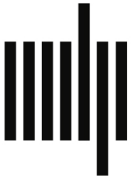
Analyzing Neural Time Series Data: Theory and Practice

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

10 The Dot Product and Convolution

In this section of the book (chapters 10–21), you will learn several techniques for extracting time-varying frequency-band-specific information from EEG data. Many techniques for extracting frequency information from a time-domain signal rely on a mathematical procedure called *convolution*, which in turn relies on a procedure called the *dot product*. The better you understand how convolution and the dot product work, the better you will understand how time-frequency-based analyses work, and the more flexibly you will be able to adapt and program time-frequency-based analyses.

10.1 Dot Product

Before learning about convolution, you need to become familiar with the basic step of convolution, which is called the dot product. There are several interpretations of the dot product. It can be thought of as a sum of elements in one vector weighted by elements of another vector (a signal-processing interpretation) or as a covariance or similarity between two vectors (a statistical interpretation) or as a mapping between vectors (the product of the magnitudes of the two vectors scaled by the cosine of the angle between them; a geometric interpretation). In different situations, you might find different interpretations to be more useful. But the algorithm is always the same, and the result is always the same: a single number that is computed based on two vectors of equal length.

To compute the dot product, you simply multiply each element in one vector by the corresponding element in the other vector (that is, the first element in vector A by the first element in vector B; the second element in vector A by the second element in vector B, and so on), and then sum all of the points.

$$dotproduct_{ab} = \sum_{i=1}^{n} a_i b_i \tag{10.1}$$

a and *b* are vectors and *n* is the number of elements in *a* (or *b;* the two vectors must have the same length). Translated into Matlab code, equation 10.1 can be expressed as a loop and then a sum, or it can be expressed as a single matrix multiplication. The online Matlab code shows you several different but equivalent methods to compute the dot product. The dot product may seem like a very simple and trivial expression, but do not underestimate its utility: the dot product is the basic building block of many data analysis techniques, including convolution and the Fourier transform.

The geometric interpretation of a dot product is particularly useful for time-frequency decomposition because this interpretation will facilitate understanding how to extract power and phase angles from complex numbers. Geometrically, the dot product is a mapping of one vector onto another. This is easiest to illustrate visually using two two-element vectors because two-element vectors can be conceptualized on a two-dimensional Cartesian plane. First, consider that the vectors [a1 b1] and [a2 b2] can be expressed as lines from the origin of the plot to the points described by [a1 b1] and [a2 b2] (figure 10.1). Now imagine drawing a line from the end of one vector to the other vector such that they intersect with a 90° angle. The length of the projection onto the vector is the dot product. Figure 10.1 illustrates situations in which the dot product can be greater than zero, equal to zero, and less than zero. Note that the dot product is zero when the vectors are orthogonal to each other (that is, when they intersect with a 90° angle).

A vector with two elements can be conceptualized as a point in a 2-D space. A vector with three elements can therefore be conceptualized as a point in a 3-D space (a cube), and so on for as many dimensions as there are elements in the vector. However, the principle for computing the dot product is the same as illustrated in figure 10.1, regardless of how many

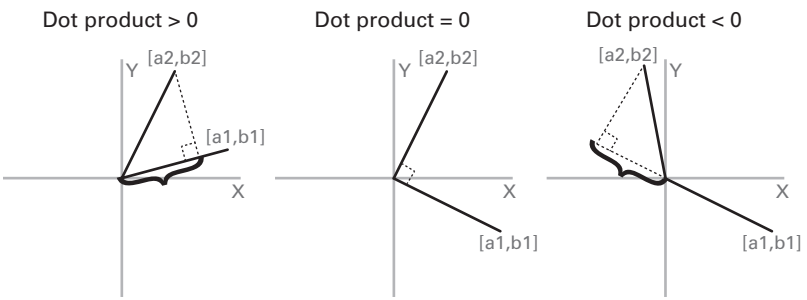


Figure 10.1
Graphical illustration of the geometric interpretation of the dot product between two two-element vectors. Curly brackets illustrate the magnitude of the projection of one vector onto the other (this is the dot product).

dimensions there are. Thus, an *N*-element vector can be conceptualized as a single point in an *N*-dimensional space, and the dot product between two *N*-element vectors is the mapping between two hyperlines from the origin of the *N*-dimensional space to the points defined by the elements (or coordinates) of those vectors. Thus, you can conceptualize an EEG signal that is 640 time points long as a single point in a 640-dimensional space, and the location of that point is defined by the values (coordinates) of the EEG signal at each time point. As you will learn over the next several chapters, several frequency and time-frequency decomposition methods involve computing the mapping between two high-dimensional vectors, such as a sine wave at a particular frequency and an EEG time series.

