Fundamentals of Time-Frequency Analyses in Matlab/Octave

Mike X. Cohen

Publisher: sinc(x) Press
Kindle edition
Publishing Date: 2014

Table of Contents

in this book

Chapter 1: Introduction

1.1 | Time, time series, signals, and time-frequency analyses

Time is an immutable force that governs nearly every aspect of our lives, from perceiving the world around us to scheduling work meetings to the inevitable process of aging. Most events are defined at least partly by their changes over time or durations in time. Thus, characterizing, quantifying, and interpreting how events in the world change over time is fundamental to many aspects of science and engineering.

This book concerns the analysis of time series data. "Time series data" simply means any measurement that changes in amplitude over time. Examples include music, speech, EEG (electroencephalogram; brain electrical activity), radio waves, bat echolocations, bird calls, etc.

The terms "time series" and "signal" are often used inter-changeably. In this book, a clear distinction will be made. A "time series" is simply a collection of numbers that is sampled at regular intervals (e.g., once each millisecond, or one thousand times per second) and that changes over time. The "signal," on the other hand, is a component of the time series that contains the information that should be isolated. That is, a time series can contain only signal, it can contain signal plus noise (and thus, one of the goals of the analysis is to separate the signal from the noise), or it can contain only noise. In the simulated time series data in this book, the signal and noise are created independently and then combined to arrive at the time series. In real (empirically obtained) time series data, it is often more difficult to separate the signal from the noise.

Many time series analyses involve representing the time series as a combination of sine waves or sine-wave-like templates. This is called a Fourier-based or frequency-based approach. In many time series, the characteristics change over time; for such data, a *time-frequency approach* is appropriate.

1.2 | This book

This book takes a practical and implementation-based approach to learning time-frequency analyses. Mathematical proofs and equations are avoided when possible, and theoretical discussions involving continuous signals are kept to a minimum. Instead, the book focuses on how to implement analysis algorithms in programming code. The idea is to gain an intuitive understanding of time-frequency-based analyses of discrete, sampled time series data, by implementing those analyses in computer code. In other words, learning by doing.

There are many analyses that can be applied to time series data, and this book does not cover all of them. Indeed, even the 770-page tome *Time-Frequency Analysis* by Boualem Boashash (2003) cannot cover all of the possible time-frequency-based analyses, parameters, and implementation details. However, by going through this book you will be well prepared to learn and implement methods that are not covered here. Indeed, the Fourier transform and its inverse, convolution, wavelets, and stationarity are fundamental

concepts that form the foundation of nearly all time-frequency-based data analysis methods.

This book is not a manual for a toolbox. Rather than describing functions that you can simply call to produce a result and a figure, the focus is on understanding the analysis methods and data transformations. You can use this book to design custom-written code to perform time-frequency analyses, or you can use this book to help understand what is happening "under the hood" when you use functions from other toolboxes or point-and-click options in graphical-user-based software packages.

This book is also not an "idiot's guide" to time-frequency analysis. Understanding the concepts and programming implementations in this book may require some effort, particularly if you do not have a strong background in math or programming. That said, the book is designed for new-comers to time series analysis, and starts with basic foundations before building up to more complex analyses. Almost anyone with some patience and perseverance should be able to understand all of the material in this book and accurately complete all of the exercises at the end of the chapters.

Readers interested in deeper mathematical explanations of the algorithms and how to extend them can consider the Boashash 2003 book mentioned above, and the book *Time Frequency Analysis: Theory and Applications* (Prentice Hall) by Leon Cohen (unrelated to the author of this book). Readers interested in the use of time-frequency analyses for brain electrical time series data can consider the book *Analyzing Neural Time Series: Theory and Practice* (MIT Press, 2014).

1.3 | eBook

Why publish this book as an eBook? Many excellent academic books are expensive, sometimes prohibitively so, usually costing over $100 and occasionally up to nearly $400. An electronic-only book helps keep costs down so it can be affordable and easily purchased from anywhere in the world with stable electricity and internet access. Traditional books are published with paper and ink—expensive and limited resources, and not without environmental impact. This eBook can be viewed on any computer, Amazon Kindle, ipad, smartphone, or any other e-reading device that supports a Kindle reading app.

On the other hand, formatting an eBook is a challenge for a book like this, precisely because there are no traditional pages. Page widths and formats depend on the reading device, different reading devices have different fonts installed, many devices do not have color, and the user (that is, you) can select the font size. This necessarily means that some visual aesthetics must be sacrificed. Thus, the visual formatting of this book may be less polished than that of a traditional book, but this must be balanced against the increased flexibility and decreased publication costs.

For the same reason, the figures in this book are intentionally kept at a somewhat low resolution, and in grayscale. This is done to decrease file size and thus keep both download/loading times and costs down, and also to ensure viewability on non-color reading devices.

However, as explained below, nearly all of the figures in this book can be reproduced on your computer. This will generate high resolution color images that can be inspected much more carefully and in much greater depth than even a paper book with high-quality glossy pages.

1.4 | Intended audience and background knowledge

This book is intended for individuals who want to learn about frequency and time-frequency analyses of time series data. It may be most helpful for students taking courses in signal processing, computer science, and engineering, but will also be useful for anyone interested in learning about how to characterize and interpret time-varying multidimensional changes in the complex signals that surround them.

There is no strict prerequisite knowledge that is necessary for this book. Some background in signal processing or mathematics will be useful but not necessary. Some familiarity with basic trigonometry (e.g., the Pythagorean theorem, calculating angles on a right-side triangle, plotting a complex number on the complex plane) will also be useful.

Programming is perhaps the most important background knowledge. Although the code in this book is not very advanced, understanding much of the book relies on being able to look at and interpret basic Matlab code. Readers with no programming experience may benefit from first going through some Matlab introductory programming resources.

Chapter 2 contains an introduction to Matlab programming that focuses on explaining the code used throughout this book, and Chapter 13 provides an alphabetical list of all Matlab functions used in this book, including explanations of inputs and outputs, and usage tips.

1.5 | Matlab and Octave

As mentioned above, the purpose of this book is to impart hands-on, practical knowledge about time-frequency analyses. Thus, the book contains computer code both to simulate and to analyze time series data. The best way to learn from this book is therefore to go through it in front of a computer with Matlab or Octave installed.

It is strongly recommended not only to look at the figures in the book, but to run the code on your computer and inspect the plots. You can change the analysis parameters to understand their impact on the results, use the same analysis on different data, and generally enhance your learning experience by exploring, using the code in the book as a starting point. It is by actively exploring, modifying, and inspecting code and figures—not by passively looking at pictures in a book—that you will develop a deep and flexible understanding of how to apply and interpret time-frequency analyses.

The code is written in the Matlab (www.mathworks.com) programming language. Matlab is widely used in engineering and sciences at many companies and in most universities. However, it is also expensive and not everyone has access to Matlab (though student licenses are available and are less expensive).

To make this book maximally accessible to those on a budget, the code in this book is tested to work in Octave (www.octave.com). Octave is a cross-platform program that is

free to install and that can interpret nearly all Matlab commands (as well as commands from some other programming languages). For the rest of the book, the term “Matlab” is used for convenience, but the code can also be run in Octave. In the few cases where Matlab and Octave differ, this is stated clearly. It is also possible to translate the code into any other language such as Python, C++, etc., but this may require some expertise from the reader.

Matlab code is printed in **`boldface Courier`** font. However, as mentioned above, due to reading-device-specific fonts and formatting, the code may appear on some readers only as **boldface**. Sections of code are separated from the text by horizontal lines. It is possible that a single line of code will wrap onto the next line; if this happens often, consider decreasing the font size on your reader. When one line of code is intentionally wrapped over multiple lines, the end-of-line break-point is delineated by an ellipsis (…). Below is an example of what the Matlab code in this book will look like, as well as a figure.

```
figure
plot(magic(20))
```

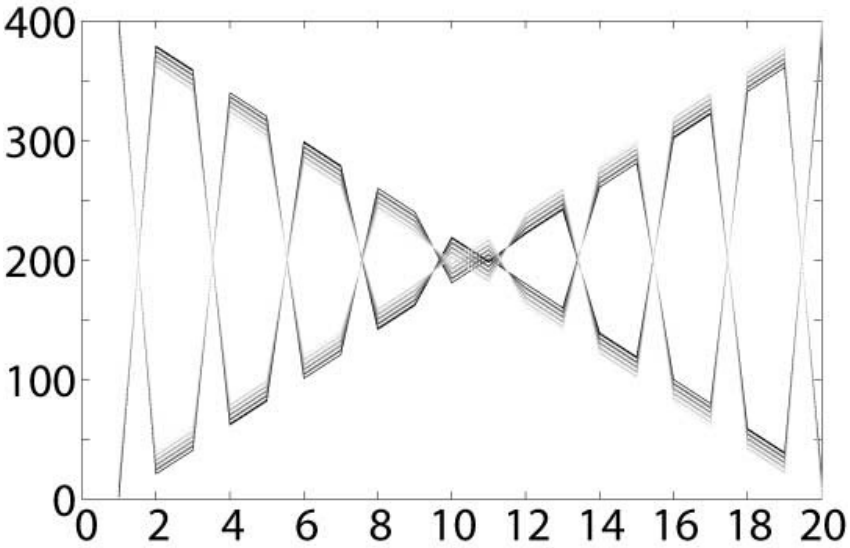

Figure 1.1 | Magic numbers

The code printed in the chapters provides the bare-bones essentials to understand the main concepts. All of the code in this book is also available online at sincxpress.com. The downloadable code contains more detailed comments and explanations, as well as additional code that is not critical but will help further explain the programming implementations and core concepts, and will produce additional figures and notes about parameters and their effects on the analyses.

If you type the code into your computer directly from the book text, you may need to open new figures or clear existing figures to prevent material from appearing on the same figure (use the Matlab command **`figure`** to open a new figure, and the command **`clf`** to clear a figure).

There is a custom Kindle dictionary (mobi-format) that is designed for this book, available at sincxpress.com. Once the dictionary is installed on your reading device, you can look

up a Matlab function to see an explanation of that function and usage tips. This dictionary is also printed in Chapter 13.

1.6 | Exercises

Each chapter ends with exercises that require applying knowledge learned from that chapter. In many cases, it will be necessary to integrate knowledge learned in previous chapters as well.

A full solution is provided only to the first of the exercises in each chapter. For the remaining exercises, figures of correct solutions are provided. If you have the correct solution, your figures will look like those in the book. Depending on selected parameters and, in some cases, random numbers, a correct solution may look similar but not identical to the figures in the book.

Chapter 2: Matlab Introduction

This chapter provides a basic introduction to Matlab programming that will help prepare you for the rest of the book. It is not intended as a full beginner's guide for programming in Matlab, but rather, as a guide to help the novice get started. Unless indicated otherwise, all functions and commands in this chapter will work the same in Matlab and in Octave.

