMT = symlet_denoising(noisy_data['WMT close'], 10, 5, method='soft')
plt.plot(denoised_WMT, label='Denoised WMT', color='red')
plt.title(f"WMT - Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# optimize_denoising function
for i in ['TSLA','WMT','JNJ','INTC','AAPL']:
    best_params, best_snr, best_mse = optimize_denoising(noisy_data[f'{i} close'], data[f'{i} close'],
wavelet='db', max_order=19, threshold_range=(2, 20), method='hard')
    print(f"Best Parameters - {i}: {best_params}")
    print(f"Best SNR - {i}: {best_snr} dB")
```

```
    print(f"Best MSE - {i}: {best_mse}")

# Plot of TSLA - Denoised and Original
original_TSLA = data['TSLA close']
denoised_TSLA_np = symlet_denoising(noisy_data['TSLA close'], 8.0, 10, method='soft')
denoised_TSLA = pd.Series(denoised_TSLA_np, index=original_TSLA.index)
log_returns_original_TSLA = np.log(original_TSLA / original_TSLA.shift(1))
log_returns_denoised_TSLA = np.log(denoised_TSLA / denoised_TSLA.shift(1))
original_volatility_TSLA = log_returns_original_TSLA.std()
denoised_volatility_TSLA = log_returns_denoised_TSLA.std()
annualized_volatility_original_TSLA = original_volatility_TSLA * np.sqrt(252*6.5)
annualized_volatility_denoised_TSLA = denoised_volatility_TSLA * np.sqrt(252*6.5)
print("Original TSLA Volatility:", annualized_volatility_original_TSLA)
print("Denoised TSLA Volatility:", annualized_volatility_denoised_TSLA)
plt.plot(original_TSLA.index, original_TSLA, label='Original TSLA', color='blue')
plt.plot(original_TSLA.index, denoised_TSLA, label='Denoised TSLA', color='red')
plt.title(f"TSLA - Original vs Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plot of WMT - Denoised and Original
original_WMT = data['WMT close']
denoised_WMT_np = symlet_denoising(noisy_data['WMT close'], 2.0, 11, method='hard')
denoised_WMT = pd.Series(denoised_WMT_np, index=original_WMT.index)
log_returns_original_WMT = np.log(original_WMT / original_WMT.shift(1))
log_returns_denoised_WMT = np.log(denoised_WMT / denoised_WMT.shift(1))
original_volatility_WMT = log_returns_original_WMT.std()
denoised_volatility_WMT = log_returns_denoised_WMT.std()
annualized_volatility_original_WMT = original_volatility_WMT * np.sqrt(252*6.5)
annualized_volatility_denoised_WMT = denoised_volatility_WMT * np.sqrt(252*6.5)
print("Original WMT Volatility:", annualized_volatility_original_WMT)
print("Denoised WMT Volatility:", annualized_volatility_denoised_WMT)
plt.plot(original_WMT.index, original_WMT, label='Original WMT', color='blue')
plt.plot(original_WMT.index, denoised_WMT, label='Denoised WMT', color='red')
plt.title(f"WMT - Original vs Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
plt.show()

# Plot of JNJ - Denoised and Original
original_JNJ = data['JNJ close']
denoised_JNJ_np = symlet_denoising(noisy_data['JNJ close'], 8.0, 16, method='hard')
denoised_JNJ = pd.Series(denoised_JNJ_np, index=original_JNJ.index)
log_returns_original_JNJ = np.log(original_JNJ / original_JNJ.shift(1))
log_returns_denoised_JNJ = np.log(denoised_JNJ / denoised_JNJ.shift(1))
original_volatility_JNJ = log_returns_original_JNJ.std()
denoised_volatility_JNJ = log_returns_denoised_JNJ.std()
annualized_volatility_original_JNJ = original_volatility_JNJ * np.sqrt(252*6.5)
annualized_volatility_denoised_JNJ = denoised_volatility_JNJ * np.sqrt(252*6.5)
print("Original JNJ Volatility:", annualized_volatility_original_JNJ)
print("Denoised JNJ Volatility:", annualized_volatility_denoised_JNJ)
plt.plot(original_JNJ.index, original_JNJ, label='Original JNJ', color='blue')
plt.plot(original_JNJ.index, denoised_JNJ, label='Denoised JNJ', color='red')
plt.title(f"JNJ - Original vs Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plot of INTC - Denoised and Original
original_INTC = data['INTC close']
denoised_INTC_np = symlet_denoising(noisy_data['INTC close'], 2.0, 17, method='hard')
denoised_INTC = pd.Series(denoised_INTC_np, index=original_INTC.index)
log_returns_original_INTC = np.log(original_INTC / original_INTC.shift(1))
log_returns_denoised_INTC = np.log(denoised_INTC / denoised_INTC.shift(1))
original_volatility_INTC = log_returns_original_INTC.std()
denoised_volatility_INTC = log_returns_denoised_INTC.std()
annualized_volatility_original_INTC = original_volatility_INTC * np.sqrt(252*6.5)
annualized_volatility_denoised_INTC = denoised_volatility_INTC * np.sqrt(252*6.5)
print("Original INTC Volatility:", annualized_volatility_original_INTC)
print("Denoised INTC Volatility:", annualized_volatility_denoised_INTC)
plt.plot(original_INTC.index, original_INTC, label='Original INTC', color='blue')
plt.plot(original_INTC.index, denoised_INTC, label='Denoised INTC', color='red')
plt.title(f"INTC - Original vs Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
plt.show()