10.2 Convolution

Convolution in the time domain (convolution in the frequency domain is discussed in the next chapter) is an extension of the dot product, in which the dot product is computed repeatedly over time. Convolution can also be performed over space, but for EEG analyses, convolution over time is used.

Similar to the dot product, convolution can have several interpretations: you can think of convolution as a time series of one signal weighted by another signal that slides along the first signal (a signal-processing interpretation) or as a cross-covariance (the similarity between two vectors over time; a statistical interpretation, but see section 10.4 concerning convolution vs. cross-covariance), as a time series of mappings between two vectors (a geometric interpretation), or as a frequency filter (more on this in chapter 11). Again, different interpretations might be helpful in different situations, but the algorithm is the same: convolution is performed by computing the dot product between two vectors, shifting one vector in time relative to the other vector, computing the dot product again, and so on.

In convolution, one vector is considered the signal, and the other vector is considered the kernel. In practice it is possible to get the same result no matter which vector is labeled signal and which is labeled kernel. Thus, the distinction is mainly one of convenience. In this book the EEG data is called the signal, and the wavelet or sine wave is called the kernel.

10.3 How Does Convolution Work?

To introduce you to convolution, it is useful to consider a simple example. A square wave will be the signal (figure 10.2A), and a five-point linear decay function will be the kernel (figure 10.2B). Imagine taking the kernel, flipping it backward, and then dragging it forward in time so it passes over the signal. As the kernel passes over the signal, it drags on the signal, and the

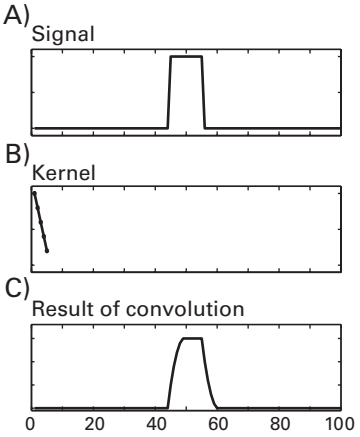


Figure 10.2
Example of convolution between two vectors, one labeled the signal (panel A) and one labeled the kernel (panel B). Panel C shows the result of the convolution between these two vectors.

result of the drag is something that is neither the signal nor the kernel, but looks a bit like each of them. Thus, conceptually, convolution entails sliding the kernel along the data and obtaining a result that shows what features the signal and the kernel have in common. This "commonality" at each time point is computed as the mapping between the kernel and the signal, that is, the dot product.

Hopefully figure 10.2 gave you an intuitive feel for what would happen when you drag the kernel along the signal. Figure 10.3 illustrates how convolution works, step by step. First, flip the kernel backward and line it up with the data as in figure 10.3A (the alignment actually starts before the signal for reasons that will be explained in a few paragraphs). Then, compute the dot product by multiplying each point in the kernel by each corresponding point in the data and summing over all of the multiplications. The dot product between the kernel and the corresponding part of the signal is then placed in a new vector in the position corresponding to the center of the kernel (figure 10.3A). It is convenient but not necessary to use kernels that have an odd number of data points, so that there is a center point.

This is the first step of convolution. Now the kernel is shifted to the right by one time step, but the signal does not move. Repeat the dot product procedure and store this result, again in the position corresponding to the center of the kernel, which is now one time step to the right. Notice that the dot product is computed with the same kernel but a slightly different part of the signal. This process of computing the dot product and then moving the kernel one step to the right continues until the kernel has reached the end of the data (figure 10.3B).

The resulting vector is a time series of dot products between the kernel and temporally corresponding equal-length windows of the signal.