To learn more about each function described below, type **`help <functionname>`** in the Matlab command. In some cases, **`doc <functionname>`** will produce additional usage guidelines. Chapter 13 contains an alphabetized list of all functions used in this book, including some usage tips. If you install the custom dictionary for this book (available from sincxpress.com), you can also get help on functions in this book in the same way you would look up the definition of a word.

2.1 | Variables and matrices

Variables are placeholders for data. Variables can store a variety of data types, including numerical and string (text).

```
varA = 4;
varB = 15;
varA*varB
```

Note that the first two lines ended with a semi-colon while the third line did not. Ending a line with a semi-colon indicates that the output should not be shown in the command terminal. Try running the previous lines without semi-colons.

It is generally useful to have variables named in sensible ways that can easily be interpreted.

```
myName = 'mike';
myAge = 35;
```

Variable names cannot have spaces or other special characters such as $%^. They may contain numbers but may not start with numbers. Numbers can help create useful variable names, such as **`frequency2plot`** or **`data4analysis`**.

In addition to storing individual items, variables can store collections of items, such as vectors and matrices.

```
a = [ 1 2 3; 4 5 6; 7 8 9 ];
b = [ 1 2 3 4 5 6 7 8 9];
```

The difference between **`a`** and **`b`** is the presence of semi-colons after the 3 and 6. Unlike the semi-colon at the end of a line, the semi-colon inside brackets indicates that the following numbers should form a new row. Thus, **`a`** is a 3x3 matrix while **`b`** is a 1x9

vector.

2.2 | Functions

Functions are encapsulated sections of code that perform some operation on input data. Functions are designed to make programming easier. For example, to compute the average of a set of numbers, the function **`mean`** can be used. Using this function is faster and more convenient than writing out the code to sum over all numbers and then divide by the number of numbers.

Most functions take input and give output. For example:

```
a = [1 3 5 3 2 4 6 3];
meanA = mean(a);
```

Some functions take multiple inputs. For example, the function **`linspace`** takes three inputs:

```
a = linspace(1,5,10)
```

This function returns a vector that contains ten linearly spaced numbers between 1 and 5.

Some functions can give multiple outputs. For example:

```
a = [1 3 2];
[maxVal,maxIdx] = max(a);
```

In this case, the function **`max`** returns both the value of the largest element in variable **`a`** and its position (in this case, 2, because the largest element is in the 2nd position of the array).

Functions are files that are stored on your computer. They have the extension .m but are just plain text files. They can be opened from your computer using a text editor or through Matlab using **`edit <functionname>`**. Some functions are compiled for speed and thus the function file itself contains only the help information.

It is possible to modify existing functions, but this is generally not a good idea. If you want to modify a function, it is best to re-save it using a different name.

Matlab can only know the existence of functions if (1) the function file is located in Matlab's current working directory (folder), or (2) if the function is in Matlab's path. To change directories, type **`cd <directoryname>`**, as you would in a unix terminal. To add a folder in which the function file is located to Matlab's path, use the **`addpath`** function.

```
addpath('/path/to/file/location') % unix
addpath('C:/Users/mxc/Desktop') % windows
addpath(genpath('/path/to/file/locations'))
```

The third line in the code above will put the folder "/path/to/file/locations/" in the Matlab path, and will also put into Matlab's path all subfolders, for example "/path/to/file/locations/new". In Matlab but not Octave, it is also possible to set the path by clicking on the "File" menu and selecting "Set Path."

2.3 | Comments

Comments are pieces of text that are not evaluated by Matlab. They are important because they allow you to keep track of the purpose of lines of code, to remember what variables mean or what data they store, and to leave notes so other people can interpret and work with your code. In Matlab, comments are indicated with percent signs.

```
a=3;
% this line is not evaluated
% d=3;
c=4; % comments after code can be useful
a*d
a*c
```

The multiplication **`a*d`** produces an error because variable **`d`** does not exist. It does not exist because it is created in a line that is commented and thus ignored by Matlab.

2.4 | Indexing vectors and matrices

Indexing is fairly intuitive. Given the vector [1 4 2 8 4 1 9 3] , one could say that the number in the 4th position is 8. In Matlab this is implemented as follows.

```
a = [1 4 2 8 4 1 9 3];
a(4)
```

The result of the second line is "8" because the 4th number is 8. Multiple indices can be extracted at the same time using square brackets:

```
a([4 5 6])
```

Caution should be taken when indexing. Consider the outputs of the following two statements, and, importantly, the ordering of the outputs.

```
a([1 3 5])
a([3 5 1])
```

The first line above returns the numbers **`1 2 4`** whereas the second line returns **`2 4 1`**.

To index specific parts of matrices, elements along all dimensions are listed.

```
a = rand(4,5);
a(3,2)
```

This returns the number corresponding to the 3rd row and 2nd column. No matter how many dimensions are in the matrix, the number of comma-separated indices should match the number of dimensions.

```
a = rand(2,5,3,2,6,8,4);
a(1,1,1,1,1,1,1)
```

In this case, variable **a** is a 7-dimensional matrix and the second line returns the value of the first element across all dimensions.

In fact, when indexing matrices, it is not necessary to use as many index dimensions as there are dimensions in the matrix. That is, the following is possible.

```
a = [1 2 6; 3 4 9];
a(1,3)
a(5)
```

The number "6" appears in the 1st row 3rd column of matrix **a**, hence **a(1,3)** is 6. However, **a(5)** also returns 6. This may initially seem strange because neither dimension contains more than three elements. However, Matlab also counts matrix elements using a linear indexing by first counting down columns and then to the right over rows. Thus, in a 2x3 matrix, location 1,3 is equivalent to location 5.

This method of linear indexing can be convenient when used correctly, but easily becomes confusing. It is a common source of programming errors, particularly for less experienced programmers. In practice, it is often best to use matrix indexing whenever possible.

2.5 | Sizes and properties of variables

A list of variables in the workspace along with their key properties can be obtained using the function **whos**.

The size of a specific variable can be found using the function **size**, for example, **size(a)**.

If the variable has multiple dimensions, the function **size** can return the number of elements only in a specified dimension by providing a second input. For example, if matrix **a** has three dimensions, **size(a)** will return a 3-element vector (each element corresponds to the length of each dimension), whereas **size(a,2)** will return only the length of the 2nd dimension.

A related function is **length(a)**, which returns the number of elements in the longest dimension. This is useful for vectors (1xN matrices) but can be confusing for matrices with several dimensions, because **length** returns only the number of elements along the longest dimension; it does not indicate from which dimension that number comes.

Another function that can be used to query the size of a matrix is **numel**. The function **numel** returns the total number of elements in the matrix. Consider the following differences and similarities:

```
a = rand(3,2,5,10);
size(a)
length(a)
numel(a)
prod(size(a))
```

To remove a variable from the workspace, type **clear <variablename>**. Typing **clear** without a variable name afterwards will clear all of the variables in the workspace.

2.6 | Counting

Counting can be done using colons. To count in integers (whole numbers) between two numbers, use one colon.

```
1:10
```

To count using increments other than 1, specify the counting interval using an additional colon.

```
1:2:10
1:1.94583:10
```

To count backwards, it is necessary to specify that the counting variable be negative.

```
10:1 % empty
10:-1:1 % works as expected
```

The colon operator can also be useful when indexing a matrix. The term **end** refers to the final element along the dimension.

```
a = 1:20;
a(1:5)
a(1:3:end)
```

2.7 | Loops and conditionals

Loops allow multiple lines of code to be repeatedly evaluated. Matlab “for-loops” start and end with **for** and **end**, and require specification of a looping variable.

```
for i=1:4
```

```
  disp([ 'iteration ' num2str(i) ])
end
```

"While loops" run endlessly until some criterion is met.

```
toggle = false;
i=1;
while toggle==false
  disp([ 'iteration ' num2str(i) ])
  if i==4, toggle=true; end
  i=i+1;
end
```

Notice that in the "for" loop it is not necessary to explicitly increment the counting variable **`i`** whereas in the "while" loop, explicit incrementing is necessary.

"If" statements are modeled after human decision-making processes, and are thus fairly intuitive to understand and to implement.

```
myAge = 52;
if myAge>25
  disp('I''m older than a 1/4 century.')
else
  disp('I''m younger than a 1/4 century.')
end

if myAge>25 && myAge<50
  disp('Still many years ahead of me.')
elseif myAge>49 && myAge<80
  disp('I''m a bit old.')
elseif myAge>79
  disp('I''m really old.')
else
  disp('Age is <25 or non-numeric.')
end
```

There are a few things to note in the previous code. First, "if" statements can be accompanied by "elseif" and "else" clauses. "else" clauses are recommended to catch circumstances in which none of the previous "if" questions is true. Second, combined tests can be done through && (|| is used to indicate "or"). Third, the **`<50`** and **`>49`** tests ensure that an exact age of 50 will not slip through. That is, if the first "elseif" statement had **`>50`** instead of **`>49`**, it would actually test for ages 51-79. If my age were exactly 50, the result of this "if" paragraph would be **`Age is <25 or non-numeric.`**

2.8 | Expanding matrices with repmat and bsxfun

Performing basic operations between two scalars (single numbers) is easy.

```
a=4; b=12;
```

```
a*b
```

Performing basic operations between one scalar and a vector or matrix is also easy.

```
b=randn(4,2);
b*a
b+a
```

Performing point-wise operations between two matrices or vectors can be done if they are the same size. Point-wise multiplication and division are distinguished from matrix multiplication and division by using the “dot” operators.

```
a=rand(4,2); b=rand(4,2);
a.*b
a.^b
a-b
```

However, point-wise operations between matrices of different sizes cannot be done in the same manner as above. Imagine you want to multiply a two-element vector by each row of another matrix.

```
a=rand(4,2); b=rand(1,2);
a.*b
```

The second line will produce an error, because the two matrices do not line up. There are two solutions to this problem. One is to use the function **`repmat`**, which stands for replicate (repeat) matrix. It takes 2 or 3 inputs: the matrix to replicate, and the number of times to replicate that matrix along each dimension.

```
repmat(b,1,1) % replicate once… no effect
repmat(b,1,2)
repmat(b,2,1)
a.*repmat(b,4,1) % this works!
```

The second solution to this problem is to use the function **`bsxfun`**, which will automatically figure out how to expand the matrices. With **`bsxfun`**, the first input argument is the operation that will be performed between the two matrices.

```
bsxfun(@times,a,b)
bsxfun(@minus,a,b)
```

2.9 | Basic line plotting

The most basic plotting function is **`plot`**, which plots data along the y-axis as a function of corresponding points along the x-axis.

```
x=1:10; y=x.^10;
plot(x,y)
```

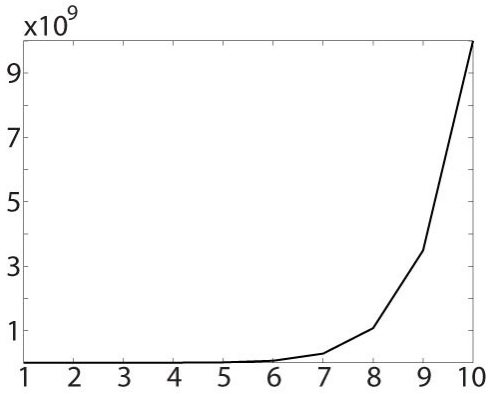

Figure 2.1 | Bigger faster

If the first input to **plot** is not provided, it is assumed to be integers from 1 to the number of elements in **y**.

