Analyzing Neural Time Series Data: Theory and Practice

Mike X Cohen

The MIT Press

From The MIT Press

MITCogNet

This book was set in ITC Stone Serif Std by Toppan Best-set Premedia Limited, Hong Kong. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Cohen, Mike X, 1979–
Analyzing neural time series data : theory and practice / Mike X Cohen.
pages cm.—(Issues in clinical and cognitive neuropsychology)
Includes bibliographical references and index.
ISBN 978-0-262-01987-3 (hardcover : alk. paper) 1. Neural networks (Neurobiology) 2. Neural networks (Computer science) 3. Computational neuroscience. 4. Artificial intelligence—Biological applications. I. Title.
QP363.3.C633 2014
612.8'2339—dc23
2013016461

10 9 8 7 6 5 4 3 2 1

11 The Discrete Time Fourier Transform, the FFT, and the Convolution Theorem

The Fourier transform is an incredibly important signal-processing technique in time-series data analysis as well as in many other branches of science, engineering, and technology. It is the computational backbone of most EEG data analyses. In turn, the backbone of the Fourier transform is the dot product, which is why the previous chapter was dedicated to that topic.

The Fourier transform works by computing the dot product between the signal (such as EEG data) and sine waves of different frequencies (the kernels). Remember from chapter 2 that sine waves have three characteristics: frequency (how fast, measured in cycles per second or hertz), power (or amplitude; power is amplitude squared, so these terms are sometimes used interchangeably), and phase (the timing of the sine wave, measured in radians or degrees). Thus, the result of the Fourier transform is a three-dimensional (3-D) representation of the time series data in which the three dimensions are frequency, power, and phase. This 3-D Fourier representation contains all of the information in the time series data, and the time series data can be perfectly reconstructed from the result of its Fourier transform (this will be demonstrated later in this chapter).

To perform a Fourier transform, you need to know how to compute the dot product, and you need to know how to create a sine wave. You already know how to compute the dot product; now it is time to learn about creating sine waves.

11.1 Making Waves

Sine waves can be represented mathematically using the following formula:

$$A\sin(2\pi ft+\theta) \tag{11.1}$$

in which A is the amplitude of the sine wave, π is pi, or 3.141 . . . , f is the frequency of the sine wave, t is time (this could also be space, but only time-varying sine waves are discussed here), and θ is the phase angle offset, which is related to the value of the sine wave

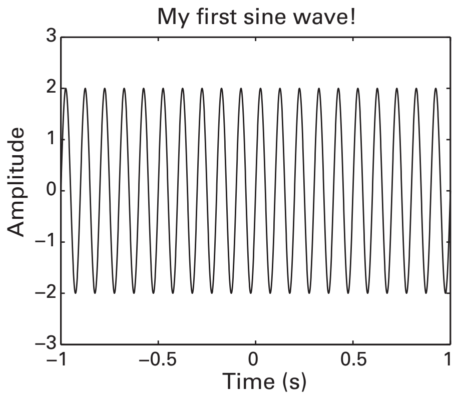

Figure 11.1
A sine wave that was created using the expression $2\sin(2\pi 10t + 0)$ (thus, amplitude of 2 and frequency of 10 Hz).

at time = 0. If the amplitude is 1 and the phase is zero, equation 11.1 reduces to the perhaps more familiar-looking $\sin(2\pi ft)$. Figure 11.1 shows a sine wave created using equation 11.1.

Take a few minutes to explore the online Matlab code that created this figure by changing the sampling rate, the frequency, the phase, and the amplitude. Try to get a feel for the effects those parameters have on the resulting sine wave. You should notice that the frequency parameter changes the speed of the oscillation, and the number indicates the cycles per second and has units of hertz. Changing the sampling rate has little effect except that the sine wave will look more jagged when the sampling rate becomes small relative to the frequency. Changing the amplitude affects the height of the peaks and troughs of the sine wave. If left undefined, the amplitude is implicitly set to 1.

The most obvious feature of a sine wave is that it goes up and down over time. Why does it go up and down? You will learn in the next chapter that sine waves can be better conceptualized as a spiral in a 3-D space and that what you see in figure 11.1 is a projection onto two of the three dimensions. This gives the appearance of going up and down.

By adding together several sine waves of different frequencies, amplitudes, and phases, the resulting time series looks more complicated than any of the individual sine waves. Figure 11.2A shows five individual sine waves, and figure 11.2B shows the result of adding all of those sine waves together. Again, spend a few minutes going through the online Matlab code that generated this figure. Try changing the parameters of the sine waves to see how the summed time series looks, and try adding a few more sine waves to see what the effect is on the resulting sum. Note that if you want to add more sine waves, the variables `frex`, `phases`, and `amplitudes` must have the same number of elements (one for each sine