# Plot of AAPL - Denoised and Original
original_AAPL = data['AAPL close']
denoised_AAPL_np = daubechies_denoising(noisy_data['AAPL close'], 8.0, 4, method='hard')
denoised_AAPL = pd.Series(denoised_AAPL_np, index=original_AAPL.index)
log_returns_original_AAPL = np.log(original_AAPL / original_AAPL.shift(1))
log_returns_denoised_AAPL = np.log(denoised_AAPL / denoised_AAPL.shift(1))
original_volatility_AAPL = log_returns_original_AAPL.std()
denoised_volatility_AAPL = log_returns_denoised_AAPL.std()
annualized_volatility_original_AAPL = original_volatility_AAPL * np.sqrt(252*6.5)
annualized_volatility_denoised_AAPL = denoised_volatility_AAPL * np.sqrt(252*6.5)
print("Original AAPL Volatility:", annualized_volatility_original_AAPL)
print("Denoised AAPL Volatility:", annualized_volatility_denoised_AAPL)
plt.plot(original_AAPL.index, original_AAPL, label='Original AAPL', color='blue')
plt.plot(original_AAPL.index, denoised_AAPL, label='Denoised AAPL', color='red')
plt.title(f"AAPL - Original vs Denoised")
plt.xlabel('Number of Data Points (Hours)')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

VI. References

"Chapter 3 Haar Bases." *Haar Wavelets*, www.cis.upenn.edu/~cis5150/cis515-20-sl-Haar.pdf.

"Continuous Wavelet Transform." *Stanford Center for Computer Research in Music and Acoustics*, ccrma.stanford.edu/~jos/sasp/Continuous_Wavelet_Transform.html.

"Daubechies Wavelet." *Daubechieswavelet-Wolfram Language Documentation*, reference.wolfram.com/language/ref/DaubechiesWavelet.html.

"Empirical Mode Decomposition: Theory & Applications." International Research Publication House, 2014, www.ripublication.com/irph/ijeee_spl/ijeeev7n8_14.pdf.

Govindan, Vidya, et al. "A Hardware Trojan Attack on FPGA-Based Cryptographic Key Generation: Impact and Detection." Journal of Hardware and Systems Security, vol. 2, 2018, doi:10.1007/s41635-018-0042-5.

"Haar Wavelet Filterbanks." *Georgia Tech School of Electrical and Computer Engineering*, mdav.ece.gatech.edu/ece-6250-fall2019/notes/11-notes-6250-f19.pdf.

Hall, Matt. “What Is the Gabor Uncertainty Principle?” Agile, 15 Jan. 2014, agilescientific.com/blog/2014/1/15/what-is-the-gabor-uncertainty-principle.html.

“Introduction to Wavelet Families.” *MathWorks*, www.mathworks.com/help/wavelet/gs/introduction-to-the-wavelet-families.html#f3-1008627.

Liu, Bing, and Huanhuan Cheng. “De-noising classification method for financial time series based on ICEEMDAN and wavelet threshold, and its application”. *EURASIP Journal on Advances in Signal Processing*, vol. 2024, no. 19, 2024, doi:10.1186/s13634-024-01115-5.

Ke, Jinchuan, et al. "Cross-Correlation Multifractal Analysis of Technological Innovation, Financial Market and Real Economy Indices." *Fractal and Fractional*, vol. 7, 2023, art. 267, doi:10.3390/fractalfract7030267.

“Python Numerical Methods.” Cubic Spline Interpolation - Python Numerical Methods, pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter17.03-Cubic-Spline-Interpolation.html.

Lindsay, R. W., et al. "The Discrete Wavelet Transform and the Scale Analysis of the Surface Properties of Sea Ice." IEEE Transactions on Geoscience and Remote Sensing, vol. 34, no. 3, May 1996, pp. 771-787, doi: 10.1109/36.499782.

“Symlet Wavelet.” *Symletwavelet-Wolfram Language Documentation*, reference.wolfram.com/language/ref/SymletWavelet.html.

“The Uncertainty Principle - UGA Math.” *University of Georgia*, www.math.uga.edu/sites/default/files/uncertainty.pdf.

“Wavelet Signal Analyzer.” *MathWorks*, www.mathworks.com/help/wavelet/gs/what-is-a-wavelet.html.

“What Are Wavelet Transforms?” *MathWorks*, www.mathworks.com/discovery/wavelet-transforms.html#:~:text=What%20Are%20Wavelet%20Transforms%3F,transients%2C%20or%20slowly%20varying%20trends.

VII. Python Packages Used

A. Pandas
B. Numpy
C. Matplotlib
D. PyWavelets (Pywt)