The function **plot** will also accept 2-dimensional matrices, and will plot each row as a separate line.

```
y(:,1) = x.^1.1;
y(:,2) = exp(x/10);
plot(x,y)
```

Plotting multiple lines of data can also be done by typing **hold on**.

```
plot(x,y(:,1))
hold on
plot(x,y(:,2))
hold off
```

Because **hold off** was typed, the next plotting command will erase the existing plot before plotting the new data.

If you run the code above, the plot will contain two blue lines. Matlab has a default order of colors used for lines, and the order is reset each time the **plot** command is called. Typing **hold all** will override the color order reset.

The colors and the shapes at each discrete data point can be specified.

```
plot(x,y(:,1),'r-o')
hold on
plot(x,y(:,2),'m*','markersize',10)
hold off
```

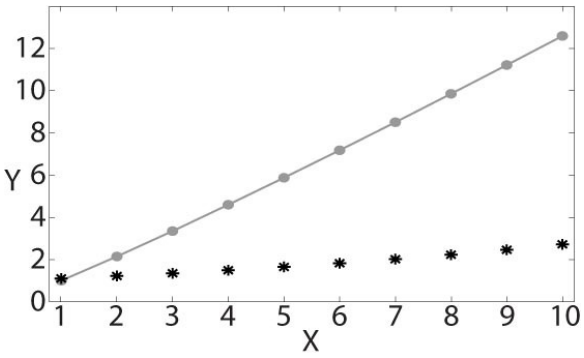

Figure 2.2 | Lines, circles, and stars

There are several other options for **`plot`** that can specify the line thickness, size of the points, etc. Type **`help plot`** for details.

Another commonly used plotting function is **`bar`**, which creates bar plots.

```
bar(x,y(:,1))
```

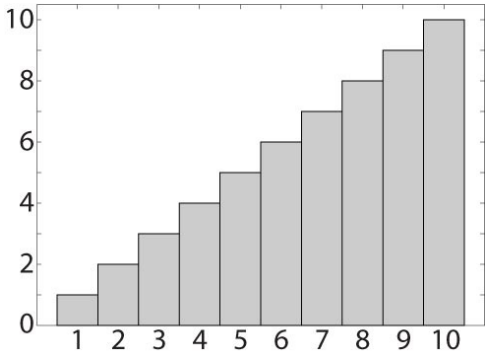

Figure 2.3 | Bars

2.10 | Get and set

Additional properties of a figure can be specified using the **`set`** function. **`set`** takes as a first input a pointer to a specific axis (part of a figure), and other inputs are specified in pairs.

```
plot(x,y)
set(gca,'xlim',[0 14])
```

This command accesses **`gca`**, which stands for “get current axis” and points to the axis most recently used, or most recently mouse-clicked. Next is the pair **`'xlim',[0 14]`**, which sets the x-axis limits to range from 0 to 14.

Multiple properties can be set in one command.

```
set(gca,'xlim',[.9 10.1],'ylim',[0 20])
```

It is possible to assign variables to refer to specific axes.

```
h = axes; % create new axis, called h
plot(h,x,y); % plot x,y in axis h
set(h,'ylim',[0 15])
```

The variable **h** may look like a normal number, but in fact is recognized by Matlab as a pointer to the specific axis in which **x,y** are plotted.

The **set** command can be used to change figure properties as well as axis properties.

```
set(gcf,'number','off','name','Figure name')
```

To obtain the current setting of axis properties, use the **get** function.

```
get(gca,'xlim')
get(gca,'yscale')
```

To see a list of axis parameters and their settings, type **get(gca)**.

2.11 | Figures and parts of figures

New figures can be created with the command **figure**. If there are no figures open, a plotting command such as **plot** or **bar** will create a new figure. Individual figures can be accessed using **figure(1)**, **figure(15)**, etc.

By default, each figure contains one axis for plotting. The axis is the white region in which the data are visually represented. It is possible to have more regions in the figure to plot data. This is useful, for example, when comparing results of different analyses on the same data. This is done with the function **subplot**, which takes three inputs: The number of rows, the number of columns, and which axis is the active one.

```
subplot(2,1,1), plot(rand(3))
subplot(2,1,2), plot(rand(10))
```

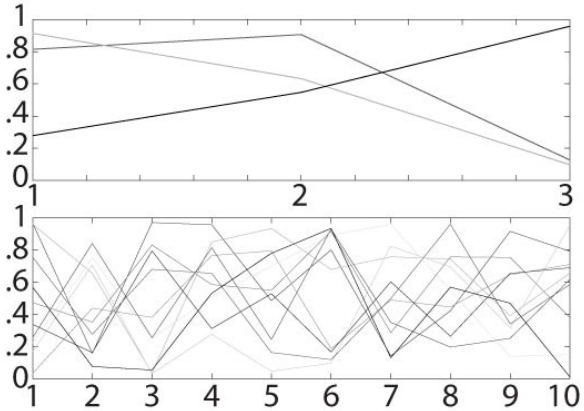

Figure 2.4 | Two subplots

It is possible to change the organization of the subplots within a figure.

```
subplot(2,1,1), plot(rand(3))
subplot(2,2,3), plot(rand(10))
subplot(2,2,4), plot(rand(20))
```

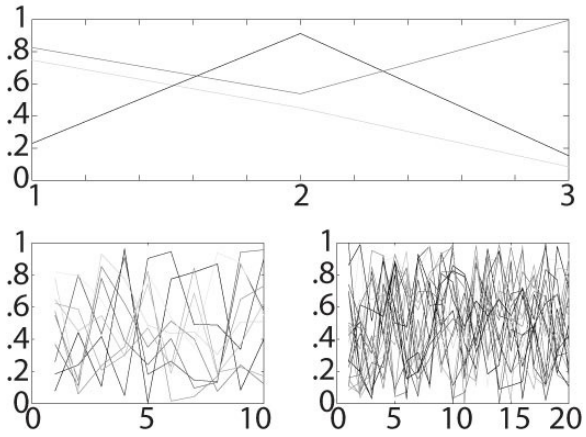

Figure 2.5 | Subplots of different sizes

If there are fewer than ten subplots, the commas are not necessary. That is, **`subplot(2,3,1)`** is equivalent to **`subplot(231)`**.

2.12 | Basic image plotting

Above, numbers were converted into points and lines connecting them. However, matrices can also be represented as images, by considering that the value at each point in a matrix can be colored according to the magnitude of the number. Consider the following.

```
x=[1 2; 3 4];
imagesc(x), colorbar
```

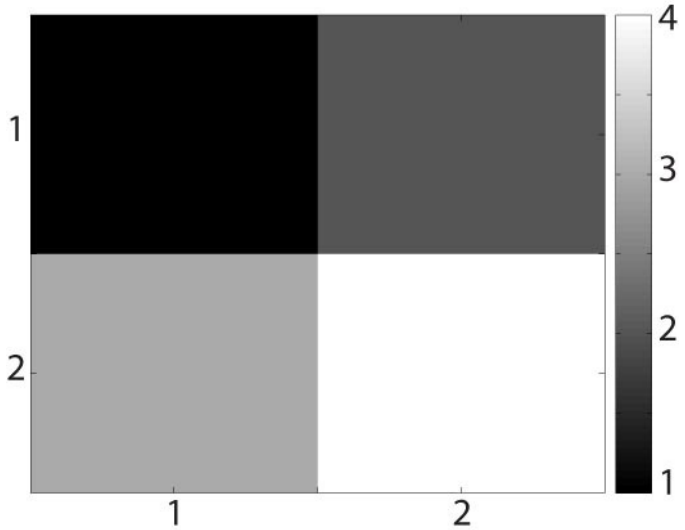

Figure 2.6 | Image of four numbers

The command **`imagesc`** is the same as the function **`image`**, except **`imagesc`** automatically scales the colors to be in the range of the data. The default color map sets the smallest value to dark blue and the largest value to deep red. There are several color maps pre-installed in Matlab, and custom color maps can be created. Type **`help colormap`** for details.

By default, the function **imagesc** shows the data how it appears in the Matlab command, with the first row on the top and the last row on the bottom. In some situations, it is useful to have the plot the other way around. This can be adjusted using **set(gca,'ydir','normal')** or **axis xy** (the default settings are **set(gca,'ydir','reverse')** or **axis ij**).

Smoother plots can be made using the **contourf** function, which is related to the **contour** function except the spaces between contours are filled.

```
w = linspace(0,1.5,300);
x = bsxfun(@times,sin(2*pi*w),sin(w)');
contourf(w,w,x)
```

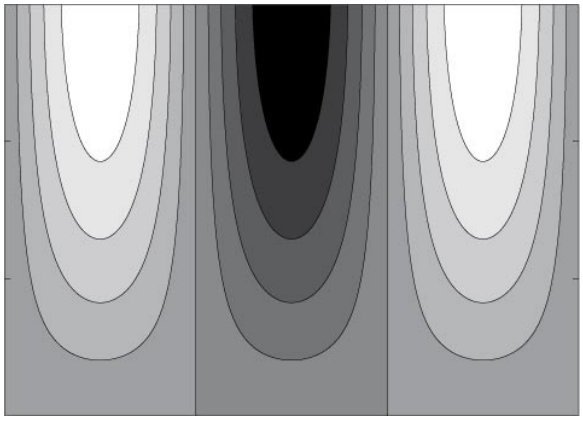

Figure 2.7 | Spatial sines

The plots produced by **contourf** can be made smoother by specifying a number of contours and removing the lines separating contours.

```
contourf(w,w,x,40,'linecolor','none')
```

Note the difference in the default y-axis direction between **imagesc** and **contourf**.

```
subplot(211)
contourf(w,w,x,40,'linecolor','none')
subplot(212)
imagesc(w,w,x)
```

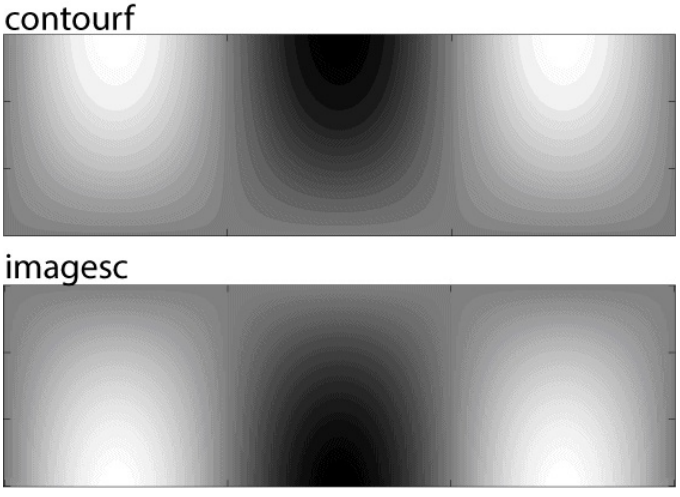

Figure 2.8 | Alternating spatial sines

Chapter 3: Simulating time series data

The purpose of this chapter is to provide an introduction to simulating time series data with characteristics that are frequently observed in real data. The simulated data used in the rest of the book to illustrate and evaluate different time-frequency analyses are based on methods provided in this chapter.