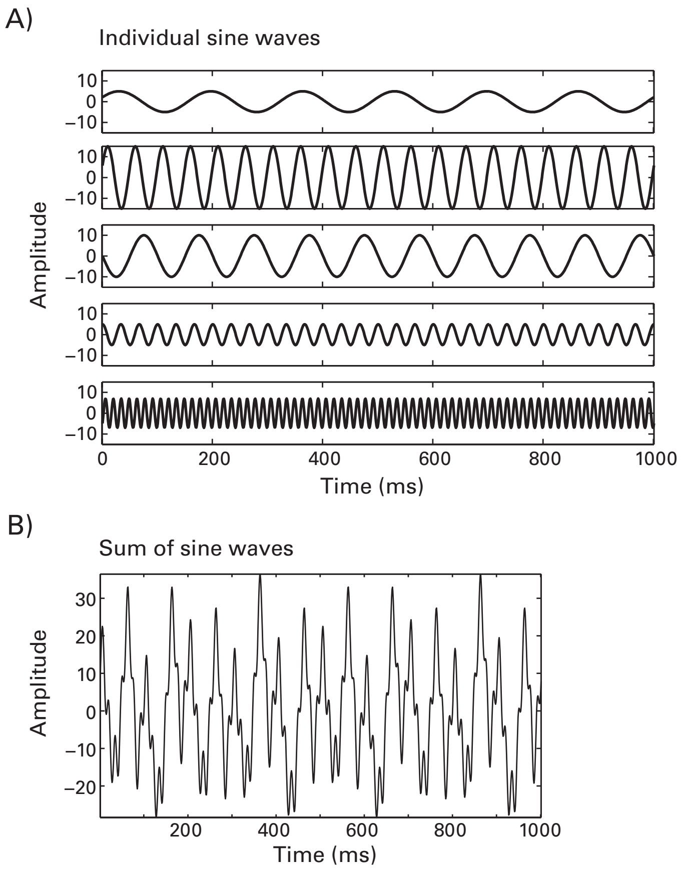

Figure 11.2
Several sine waves of differing amplitudes, frequencies, and phases, plotted separately (panel A) and after being added together (panel B).

wave). Try making `frex` longer than `phases` and `amplitudes`. Where does Matlab crash and why?

Although it is easy to distinguish the time series in figure 11.2B from real EEG data, adding some noise makes the signal look more like real data, as shown in figure 11.3.

11.2 Finding Waves in EEG Data with the Fourier Transform

In the previous examples, time series were created by summing collections of sine waves. However, this does not help you analyze real EEG data. With real data you have the opposite

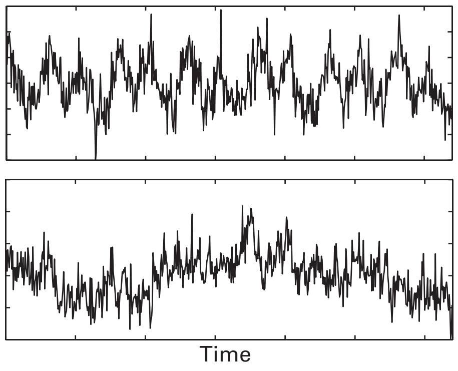

Figure 11.3
One of these time series was generated by summing different sine waves and adding noise; the other time series is real EEG data. Can you guess which is which? What evidence do you use to support your decision?

problem: you already have the time series, and you want to know which sine waves with which frequencies, amplitudes, and phases will reconstruct that time series. This is the purpose of the Fourier transform. Essentially, the Fourier transform works by computing the dot product between sine waves of different frequencies and the EEG data.

The best way to learn how the Fourier transform works is by programming the discrete time Fourier transform. It may seem tedious to program the Fourier transform from scratch when you could simply use the Matlab function `fft`. However, the Fourier transform is one of the two most important and fundamental mathematical expressions for time-frequency-based data analyses (the other is Euler's formula, which you will learn about in chapter 13). The better you understand how the Fourier transform works, the better you will understand the mechanics of the more advanced analyses.

11.3 The Discrete Time Fourier Transform

The idea behind the discrete time Fourier transform is fairly simple: create a sine wave and compute the dot product between that sine wave and the time series data. Then create another sine wave with a different frequency and compute the dot product between that sine wave and the same time series data. The number of sine waves you create, and the frequency of each sine wave, is determined by the number of data points in the time series data (more on this in a few paragraphs). The following Matlab code will perform a discrete time Fourier transform of random data.

```
N = 10; % length of sequence
data = rand(1,N); % random numbers
% initialize Fourier coefficients
fourier = zeros(size(data));
time = (0:N-1)/N; % time starts at 0; dividing by N normalizes to 1
% Fourier transform
for fi=1:N
    % create sine wave
    sine_wave = exp(-1i*2*pi*(fi-1).*time);
    % compute dot product between sine wave and data
    fourier(fi) = sum(sine_wave.*data);
end
```

There are two steps that occur within each iteration of this for-loop. The first step is that a sine wave is created. The frequency of that sine wave is defined by the looping index, and the looping index in turn is defined by the number of points in the data (in this case, the variable *N*). The equation for the sine wave looks different here compared to equation 11.1—you can still see the $2\pi ft$ part, but there is no "sin" function, and instead there is an `exp` and a `-1i`. This makes a complex sine wave, which is something you will learn about in chapter 13; for now, just keep in mind that this is a sine wave. Note that there are other equivalent ways of expressing this equation, such as a cosine wave plus an imaginary sine wave. The expression used above is preferred here because it will help you understand the link between the Fourier transform and Euler's formula.

You can also see that the frequency of the sine wave is defined by one less than the looping index (`fi-1`), whereas you might expect it to be defined by `fi`. The sine wave at each iteration corresponds to a frequency one step slower than that iteration. Thus, the first iteration of this loop produces a sine wave of zero frequency. This is also sometimes called the "DC component," where DC stands for direct current. A sine wave with zero frequency is a flat line, and thus the zero frequency captures the mean offset over the entire signal. For example, if you subtract the mean of the signal before computing its Fourier transform, the DC or zero frequency component will be zero.

The second step inside the loop of the Fourier transform computes the dot product between the sine wave and the data. As you learned in chapter 10, the dot product computes the mapping between two vectors, which can be thought of as a way to measure the similarity between those two vectors. In figure 10.1 you saw how the dot product works visually on two 2-D vectors; here, the dot product is computed between two 10-D vectors. Thus, the

Fourier transform works by computing the dot product between the signal and sine waves of different frequencies.