There is one more thing to explain about convolution. You may have noticed that if you line up the kernel and the data such that they have their leftmost points overlapping and continue computing dot products until the kernel and the signal have their rightmost points overlapping, the result of the convolution contains fewer points than the signal (figure 10.3B). In this example with a short kernel, this is not a big loss, but imagine if the kernel and the data were equally long—the result of convolution would be one single dot product. This is not good. Therefore, convolution actually begins with the kernel aligned to the left such that the rightmost point of the kernel (after it is reversed) overlaps with the leftmost point of the data (10.3C). Because the two vectors must have the same number of points in order to compute a dot product, zeros are added before the start of the signal. The number of zeros added is the length of the kernel minus one (because the rightmost point of the kernel overlaps with the first point of the signal). These extra zeros do not contribute to the result because zero times a kernel is zero. Then, convolution runs each step toward the right, and it stops when the leftmost point of the kernel is aligned with the rightmost point of the signal. Again, zeros are added after the end of the signal to be able to perform a dot product.

As you can see in figure 10.3C, this means that the result of the convolution between the kernel and the signal is longer than the signal. How much longer? The result of convolution is one-half of the length of the kernel too long in the beginning and one-half of the length of the kernel too long at the end. The formula for how long the result of the convolution will be is length(signal) + length(kernel) – 1. Thus, after you run convolution, you need to trim the result by removing one-half of the length of the kernel from the beginning and one-half of the length of the kernel plus one from the end. This leaves you with a result of convolution equal to the length of the signal.

When written in equations, convolution is often denoted using the asterisk (*), as in: the convolution between a and b is $a*b$. This can get confusing because in many programming languages, including Matlab, the asterisk indicates multiplication.

The Matlab code online takes you through time-domain convolution step by step. You should take your time going through this code until you understand how time-domain convolution works. It may seem tedious to go through this code and perform convolution in the time domain when you could simply use the Matlab function `conv`. However, as mentioned at the beginning of this chapter, convolution is the basic step of many time-frequency decomposition methods, and so it is important to understand how it works.

Figure 10.4 shows that the result of this "manual" convolution is the same as the result of using the Matlab `conv` function, which performs convolution in the frequency domain

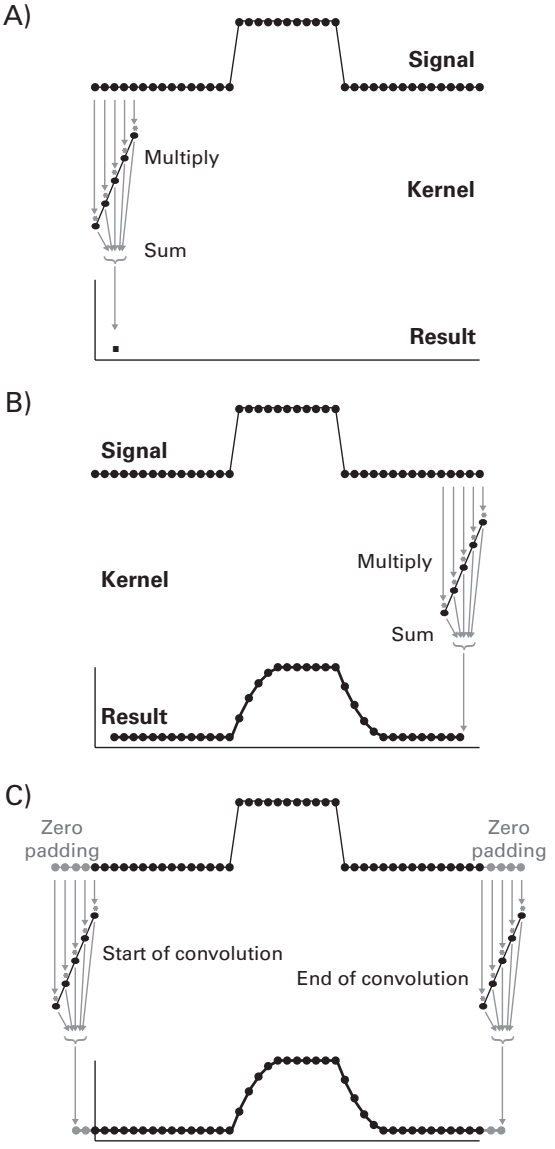

A)
Signal
Multiply
Kernel
Sum
Result
B)
Signal
Kernel
Multiply
Sum
Result
C)
Zero padding
Zero padding
Start of convolution
End of convolution