3.1 | Uniformly and normally distributed white noise, and pink noise

Noise can be generated via the functions **`rand`** (uniformly distributed) and **`randn`** (normally [Gaussian] distributed). The inputs to these functions specify the size of the resulting matrices.

```
Yu=rand(1000,1); Yn=randn(1000,1);
subplot(211)
plot(Yn), hold on, plot(Yu,'r')

subplot(223), hist(Yu,200)
 title('Distribution of uniform noise')
subplot(224), hist(Yn,200)
 title('Distribution of random noise')
```

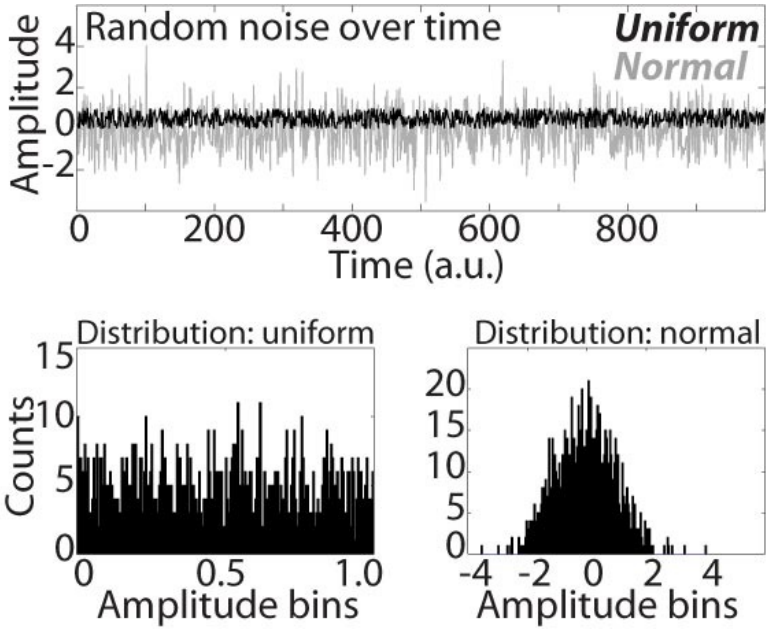

Figure 3.1 | Uniform and random noise

The term “white noise” refers to noise that has a flat power spectrum. The functions **`rand`** and **`randn`** produce data that have roughly flat power spectra and thus can be considered white noise.

“Pink noise” refers to noise with a non-uniform frequency structure—typically, that the power decreases with increasing frequency. There are several ways to compute pink noise; one is to apply a vanishing frequency filter. The code is presented below; more details on the **`fft`** and **`ifft`** functions and filtering will be provided in chapters 4 and 7. FFT stands

for “fast Fourier transform.”

```
wn = randn(10000,1); % wn = white noise
wnX = fft(wn);

pn = real(ifft( wnX .* ...
     linspace(-1,1,length(wnX))'.^2 ))*2;

subplot(221)
plot(wn), hold on
plot(pn,'r')
xlabel('Time (a.u.)')
ylabel('Amplitude (a.u.)')
legend({'white','pink'})

subplot(222)
plot(wn,pn,'.')
xlabel('Amplitude white noise')
ylabel('Amplitude pinkified noise')

subplot(212)
plot(abs(fft(wn))), hold on
plot(abs(fft(pn)),'r')
legend({'white';'pink'})
xlabel('Frequency (a.u.)'), ylabel('Amplitude')
```

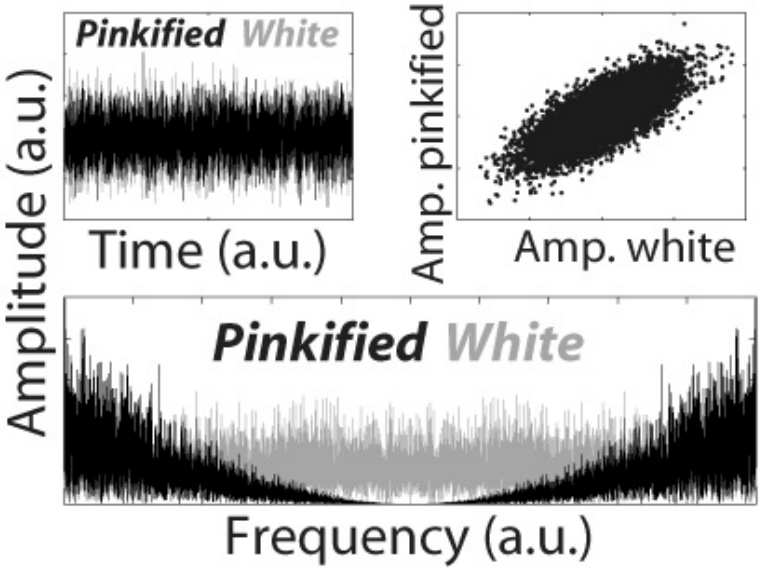

Figure 3.2 | White noise, pink noise (amp. = amplitude).

“a.u.” stands for “arbitrary units” and is often used when the values of the axes are uninterpretable or irrelevant.

3.2 | Sine waves

Sine waves form the basis of many frequency and time-frequency analyses. A sine wave can be created with three parameters: frequency (the speed of the sine wave), amplitude (the height or amount of energy of the sine wave), and phase (the timing of the sine wave).

Frequency is arguably the most important parameter; amplitude and phase can be implicitly set to 1 and 0, respectively.

The formula for a sine wave function of time is: ***y=asin(2pft+j)***, where ***a*** is the amplitude (height of the sine wave, which can be measured as one-half of the trough-to-peak distance on the y-axis), ***p*** is pi (3.141…; Greek characters are avoided here to prevent e-reader typographical errors), ***t*** is time in seconds, ***f*** is frequency in Hz (1/second), and ***j*** is the phase angle in radians, which defines the amplitude of the sine wave when it passes through t=0.

```
t = 0:.001:5; % time: 0 to 5 sec. in 1-ms steps
a = 10; % amplitude
f = 3; % frequency in Hz
p = pi/2; % phase angle

y = a*sin(2*pi*f*t+p);
plot(t,y)
xlabel('Time (s)'), ylabel('amplitude')
```

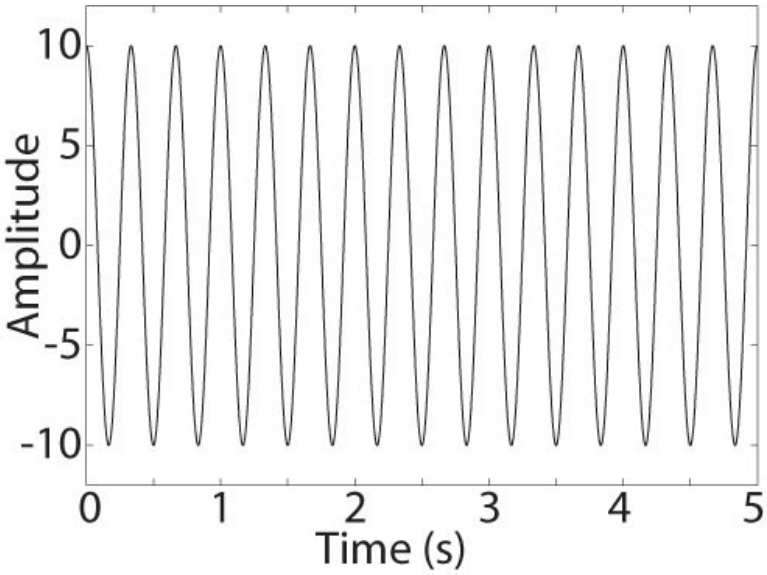

Figure 3.3 | Sine wave

`t` is a vector of time points in seconds, in steps of 0.001 seconds (that is, 1 ms). This means the time series has a sampling rate of 1000 Hz, or 1 kHz.

Multiple sine waves of different amplitudes, phases, and frequencies, can be summed. The resulting time series may be difficult to interpret in the time domain, but the individual waves can easily be isolated in the frequency domain, as will be demonstrated in the next chapter.

```
t = 0:.001:5; % sampling rate of 1000 Hz
n = length(t);
a = [10 2 5 8];
f = [3 1 6 12];
p = [0 pi/4 -pi pi/2];

swave = zeros(size(t));
for i=1:length(a)
```

```
  swave = swave + a(i)*sin(2*pi*f(i)*t+p(i));
end
plot(t,swave)
xlabel('Time (s)'), ylabel('amplitude')
```

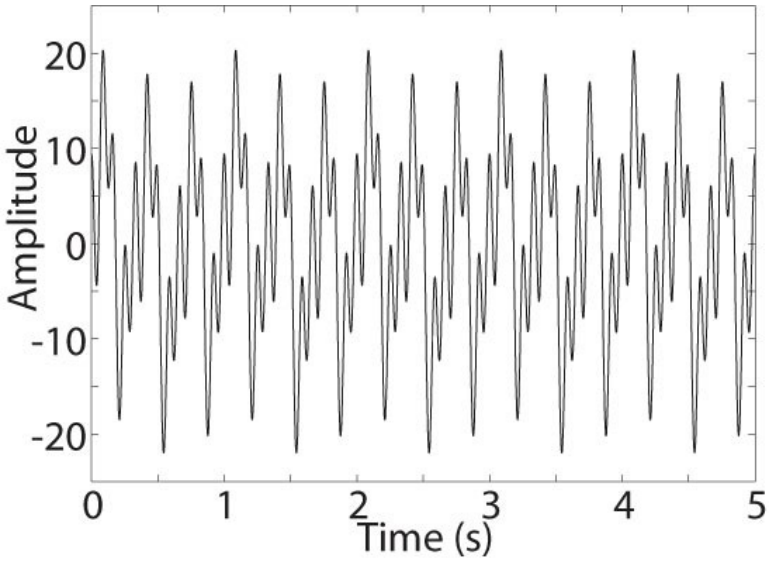

Figure 3.4 | Several summed sines

Noise can be added to sine waves.

```
swave = swave + mean(a)*randn(size(t));
plot(t,swave)
```

In addition to containing sine waves of multiple frequencies simultaneously, sine waves can also contain sudden changes in frequency and in amplitude.

```
a = [10 2 5 8];
f = [3 1 6 12];
% non-overlapping time chunks
tchunks = round(linspace(1,n,length(a)+1));
swave = 0;
for i=1:length(a)
  swave = cat(2,swave,a(i)* ...
   sin(2*pi*f(i)*t(tchunks(i):tchunks(i+1)-1)));
end
plot(t,swave)
```