The frequencies are defined by a looping index, but it would be convenient to know what those frequencies are in hertz. To convert the frequencies to hertz, consider that the number of unique frequencies that can be extracted from a time series is exactly one-half of the number of data points in the time series, plus the zero frequency. This is due to the Nyquist theorem: you need at least two points per cycle to measure a sine wave, and thus half the number of points in the data corresponds to the fastest frequency that can be measured in those data. Thus, the number of unique frequencies that can be extracted from a time series of length N is $N/2 + 1$ (the +1 is for the zero frequency component). This also means that $N/2 + 1$ is the frequency resolution of the data. Because frequencies start at zero and go up to the Nyquist frequency, to convert frequencies from indices to hertz, you create linearly equally spaced numbers between zero and the Nyquist frequency in $N/2 + 1$ steps. This can be done in Matlab using the function `linspace`; the online Matlab code shows you how to do this.

For completeness, below is the formula for the discrete Fourier transform of variable x at frequency f.

$$X_f = \sum_{k=1}^{n} x_k e^{-i2\pi f(k-1)n^{-1}} \tag{11.2}$$

in which n refers to the number of data points in vector x, and the capital letter X_f is the Fourier coefficient of time series variable x at frequency f. Again, do not worry too much about the exponential and imaginary number; these are explained in detail in chapter 13. You might also sometimes see this equation written with the summation variable from $k = 0$ to $n - 1$, but equation 11.2 is written with indices starting at one instead of zero to facilitate comparison with the implementation in Matlab.

11.4 Visualizing the Results of a Fourier Transform

Now that you have computed the Fourier transform, how can you view the results? Remember that the Fourier transform provides a 3-D representation of the data in terms of frequency, power, and phase. Very often, the phase information is ignored when results of a Fourier analysis are shown, thus leaving two dimensions to show (the 3-D representation is considered below). Thus, the most typical way of showing results from a Fourier analysis is a 2-D plot with frequency on the x-axis and power (or amplitude) on the y-axis, as illustrated in figure 11.4.

Notice that figure 11.4 uses bars instead of lines to plot the results of the Fourier transform. Because frequencies are discretely sampled, it is appropriate to use bars or dots to plot

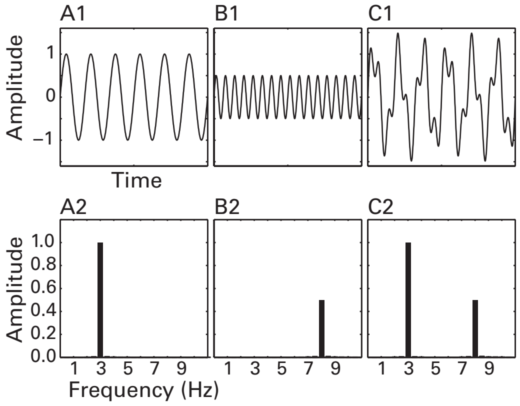

Figure 11.4
Two sine waves of 3 Hz and 8 Hz (panels A1 and B1), and their sum (panel C1) have a Fourier power spectrum that can be represented in plots of frequency (*x*-axis) by amplitude (*y*-axis) (see plots in the bottom row). The frequency of the sine wave corresponds to the position on the *x*-axis, and the amplitude of the sine wave corresponds to the position on the *y*-axis.

Fourier results instead of lines, which would imply a smooth transition between sampled points. However, with long time series, the discretization becomes very fine, and bar plots can be impractical for viewing, particularly for viewing several results on the same plot. Thus, in practice, line plots can be used because they increase visibility and the possibility for direct comparisons across Fourier results from different time series.

As mentioned earlier, although the result of a Fourier transform is often presented in 2-D plots for convenience, the result of a Fourier transform actually occupies a 3-D space. This is shown in figure 11.5. The Fourier transform was taken from randomly generated data (shown in figure 11.5A). Figure 11.5B shows that the Fourier transform of that time series data can be represented in a frequency-power-phase space. Note that the phase here refers to the position of the sine wave at each frequency when it crosses time = 0; this is slightly different from the phase angle time series. You will learn more about this in chapter 13. Figure 11.5C–D shows that rotating the 3-D space to show only two dimensions at a time allows power and phase spectra to be plotted in a format that you might be more used to seeing.

11.5 Complex Results and Negative Frequencies

How does this 3-D information come from the single variable `fourier` in the Matlab code shown earlier? If you look at the contents of this variable in Matlab, you will see that it has the form $a + ib$. This is the representation of a complex number and is crucial for extracting