Figure 10.3
Overview of convolution. Panel A shows one step of convolution in which each point in the kernel is multiplied by each corresponding point in the signal, the multiplications are summed, and that value—the dot product—is placed in a position corresponding to the center of the kernel. Note that the kernel is reversed (compare with figure 10.2B). The kernel is then moved one time point to the right, and the dot product is computed again with a slightly different part of the signal. Panel B shows the result after many steps of convolution. Note that by lining up the leftmost point of the kernel with the leftmost point of the data at the start of convolution (and the rightmost point of the kernel with the rightmost point of the data at the end of convolution), the result of convolution is shorter than the signal. For this reason convolution actually begins with the rightmost point of the flipped kernel aligned to the leftmost point of the data, and convolution ends when the leftmost point of the kernel is aligned with the rightmost point of the data, as seen in panel C. The data are zero-padded on either end to compute the dot product with an equal number of points. The result of convolution therefore contains one-half of the length of the kernel too many points at the beginning and at the end. These extra points are discarded, leaving the result of convolution with as many points as the signal.

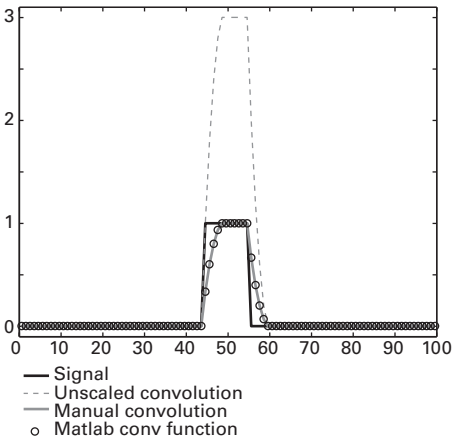


Figure 10.4
Overlap of results of manual time-domain convolution and the Matlab frequency-domain convolution function `conv`. Also shown is the result of convolution after normalization, which is not necessary when the sum of the kernel points is zero, as is the case with wavelets.

rather than the time domain (more on how this works later). To compare the result of convolution with the original signal, it is useful to scale the result of the convolution by the sum of the kernel points. Keep in mind that postconvolution scaling is not the same thing as mean-centering the kernel before convolution—the mean-centered kernel will produce a different convolution result because it will have negative numbers.

For completeness, equation 10.2 shows the mathematical formula for convolution.

$$(a * b)_k = \sum_{i=1}^{n} a_i b_{k-i} \tag{10.2}$$

In this equation, a and b are the two vectors for which the dot product is computed, k indicates that this equation refers to the convolution at time point k, and i corresponds to the elements in the equal-length parts of vectors a and b. Vector b is the kernel because it is flipped backward (this is the effect of $k - i$). You can see clearly from equation 10.2 that each step of the convolution is simply the dot product between the vector a and the time-reversed vector b.

10.4 Convolution versus Cross-Covariance

Convolution and cross-covariance (or cross-correlation; the cross-correlation is simply the cross-covariance scaled by the variances) are similar in concept but slightly different in algorithm. Both are methods to compute the time-varying mapping between vectors, but with convolution the kernel is reversed, whereas with cross-covariance the kernel is kept in its original orientation. If the convolution kernel is temporally symmetric—and thus is the same whether flipped backward or not—convolution and cross-covariance yield identical results.

10.5 The Purpose of Convolution for EEG Data Analyses

In EEG data analyses, convolution is used to isolate frequency-band-specific activity and to localize that frequency-band-specific activity in time. This is done by convolving wavelets—time-limited sine waves—with EEG data. As the wavelet (the convolution kernel) is dragged along the EEG data (the convolution signal), it reveals when and to what extent the EEG data contain features that look like the wavelet. When convolution is repeated on the same EEG data using wavelets of different frequencies, a time-frequency representation can be formed. Before learning about how this works and how to implement it yourself in Matlab, you will need first to learn about the Fourier transform and the convolution theorem. These are the topics of the next chapter.

10.6 Exercises

1. Create two kernels for convolution: one that looks like an inverted U and one that looks like a decay function. There is no need to be too sophisticated in generating, for example, a Gaussian and an exponential; numerical approximations are fine.

2. Convolve these two kernels with 50 time points of EEG data from one electrode. Make a plot showing the kernels, the EEG data, and the result of the convolution between the data and each kernel. Use time-domain convolution as explained in this chapter and as illustrated in the online Matlab code. Based on visual inspection, what is the effect of convolving the EEG data with these two kernels?