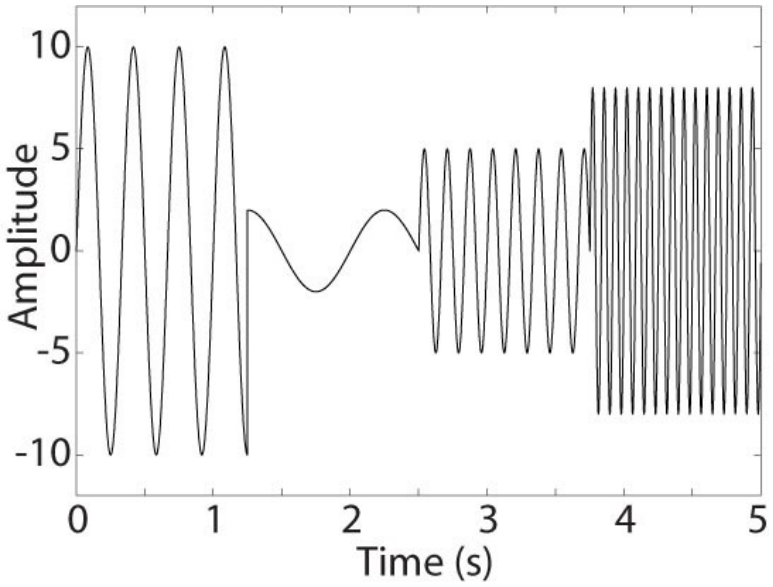

Figure 3.5 | Sine wave changes

Time-varying changes in frequency can also be smooth. For linear or quadratic changes in frequency, the resulting signal is often called a “chirp” or a sweep signal. A linear chirp can be computed by having the frequencies vary as a function of time, scaled by the slope of the change in frequency. Chapter 8.1 will show a more general method for generating arbitrary time-varying changes in oscillation frequency.

```
f = [2 10]; % frequencies in Hz
ff = linspace(f(1),f(2)*mean(f)/f(2),n);
swave = sin(2*pi.*ff.*t);
plot(t,swave)
```

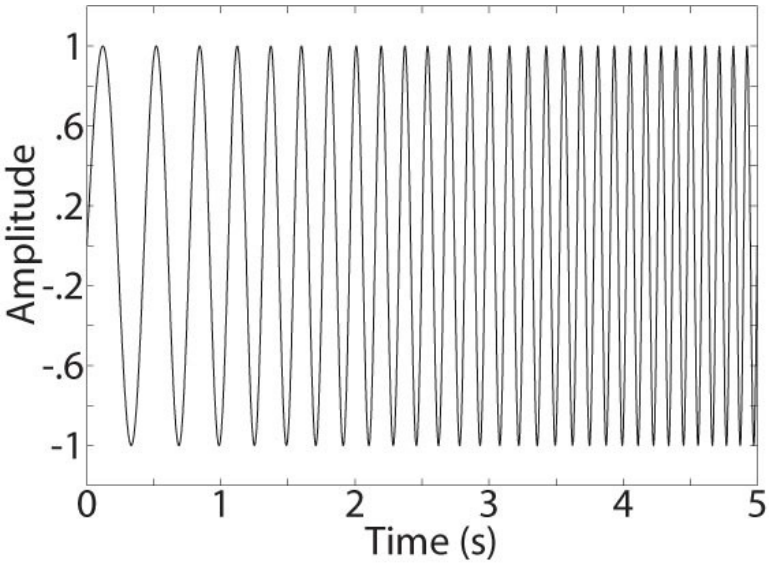

Figure 3.6 | Chirp in time

Sine waves need not be amplitude-stationary over time. Indeed, time-varying changes in the amplitude of oscillations is the primary outcome measure in many time-frequency analyses and real-world applications.

```
a = linspace(1,10,n); % time-varying amplitude
f = 3; % frequency in Hz
```

```
y = a.*sin(2*pi*f*t);
plot(t,y)
```

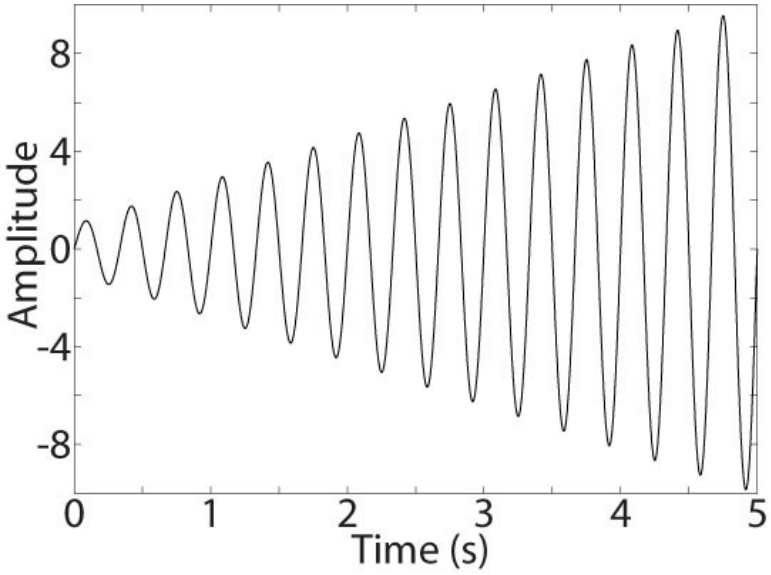

Figure 3.7 | Sine wave, getting stronger.

3.3 | Gaussian

Another important time-domain function is the Gaussian, which is often used to taper, or dampen, parts of a time series. The formula for a time-domain Gaussian is $ae[-(t-m)^2/2s^2]$ (note that e[x] indicates e^x; this convention is used to ensure readability on all e-readers). In this equation, ***a*** is the peak amplitude of the Gaussian, ***t*** is time, ***m*** is the time of the peak of the Gaussian, and ***s*** is the standard deviation of the Gaussian (later in this book, $2s^2$ will be referred to as ***w***, for width). If ***m*** is not specified, the peak of the Gaussian will be at the zero point of time.

```
t = -1:.001:5;
s = [.5 .1]; % widths
a = [4 5]; % amplitudes

g1 = a(1)*exp( (-(t).^2)   /(2*s(1)^2) );
g2 = a(2)*exp( (-(t-2).^2) /(2*s(2)^2) );

plot(t,g1), hold on
plot(t,g2,'r')
```

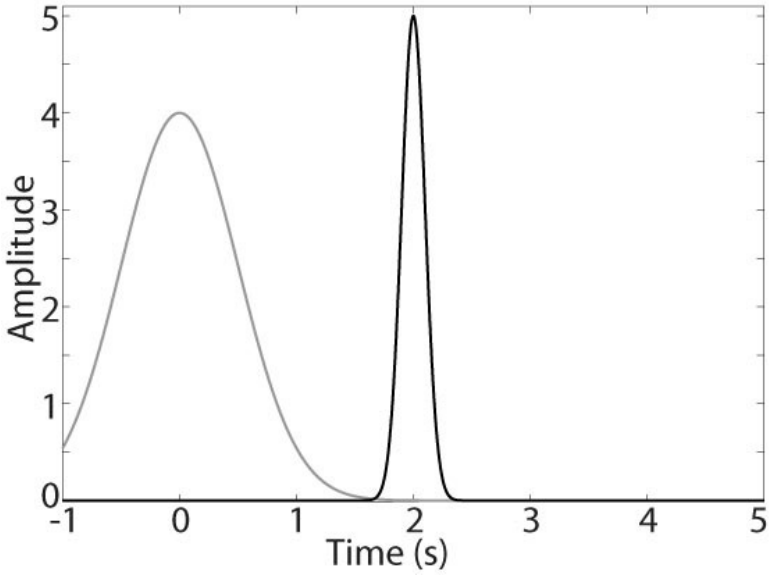

Figure 3.8 | Two Gaussians, similar yet distinct.

The two Gaussians plotted here differ by three parameters: Their peak time (**`m`**), standard deviation (**`s`**), and amplitude (**`a`**).

3.4 | Square waves

Not all time series are smooth; some are squares.

```
t = 0:.01:11;
plot(t,mod(t,2)>1), hold on
plot(t,.02+(mod(.25+t,2)>1.5),'r')
set(gca,'ylim',[-.1 1.1])
```

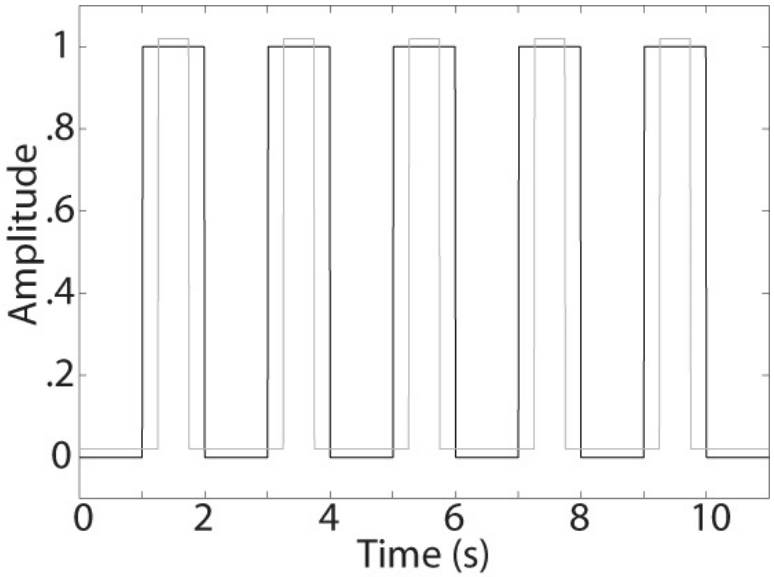

Figure 3.9 | Square waves

Here, the modulus (function **`mod`**) in combination with the Boolean true/false test produces a time series of 0's and 1's that can vary in width (in the above code, 1 or 1.5; this corresponds to 50% or 75% of the modulus-2 result) and peak (the **`.25`** added to variable t in the second line). The y-axis offset in the second time series (**`.02+`**) is added

to facilitate visual inspection.

More generally, square waves can be created by converting a time series of numbers to a time series of Boolean outcomes. Thus, to define frequency specificity of a box-like time series, a Boolean test can be performed on amplitude values of a sine wave. This is demonstrated in exercise 1 at the end of this chapter.