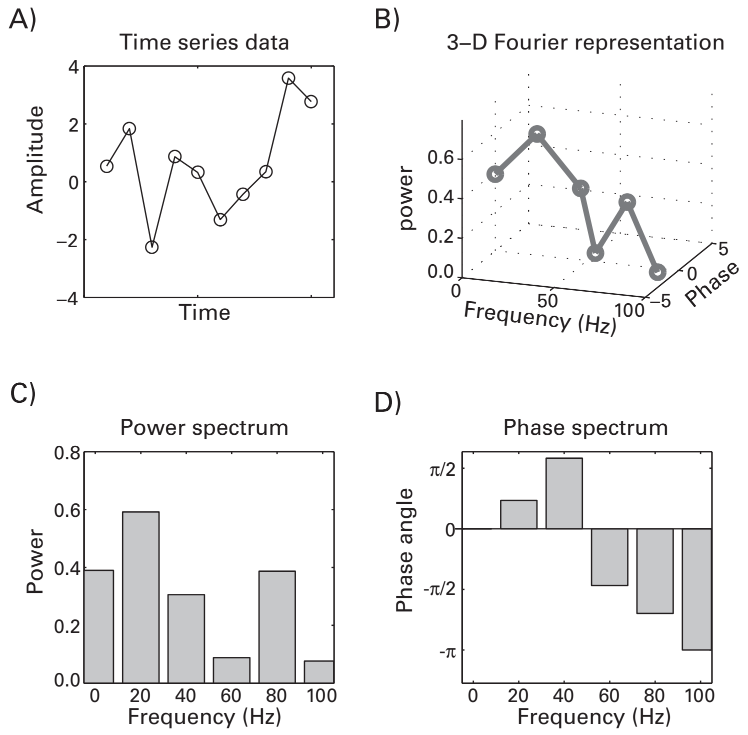

Figure 11.5
Panel A shows a time series of randomly generated data (sampled at 200 Hz). Panel B shows a frequency representation of the data as a line through 3-D Fourier space (frequency, power, and phase; see axis labels). Panels C and D show the projections of the 3-D Fourier result onto two dimensions at a time. Panel C shows only the frequency and power information from the Fourier transform, and panel D shows only the frequency and the phase information from the Fourier transform.

power and phase information, and will be covered in greater detail in chapter 13. For now, you can just plot `abs(fourier)` in Matlab, which gives you the amplitude of the Fourier series.

If you inspect the online Matlab code that generates figure 11.5, you may notice that only the first half of the results of the Fourier transform are plotted. The first half of the Fourier series contains the “positive” frequencies, and the second half of the Fourier series contains the “negative” frequencies. The negative frequencies capture sine waves that travel in reverse order around the complex plane compared to those that travel forward (positive frequencies). This feature of the Fourier transform will become relevant when you learn about the Hilbert transform (section 14.1) but is not further discussed here. For a signal with only real numbers (as is the case with EEG data), the negative frequencies mirror the positive frequencies; you can thus ignore the negative frequencies and double the amplitudes of the positive frequencies. However, do not remove the negative frequencies from the Fourier series because you will need them to compute the inverse Fourier transform.

11.6 Inverse Fourier Transform

The Fourier transform allows you to represent a time series in the frequency domain. It is often stated that the Fourier transform is a perfect frequency-domain representation of a time-domain signal. If this is true, it must be possible to reconstruct the original time series data with only the frequency domain information. This is done through the inverse Fourier transform.

Once you understand how the Fourier transform is computed, the inverse Fourier transform is easy to understand and compute. Remember that the result of a Fourier transform is a series of coefficients that represent the mapping (the dot product) of each complex sine wave to the data. The inverse Fourier transform reverses this procedure. In fact, figure 11.2 was conceptually similar to an inverse Fourier transform: you started with sine waves of different frequencies, amplitudes, and phases, and you summed them together to form a single time series. Thus, to compute the inverse Fourier transform, you build sine waves of specific frequencies, multiply them by the respective Fourier coefficients at those frequencies, sum all of these sine waves together, and then divide by the number of sine waves (which is also the number of points in the time series). This will result in a perfect reconstruction of the original time series data. The online Matlab code shows you how to perform an inverse Fourier transform, and figure 11.6 shows the time series reconstructed from the Fourier coefficients and the original time series data.

For completeness, equation 11.3 shows the formula for the inverse Fourier transform, using the same format that was used in equation 11.2.

$$x_k = \sum_{k=1}^{n} X_k e^{i2\pi f(k-1)n^{-1}} \tag{11.3}$$

Notice that in equation 11.3, the dot product (here, a single multiplication) is computed between the complex sine wave and the Fourier coefficients *X* instead of the time series data *x*. The other difference between equations 11.2 and 11.3 is that the exponential contained a negative sine in equation 11.2, which is absent in equation 11.3.

This exercise shows that the original time series can be perfectly reconstructed based on the Fourier transform. This becomes important for learning about the convolution theorem and why you should perform convolution in the frequency domain.

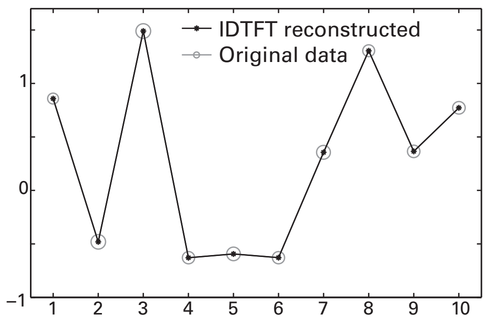

Figure 11.6
Time series data of randomly generated numbers and the reconstruction of the data using the inverse discrete time Fourier transform (IDTFT).

For the statistically oriented readers, there is also a statistical explanation of why the Fourier transform can perfectly represent a time series. If the time series is modeled using the same number of sine waves (analogous to regressors) as data points, there are no remaining degrees of freedom, and thus, the model must explain 100% of the variance of the data.

11.7 The Fast Fourier Transform

It is important to understand how the Fourier transform works at both the conceptual and the implementational levels. But, didactic purposes notwithstanding, you should never use the discrete-time Fourier transform as implemented in the online Matlab code for this chapter when analyzing your data. There is a much faster, more efficient, and more elegant way to compute the Fourier transform. It turns out that if you break up the Fourier transform into all of its individual multiplication and addition steps, many of these individual elements are redundant and can be eliminated, thus allowing the Fourier transform to be computed much faster and with no loss of information. There are several methods for increasing the efficiency of the Fourier transform, including the fast Fourier transform (FFT) that is implemented in Matlab using the function `fft`. The Matlab function for the inverse Fourier transform is called `ifft`. Figure 11.7 shows that the FFT and the "manual" discrete time Fourier transform produce identical results.