Time series can also have triangular shapes, or more unusual shapes.

```
t = 0:.01:11;
plot(t,abs(mod(t,2)-1.25)), hold on
plot(t,abs(mod(t,2)-1),'r')
```

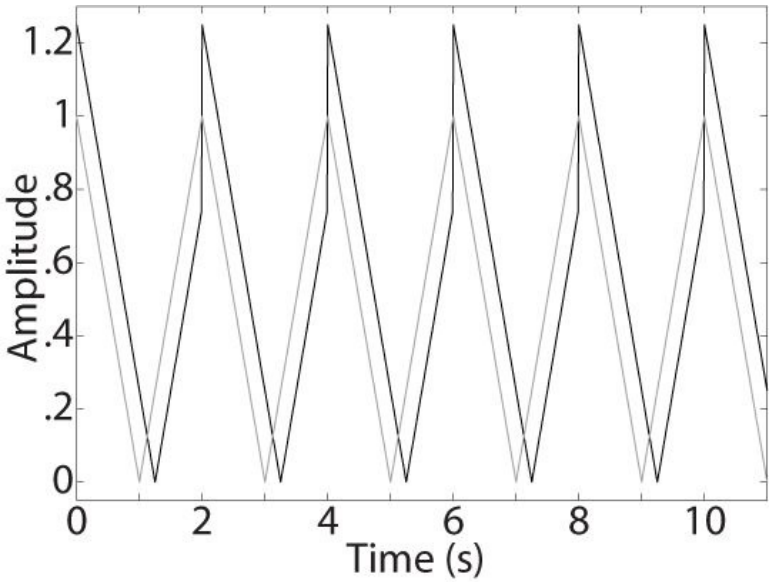

Figure 3.10 | Triangles, regular and otherwise

In these functions, the subtraction of 1.25 and 1 are meaningful relative to the modulus (thus, 1 indicates 50% of 2; this produces triangles).

3.5 | Other time-domain functions

This chapter provides only an introduction to possible ways to simulate time series data. Many other time-domain functions can be produced by combining basic functions.

```
t = 1:.01:11;
subplot(221), plot(t,sin(exp(t-5)))
subplot(222), plot(t,log(t)./t.^2)
subplot(223), plot(t,sin(t).*exp((-(t-3).^2)))
subplot(224), plot(t,abs(mod(t,2)-.66) ...
                    .*sin(2*pi*10*t))
```

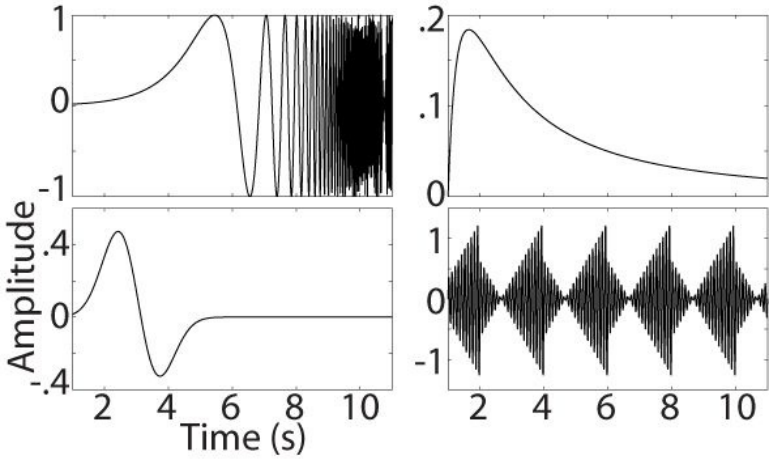

Figure 3.11 | Diversity in time series

3.6 | Stationary and non-stationary time series

Stationarity means that the statistical properties of a time series do not change over time. One can speak of mean-stationarity, variance-stationarity, frequency-stationarity, covariance-stationarity (for a multidimensional time series), and so on. Examples of stationary and non-stationary time series are shown below.

```
t=0:50; a=randn(size(t));
a(3:end)=a(3:end)+.2*a(2:end-1)-.4*a(1:end-2);

b1=rand;
for i=2:length(a)
  b1(i) = 1.05*b1(i-1) + randn/3;
end

subplot(221), plot(t,a)
subplot(222), plot(t,b1)
```

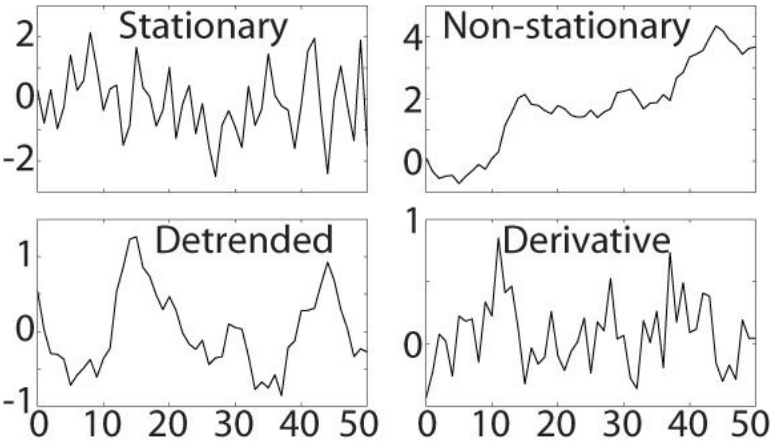

Figure 3.12 | Stationarity and its correction (see below for explanation of lower two panels)

Stationarity is an important concept in time series analysis, because many analyses (including the Fourier transform) assume that the data are stationary. Violations of stationarity do not necessarily make the analyses incorrect, but they can make some results difficult to interpret. This will be demonstrated in later chapters.

There are several simple techniques to make a non-stationary time series more stationary. One is detrending, which involves removing a linear fit to the data.

```
subplot(223), plot(t,detrend(b1))
```

The detrended time series is shown in the lower left subplot of Figure 3.12.

Another method is to take the derivative (the value at each time point is re-assigned to be the difference between that and the value at the previous time point). In the context of time-frequency analyses, this procedure is sometimes referred to as pre-whitening, and attenuates low-frequency activity. Because the derivative will reduce the number of time points by one, the time series can be padded at the end.

```
b1deriv=diff(b1); b1deriv(end+1)=b1deriv(end);
subplot(224), plot(t,b1deriv)
```

The derivative time series is shown in the lower right subplot of Figure 3.12.

Another approach to making non-stationary data more stationary is to apply a filter (high-pass, low-pass, or band-pass) to the data. Filtering is discussed in Chapters 6 and 7.

In general, there is no perfect solution for making non-stationary time series stationary. In the example shown above, detrending helped, but the time series retained some local non-stationarities. Pre-whitening (taking the derivative) also helped, but resulted in some loss of real signal, particularly in lower frequencies.

Methods to identify and statistically quantify stationarity are addressed in more detail in Chapter 9.

3.7 | Precision vs. resolution

Precision and resolution are different but easily confused terms. Resolution is simply the number of measurement points per unit of time, while precision is related to the amount of information that each data point contains. Consider measuring a sine wave: The resolution is determined by the sampling rate, while the precision is related both to the sampling rate and to the frequency of the sine wave.

First consider a 1-Hz sine wave. If the sampling rate is 100 Hz vs. 1000 Hz, the resolution changes. However, because the sampling rate is much faster than the sine wave, the precision remains roughly the same, which is to say, a data point in the 100-Hz signal and its corresponding point in the 1000-Hz signal contains a similar amount of information about the 1 Hz sine wave.

```
srates = [100 1000];
t1=0:1/srates(1):2; t2=0:1/srates(2):2;
sine1=sin(2*pi*t1); % f implicitly set to 1
sine2=sin(2*pi*t2);
plot(t1,sine1,'bo'), hold on
plot(t2,sine2,'r.')
```

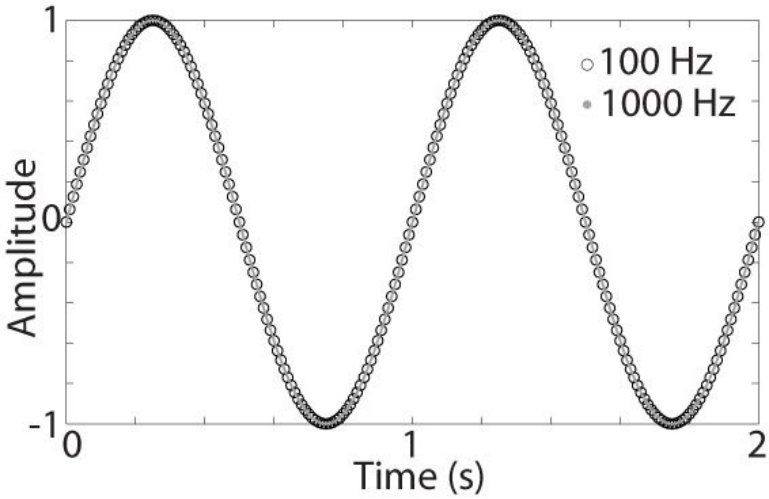

Figure 3.13 | A sine, sampled and oversampled

The question is whether the sine wave is better represented by the 1000 Hz or the 100 Hz sampling rate measurement. In this case, it seems that both sampling rates have similar precisions, even though their resolutions differ by an order of magnitude.

In contrast, if the sampling rate were 3 Hz, it is clear that the 100-Hz measurement has higher precision.

```
srates = [100 3];
t1=0:1/srates(1):2; t2=0:1/srates(2):2;
sine1=sin(2*pi*t1);
sine2=sin(2*pi*t2);
plot(t1,sine1,'bo'), hold on
plot(t2,sine2,'r.-')
```

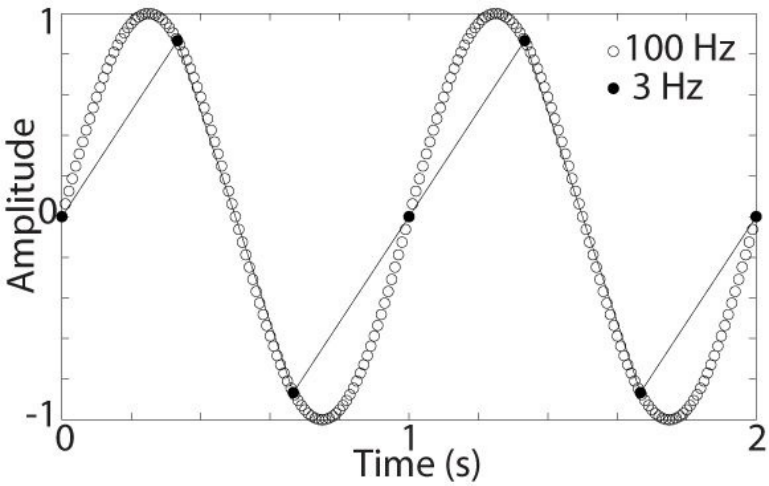

Figure 3.14 | A sine, sampled and undersampled

Precision and resolution also characterize the frequency domain. Frequency resolution simply refers to the number of frequency bins that can be measured (which, as will be explained in Chapter 4, is determined by the number of time points), whereas frequency precision is related to how much information each frequency bin contains relative to the characteristics of the time series.

If a time series contains a frequency response that spans a range (e.g., 30-35 Hz, rather than exactly 32.521586 Hz), it may not be necessary to sample each frequency with 0.001

Hz resolution; in this case, the resolution would be higher than the precision.

Resolution is important for time-frequency analyses because it defines which frequencies can be extracted out of the data. Precision is important because it determines whether a higher or lower resolution is needed. This will be explained in more detail in later chapters.

3.8 | Buffer time at the start and end of the time series

It is ideal to have data before and after the event of interest. For example, if the "event of interest" is a bird call, the recording should start a few seconds before the start of the bird call and should end a few seconds after the end of the bird call. The extra times before and after the event of interest are called "buffer" periods.

Buffer periods are useful for three reasons. First, sudden changes in time series amplitude can introduce artifacts in time-frequency results. These "edge artifacts" will be demonstrated in Chapter 4, and their effects on the results can be minimized by having buffer periods. The edge artifacts will contaminate the buffer periods but will subside by the time the signal of interest begins.

Second, many time-frequency analyses require computing frequency dynamics in temporal windows surrounding each time point; thus, in order to compute frequency dynamics at the start and end of a time series, it is necessary to have some data before and after the signal under investigation.

Third, for some applications, it is useful to have a pre-event "baseline" period against which to compare the time-frequency characteristics of the signal. This is useful when the signal reflects changes on top of ongoing background activity. These three reasons will become clearer in subsequent chapters (mainly Chapter 6.8).

There is no specific amount of time that is necessarily appropriate as a buffer for all time series. In general, three cycles of the lowest frequency that will be analyzed should be sufficient (for example, this would be 30 ms if the lowest frequency is 100 Hz), but if there are large edge artifacts or if an additional pre-signal baseline period is necessary, a longer buffer time may be useful.

If the recording is already made and there is no buffer period, or if the buffer periods are short, a suitable alternative is reflection. Reflection means that the time series is flipped around and attached to the beginning and to the end of the time series. These backwards periods can be used as buffer zones (although they cannot be considered baseline activity), and are then trimmed off after the time-frequency analysis is performed.

```
t=0:.01:10;
x = sin(t).*exp((-(t-3).^2));
xflip = x(end:-1:1);
reflectedX = [xflip x xflip];
subplot(211), plot(x)
subplot(212), plot(reflectedX)
```

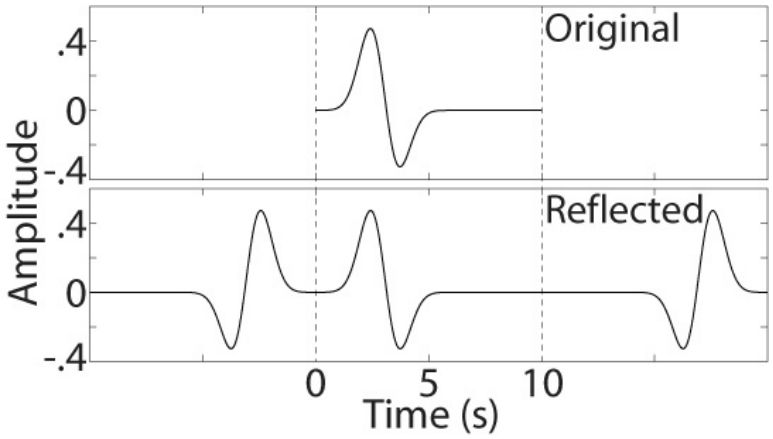

Figure 3.15 | Time series, reflected

3.9 | Generating multivariate time series

Although the main focus of this book is on time-frequency analyses of individual (univariate) time series, basic analyses of multivariate time series—those containing time series from several channels recorded simultaneously—are introduced in a few places.

A multivariate time series of correlated random numbers can be generated by specifying a covariance matrix (covariance refers to the unscaled correlations among all pairs of variables) and a number of time points to generate. In the implementation of correlated multivariate simulations shown in this book, the covariance matrix must be positive-definite (obtained by multiplying the matrix by its transpose), and must have the Cholesky decomposition applied.

```
% covariance matrix
v = [1 .5 0; .5 1 0; 0 0 1];
% Cholesky decomposition of
% positive semi-definite covariance
c = chol(v*v');

% n time points
n = 10000;
d = randn(n,size(v,1))*c;
```

In the code above, the matrix **`d`** contains a 10,000 X 3 matrix of random numbers, such that the first two are correlated at around 0.8 while the third is uncorrelated to the first two. Note that **`cov(d)`** (**`cov`** is the Matlab function to compute a covariance matrix) is very similar to **`v*v'`**.

This is a basic algorithm for generating multivariate datasets, and will suffice for this book (mainly Chapter 11). The Matlab statistics toolbox contains several functions that will create more specialized sets of multivariate data. Interested readers can refer to the functions **`mvnrnd`** and **`mvnpdf`** to get started.

3.10 | Exercises

1) Generate a time series of square-wave-modulated sine waves, such that the sine waves are present only when the square wave is in the ‘upper’ state.

```
%%% Solution to exercise 1
% There are several ways to produce this wave.
% One is to create a stationary sine wave
%  and a square wave with values of 0 or 1.
% When the signals are point-wise multiplied,
%  the result is the solution.

% first, the basics:
srate    = 1000;
time     = 0:1/srate:6;
sinefreq = 10; % in hz
boxfreq  =  1; % in hz

% second, create a sine wave
sinewave = sin(2*pi*sinefreq*time);

% third, create a box wave
boxwave  = sin(2*pi*boxfreq*time)>0;

% fourth, point-wise multiply the two signals
solution = boxwave .* sinewave;

% Although the above is a correct solution,
%  it would be nice to see the box better.
solution = solution + 3*boxwave;

% and plot the result
plot(time,solution)
```

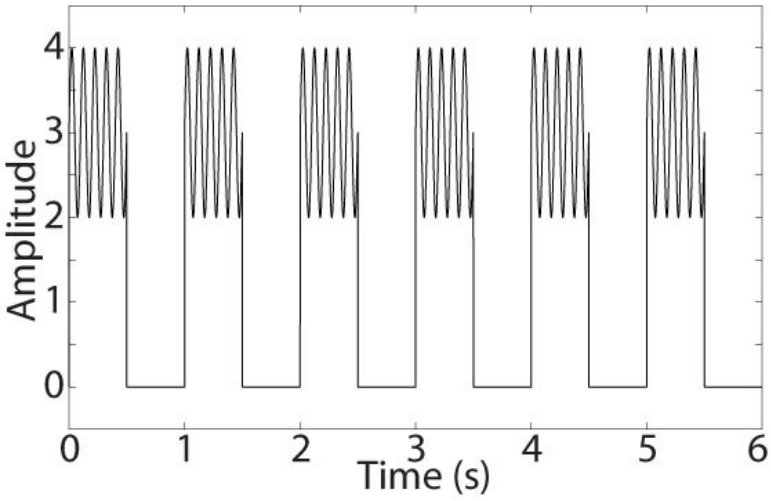

Figure 3.17 | Squares take sines

2) Generate a time series by combining sine waves of several frequencies. How many can you add together and still recognize the individual sine wave components?

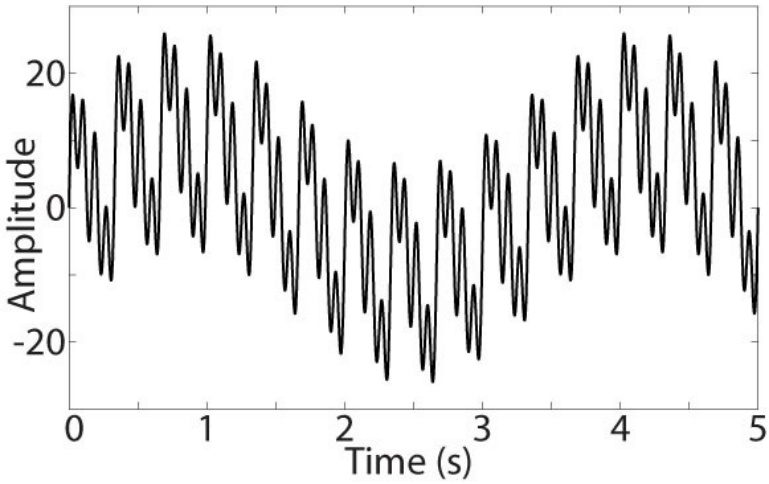

Figure 3.18 | About four sine waves can be distinguished, depending on their relative frequencies and amplitudes.

3) Generate a time series by combining three sine waves of different frequencies (all with constant amplitude of 1). Make sure you can visually identify all three sine wave components. Now add random noise. What is the (approximate) amplitude of noise such that the individual sine waves are no longer visually recognizable?

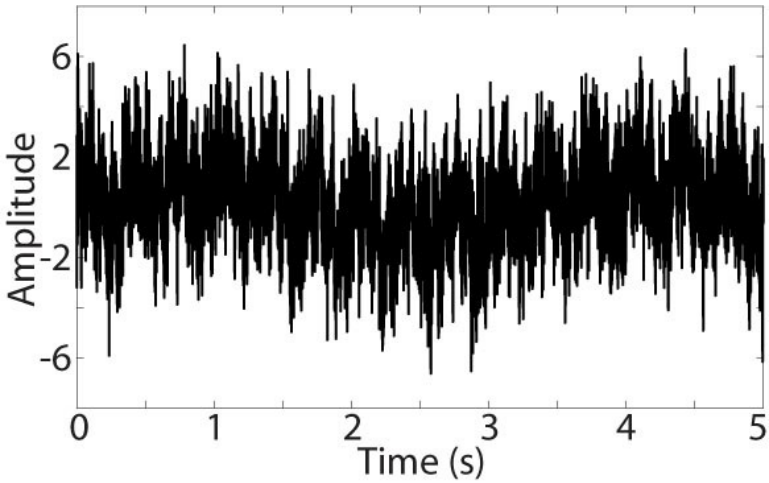

Figure 3.19 | A noise factor of ~1.5 is the maximum amount of noise such that the sine waves are still somewhat visible.

4) Generate two Gaussian time series, one with stationary noise added and one with non-stationary noise added.

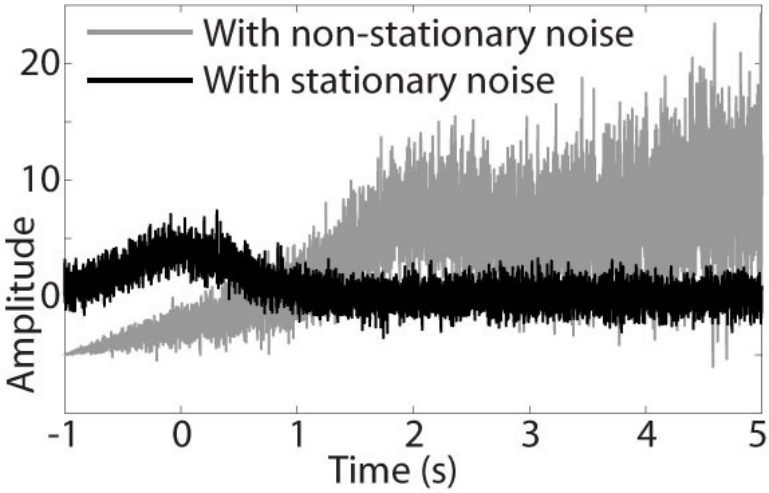

Figure 3.20 | Two Gaussians with noise.