How much faster is the FFT compared to the discrete time Fourier transform? Figure 11.8 shows results of a computation time test. In this test the Fourier transform was computed on 640 time points of data using the discrete time Fourier transform and the FFT. The FFT was

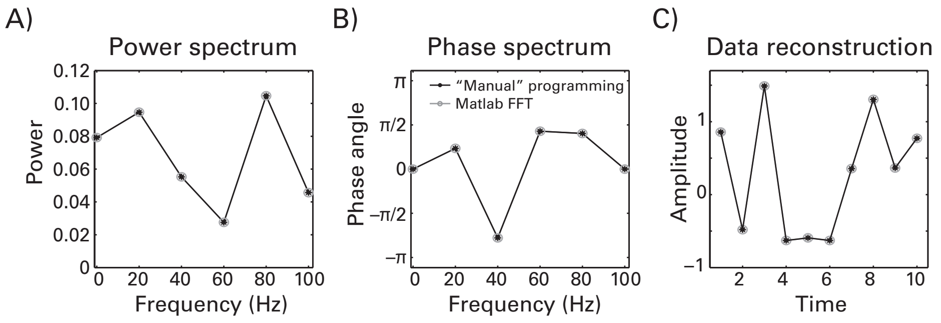

Figure 11.7
The fast Fourier transform (FFT) produces identical results as the discrete time Fourier transform and is much faster. (Note that due to very small computer rounding errors, you might see 0 or 2π, or $-\pi$ or $+\pi$, for the phase value at the Nyquist frequency.)

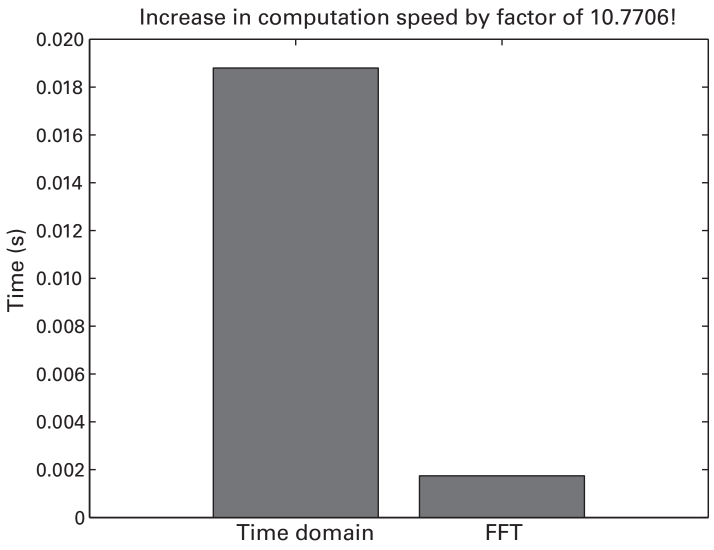

Figure 11.8
Computation times for the discrete time Fourier transform versus the FFT.

around ten times faster than the discrete time Fourier transform. In practice, though, this is a huge underestimate: the more data you have, the bigger the increase in speed. Furthermore, the speed of the FFT can be further decreased by using vectors with the number of elements corresponding to a power of two. In real data analyses, you may have millions of data points, and if the number of points is a power of two, the decrease in speed for the FFT compared to the discrete time Fourier transform can be around 3 orders of magnitude. This could be the difference between 1 min and 1000 min (17 h) of computation time. Technically, the speed of a discrete time Fourier transform is related to the number of computations as N^2, where N is the number of data points; the speed of the FFT is related to the number of computations as $N\log N$.

11.8 Stationarity and the Fourier Transform

One assumption of the Fourier transform is that your data are stationary, which means that the statistics of the data, including the mean, variance, and frequency structure, do not change over time (that is, the time series is "well behaved"). This is clearly not the case for EEG data: the frequency structure of neurophysiological activity changes over time both because of task events and because of endogenous processes. In this sense one assumption of the Fourier transform is likely to be violated in real data, particularly data collected during cognitive tasks. Although the lack of stationarity of EEG data reflects dynamic properties of the brain (Kaplan et al. 2005), it can also introduce difficulties for some analyses, including the Fourier transform and Granger prediction (chapter 28).

Violations of stationarity can decrease the peakiness of the results of the Fourier transform. This can be seen in figure 11.9. Two time series were created from the same sine waves of frequencies 3, 5, 7, and 10 Hz (figure 11.9A). One signal was stationary, and the other was nonstationary—variations were introduced by having the frequency structure change over time and also by having the amplitude and therefore also the variance increase over time within each sine wave. As seen in figure 11.9B, the power spectrum clearly shows the spectral peaks for both the stationary and nonstationary time series. However, the spectral peaks for the nonstationary data are less well defined, and there seems to be power at many other frequencies between the peaks, even though those frequencies were not explicitly defined in the simulated data. This is not an artifact but rather a feature: the nonstationary time series has a more complicated structure and therefore requires energy at a larger number of frequencies in order to represent the time series in the frequency domain.

This is one of two main reasons to perform temporally localized frequency decomposition methods such as wavelet convolution, filter-Hilbert, or short-time FFT. With temporally

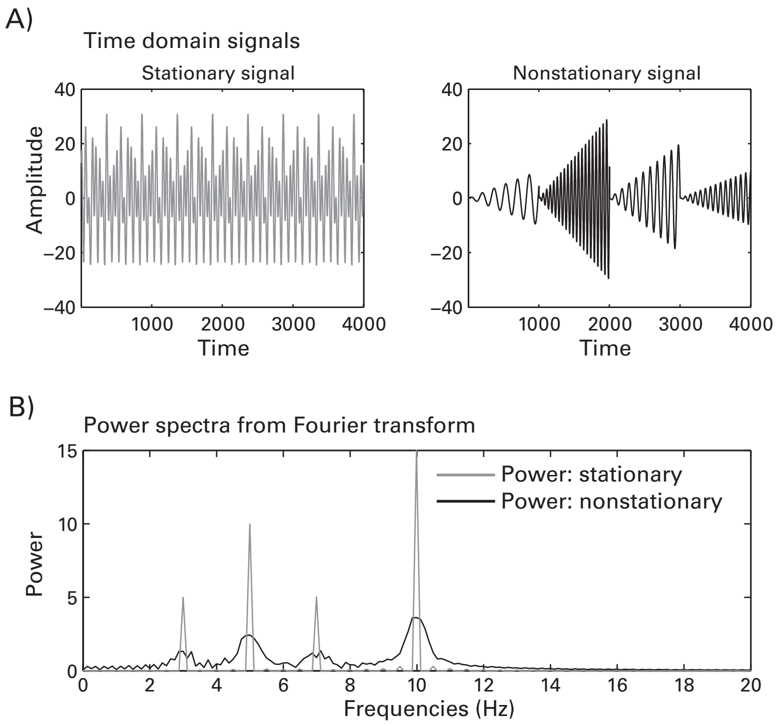

Figure 11.9
Violations of stationarity (here introduced by changes in frequency and amplitude over time) result in energy at frequencies that were not explicitly generated when creating the data. Nonetheless, spectral peaks can clearly be observed for both the stationary and the nonstationary time series. Note that for the stationary time series, the power peaks appear to be carrot shaped, but this is due to using lines instead of bars to represent discretely sampled frequencies. If you look closely, you can also see small but nonzero power at many frequencies for the stationary time series; these are artifacts resulting from the sharp edges at the beginning and ends of the time series.

localized methods, you assume that the data are stationary within relatively brief periods of time (generally a few hundred milliseconds for task data), which seems to be a reasonable assumption for EEG data (Florian and Pfurtscheller 1995).

The second reason to perform a temporally localized frequency decomposition was mentioned earlier: The Fourier transform does not show how dynamics change over time. That is, although the Fourier transform contains all temporal information of the time series data, as demonstrated by the ability of the inverse Fourier transform to reconstruct a time series, time-varying changes in the frequency structure cannot be observed directly in power or phase plots.

11.9 Extracting More or Fewer Frequencies than Data Points

Earlier you learned that the number of frequencies you get from a Fourier transform is $N/2 + 1$, where N is the number of time points in the data, and +1 is for the DC or zero-frequency component. Thus, the frequency resolution of the Fourier analysis is specified by the number of time points in the data. However, it is possible to get more or fewer frequencies from a Fourier transform (that is, it is possible to increase or decrease the frequency resolution). Because the number of frequencies is related to the number of data points, you can simply add data points to get more frequencies. However, because you may not have more real data to add, you can add zeros after the end of the time series. This is called zero padding. Zero padding increases the N of the Fourier transform without changing the data, thus allowing you to get more frequencies from the Fourier transform. However, because zero padding does not increase the amount of information in the data, it does not increase the frequency *precision* of the Fourier transform, only the frequency *resolution* (see section 2.8 for a discussion of resolution versus precision).

The Matlab function `fft` has an optional second input, which is the order of the FFT (N), which determines the number of frequencies that are extracted. By default, N is equal to the number of data points, but you can specify this number to be larger or smaller. If N is larger than the number of data points, the time series will be zero-padded, and the FFT will return more frequencies (Matlab adds the zeros; you do not need to do this yourself). Conversely, if N is smaller than the number of data points, data points will be removed from the end of the time series. Although zero padding does not provide any additional information, it can make frequency-domain convolution more convenient and faster to perform. Frequency-domain convolution is discussed in the next section, and the methods to utilize the power-of-two feature to decrease the speed of convolution are shown in chapter 13.

11.10 The Convolution Theorem

Learning how a Fourier transform works is important for several reasons, one of which is that the Fourier transform provides a means to perform convolution in an efficient and elegant manner. This is because of the convolution theorem, which states that convolution in the time domain is the same as multiplication in the frequency domain. (The opposite is also true, although it is less relevant for EEG data analyses: convolution in the frequency domain is the same as multiplication in the time domain.)

Because convolution in the time domain is the same as multiplication in the frequency domain, you can perform convolution in two ways. The first way is the time-domain version of convolution shown in the previous chapter: flip the kernel backward, slide it along the signal, and compute the dot product at each time step. The second way to perform convolution is by taking the Fourier transforms of the signal and the kernel, multiplying the Fourier transforms together point-by-point (that is, frequency-by-frequency), and then taking the inverse Fourier transform (figure 11.10). Although this may not seem as if it should make a difference, it does: time-domain convolution is slow, and frequency-domain convolution is fast.

When you perform the frequency-by-frequency multiplication of the Fourier transforms of the kernel and the signal, you are scaling the frequency spectrum of the signal by the frequency spectrum of the kernel. In other words, the result of the multiplication (and hence, the result of the convolution) is the frequency structure that is common to both the kernel and the signal. This is why convolution can be conceptualized as a frequency-domain filter, as was mentioned in section 10.2. Figure 11.11 shows an example illustrating how

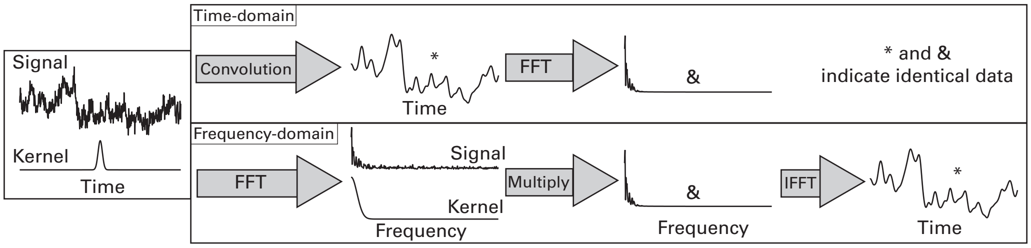

Figure 11.10
Illustration of the convolution theorem and the interchangeability of time-domain convolution and frequency-domain multiplication. The two time series with asterisks are identical, as are the two frequency spectra with ampersands.

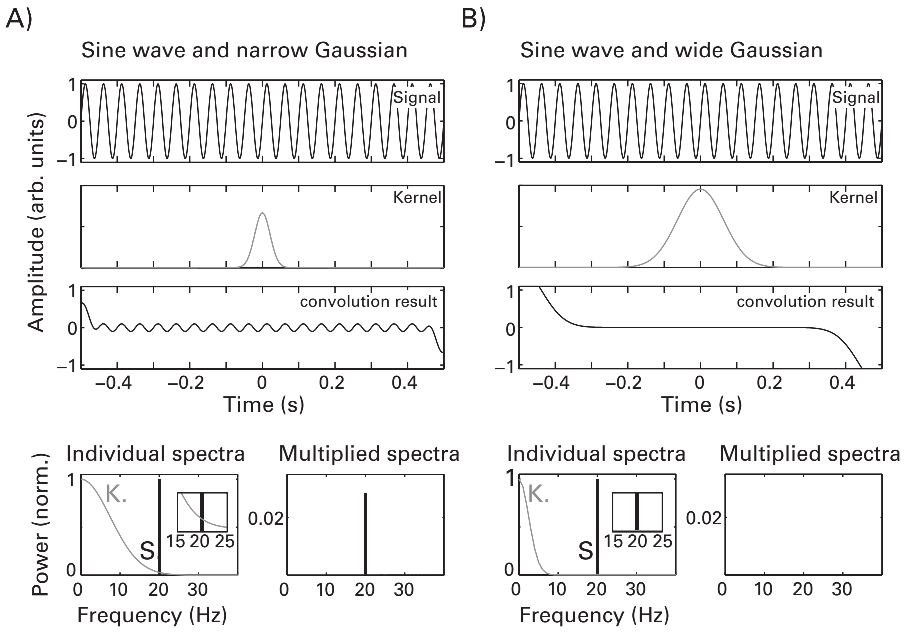

Figure 11.11
The convolution between a 20-Hz sine wave and a narrow Gaussian (panel A) dampens the sine wave, whereas the convolution between the same sine wave and a wide Gaussian (panel B) obliterates the sine wave. This is because the power spectrum of the sine wave (bottom row) overlaps slightly with the power spectrum of the narrow Gaussian at 20 Hz, but the power spectrum of the sine wave does not overlap with the power spectrum of the wide Gaussian. The gray line corresponds to the kernel (K), and the black bar corresponds to the signal (S). The insets in the power plots highlight the overlap (or lack thereof) between the frequency representations of the Gaussians and the frequency representation of the sine wave. The power spectra were normalized to 1 to facilitate visual comparison. The sharp rise and drop at the beginning and end of the result of convolution are edge artifacts.

convolution acts as a frequency filter, using simulated data. A sine wave of 20 Hz was convolved with two Gaussians—one with a narrow width and one with a wide width (the frequencies of the Gaussian widths were 15 Hz and 5 Hz; you'll learn more in the next chapter about specifying the widths of Gaussians). The result of the convolution of the sine wave and the narrow Gaussian simply dampened the amplitude of the sine wave, whereas the result of the convolution with the wide Gaussian obliterated the sine wave.

To understand why this is the case, consider the frequency spectra of the sine wave and the two Gaussians, and consider the overlap of their spectra. The sine wave has a very narrow spectral peak at 20 Hz, and the Gaussians have a smoother slope-looking shape. What you should notice is that the power spectrum of the narrow Gaussian has nonzero values that overlap with the nonzero values of the power spectrum of the 20-Hz sine wave (see small inset plots in the bottom row of figure 11.11). In contrast, the power spectrum of the wide Gaussian is zero at frequencies where the power spectrum of the sine wave is nonzero. The resulting frequency spectrum multiplications will produce an amplitude-attenuated sine wave for the narrow Gaussian and no sine wave at all for the wide Gaussian.

Thus, convolution acts as a filter such that the frequency profile of the signal is passed through the frequency profile of the kernel. As you will see in the next several chapters, this is the basis of wavelet convolution: you pass the EEG data through a set of filters (wavelets) that are tuned for specific frequencies, and the result of convolution is the frequency-band intersection between the EEG data and the wavelet.

One more example is given of how convolution acts as a frequency-band filter, this time using real EEG data. Figure 11.12 shows one trial of EEG data, a Gaussian, and the result of convolution between them. This figure is laid out in the same way as figure 11.11 is. Thus, in

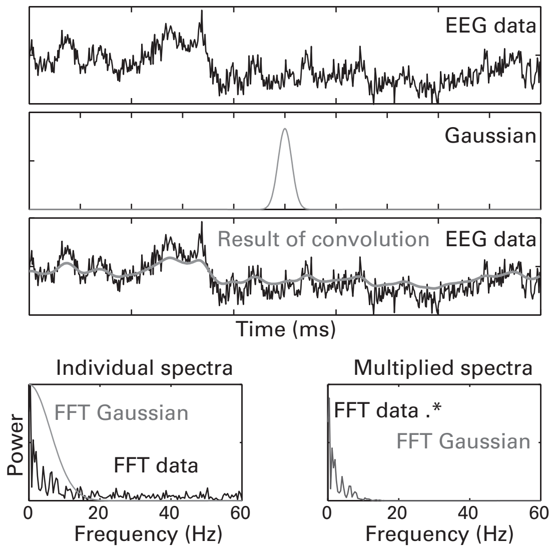

Figure 11.12
Convolving an EEG time series from one trial with a Gaussian low-pass-filters the data. This results from the frequency spectrum of the Gaussian tapering the higher frequencies in the EEG data. Note that this example here is meant for illustration of the convolution theorem and how convolution acts as a frequency filter; convolution with a Gaussian is not necessarily the best method for filtering EEG data. Chapter 14 contains more in-depth discussions of how to construct and apply bandpass filters to EEG data. The *y*-axis scaling of the power spectra is arbitrary to improve visibility.

the bottom row of figure 11.12 you can see the frequency characteristics of the data and the Gaussian (left-side plot), and you can see the result of multiplying their spectra frequency-by-frequency (right-side plot). The result of convolving the EEG data with a Gaussian is a time series comprising the frequency characteristics of the EEG data weighted by the frequency characteristics of the Gaussian. Thus, the Gaussian kernel is a low-pass filter because its spectral characteristics are dominated by low frequencies, and thus, the high frequencies of the EEG data are severely attenuated.

11.11 Tips for Performing FFT-Based Convolution in Matlab

If you try to perform FFT-based convolution by taking the inverse Fourier transform of the products of the Fourier transforms of the EEG data and the wavelet (or any signal and kernel), you will get an incorrect result. In fact, you may get a Matlab error if the signal and the kernel are not the same length. Even if the signal and the kernel are the same length, the following Matlab command will not produce a valid convolution:

```
result = ifft(fft(signal) .* fft(kernel));
```

The reason the above line of code will not produce a valid convolution is that the result of convolution must be equal to the length of the signal plus the length of the kernel minus one (the reason for this is discussed in section 10.3). Thus, you will need to make the inverse FFT return the correct number of time points, and to do this, you will need to make sure to compute the FFTs of the signal and the kernel using the appropriate number of time points (in other words, the length of the signal plus the length of the kernel minus one). Remember that the result of the FFT can be greater than the number of time points in the input if the `fft` function is called with a second argument that specifies the desired number of frequency points. After you compute the inverse Fourier transform, you will then need to remove the appropriate number of time points from the beginning and from the end of the time series. This is the correct way to perform a convolution using the `fft` and `ifft` functions.

This is a tricky point, and it is very important. If you do not use the `fft` function appropriately, you will get a result (if the signal and the kernel have the same number of time points), but that result will not be a convolution! You can easily double-check your code to make sure you've done it correctly by using the Matlab `conv` function:

```
result = conv(signal,kernel,'same');
```

This result should be identical to the result of the FFT-based convolution. If you use the `'full'` option instead of the `'same'` option, the result will the length of the result of convolution, that is, the length of the signal plus the length of the kernel minus one.

In theory, you can always use the `conv` function instead of performing convolution using `fft` and `ifft`. However, using the `conv` function is inefficient for wavelet convolution with many frequencies because you will redundantly recompute the FFT many times. This can slow data analysis times by an order of magnitude or more. This is explained more toward the end of chapter 13.

11.12 Exercises

1. Reproduce the top three panels of figure 11.12 three times. First, perform time-domain convolution using the "manual" convolution method shown in chapter 10. Second, perform frequency-domain convolution using the discrete time Fourier transform presented at the beginning of this chapter. Finally, perform frequency-domain convolution using the Matlab functions `fft` and `ifft` (do not use the function `conv`). (You can optionally reproduce the bottom panel of figure 11.12 for the frequency domain analyses; keep in mind that the power scaling is for display purposes only.)

2. From the three sets of Matlab code you have for reproducing figure 11.12, run a computation time test. That is, time how long it takes Matlab to perform 1000 repetitions of each of the three methods for computing convolution that you generated in the previous exercise (do not plot the results each time). You can use the Matlab function pairs `tic` and `toc` to time a Matlab process. Plot the results in a bar plot, similar to figure 11.8.

3. Generate a time series by creating and summing sine waves, as in figure 11.2B. Use between two and four sine waves, so that the individual sine waves are still somewhat visible in the sum. Perform a Fourier analysis (you can use the `fft` function) on the resulting time series and plot the power structure. Confirm that your code is correct by comparing the frequencies with nonzero power to the frequencies of the sine waves that you generated. Now try adding random noise to the signal before computing the Fourier transform. First, add a small amount of noise so that the sine waves are still visually recognizable. Next, add a large amount of noise so that the sine waves are no longer visually recognizable in the time domain data. Perform a Fourier analysis on the two noisy signals and plot the results. What is the effect of a small and a large amount of noise in the power spectrum? Are the sine waves with noise easier to detect in the time domain or in the frequency domain, or is it equally easy/difficult to detect a sine wave in the presence of noise?