5) Create two overlapping sinc functions of different frequencies and different peak latencies. A sinc function generally takes the form of *sin(t)/t*, and can be expanded to frequency and peak-timing specificity using, in Matlab code, **`sin(2*pi*f*(t-m))./(t-m)`**, where **`f`** is the center frequency, **`t`** is a vector of time indices in milliseconds, and **`m`** is the peak time of the sinc function. Note that when **`t-m`** is zero, the function will have an NaN value (not-a-number). This can cause difficulties in frequency and time-frequency analyses, and should be replaced with the average of surrounding time points.

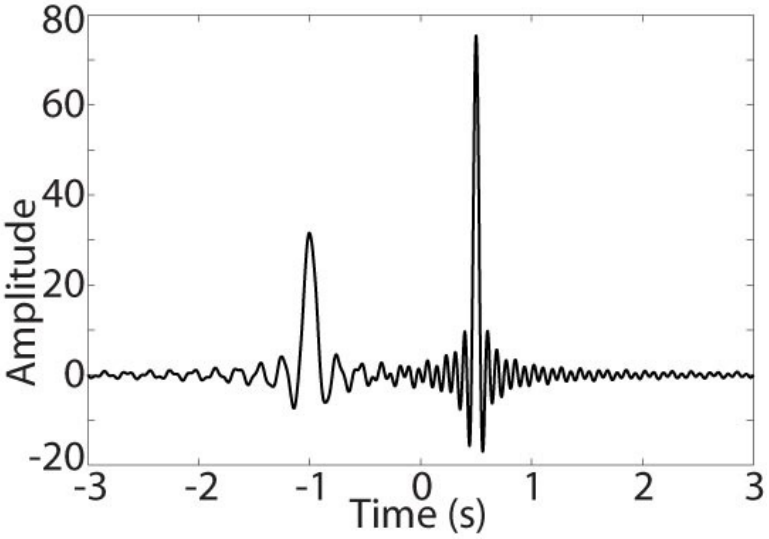

Figure 3.21 | Two sincs, summed.

Chapter 4: The Fourier transform and the inverse Fourier transform

The Fourier theorem states that any time series can be represented using a sum of sine waves of different frequencies, phases, and amplitudes. In time series analysis, the purpose of a Fourier transform is to represent the time series in the frequency domain. This is useful both to examine the characteristics of the time series, as well as to transform the data, e.g., in band-pass filtering.

4.1 | Complex sine waves

The basis of the Fourier transform is a complex sine wave. A complex sine wave is similar to a real-valued sine wave shown in Chapter 3, except that it also contains an imaginary part. A complex sine wave is thus a 3-dimensional time series. It is created by embedding the formula for a sine wave into Euler's formula (e^{ik}), which represents the phase angle k as a unit-vector on a polar plane.

```
srate=1000; t=0:1/srate:10; n=length(t);
csw = exp(1i*2*pi*t);% csw=complex sine wave
plot3(t,real(csw),imag(csw))
xlabel('time'), ylabel('real part')
zlabel('imaginary part')
rotate3d % active click-and-drag in Matlab
```

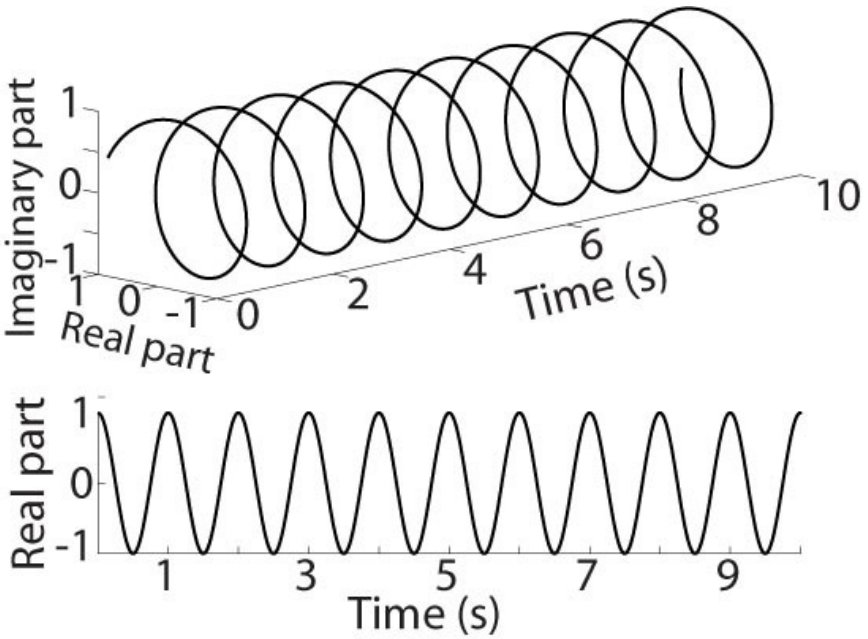

Figure 4.1 | The top panel shows the three dimensions of a complex sine wave; the bottom panel shows the familiar sine wave after rotation.

The imaginary operator is the result of the square root of -1, and can be implemented in Matlab as `1i` or `1j`. `i` or `j` also work but are less preferred because they are often used as counting or indexing variables.

4.2 | The Fourier transform, slow and fast

The Fourier transform of a time series works by computing the dot-product between a number of complex sine waves and the time series. The dot-product is the sum of point-wise multiplications between two vectors of equal length, and indexes the "similarity" between the two vectors (the dot-product is also the basis of a correlation coefficient).

The number of complex sine waves used in a Fourier transform is equal to the number of time points in the data. Because the sine wave is complex, the resulting dot products are complex and are termed "Fourier coefficients."

In Matlab programming, it is convenient to use a similar variable name for the Fourier coefficients as is used for the time-domain data. In this book, this will be indicated by an X after the variable name (X is for spectral [xpectral]). Thus, the Fourier coefficients of a time series variable **`signal`** will be called **`signalX`**. This convenient notation will help increase code readability and prevent confusion.

```
signal = 2*sin(2*pi*3*t + pi/2);
fouriertime = (0:n-1)/n; % n is length(t)

signalX = zeros(size(signal));
for fi=1:length(signalX)
  csw = exp(-1i*2*pi*(fi-1)*fouriertime);
  signalX(fi) = sum( csw.*signal )/n;
end
```

The above formulation is called a discrete-time Fourier transform. Note that the time vector to create the complex sine wave—**`fouriertime`**—must be normalized; it is not time in seconds.

For long time series, this implementation is computationally intensive and thus very slow. Therefore, an implementation of the fast Fourier transform should be used. This will produce equivalent results in much less time.

```
signalXF = fft(signal)/n;
```

Dividing the Fourier coefficients by the number of time points scales the coefficients to the amplitude of the original data.

4.3 | Plotting and interpreting the result of a Fourier transform

Each Fourier coefficient corresponds to one frequency. The frequencies defined in the sine waves are **`fi-1`** (**`fi`** is simply the looping integer counting variable, from 1 to the number of time points in the time series). This means that the first frequency is 0. This is often called the DC frequency (direct-current), and captures the average amplitude offset (if the time series is mean-centered, the DC will be zero). Because the highest frequency that can be measured in a time series is one half of the sampling rate (called the "Nyquist" frequency), the conversion from frequency indices (**`fi`** in the above code) to frequencies

in hertz can be made by computing linearly increasing steps from 0 to the Nyquist in N/2+1 steps, where N is the number of time points in the time series (+1 because of the DC frequency).

```
nyquistfreq = srate/2;
hz = linspace(0,nyquistfreq,floor(n/2)+1);
```

Note that there are one half as many frequencies as there are Fourier coefficients. The frequencies between zero and the Nyquist (corresponding to the first half of the Fourier coefficients) are called “positive frequencies.” The second half are called “negative frequencies” and capture sine waves traveling in reverse (clockwise) direction. For a real-valued time series, the negative frequencies will mirror the positive frequencies, and they are thus often summed onto the positive frequencies. In practice, this is done by ignoring the negative frequencies and multiplying the positive frequency amplitudes by two. However, the negative frequency coefficients are necessary for the inverse-Fourier transform, and thus should not be removed.

Each Fourier coefficient contains information about the amplitude and the phase of each sine wave. They can be extracted using the functions **`abs`** and **`angle`**. Power is amplitude squared (**`abs(`**…**`)^2`**).

Plotting the amplitudes as a function of frequency in the above example reveals one peak at 3 Hz. This is the frequency-domain (or spectral) representation of the sine wave.

```
subplot(211)
plot(hz,2*abs(signalX(1:length(hz)))), hold on
plot(hz,2*abs(signalXF(1:length(hz))),'r')
xlabel('Frequencies (Hz)')
ylabel('Amplitude')
set(gca,'xlim',[0 10])
legend({'slow Fourier transform';...
        'fast Fourier transform'})

subplot(212)
plot(hz,angle(signalX(1:floor(n/2)+1)))
xlabel('Frequencies (Hz)')
ylabel('Phase (radians)')
```

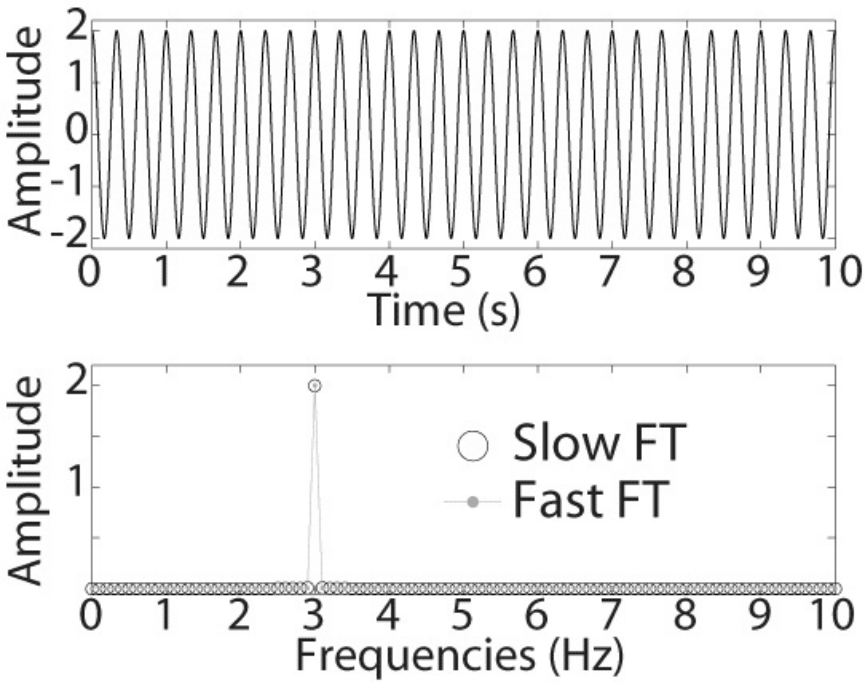

Figure 4.2 | Sine wave, in time (top) and in frequency (bottom)

Amplitude-by-frequency or power-by-frequency is the most typically plotted result of the Fourier transform.

The phase values are related to the position on the y-axis of each constituent sine wave when time (x-axis) is zero. The phase information is crucial for reconstructing the timing of the time series, but is otherwise less frequently used to characterize time series data, compared to power.

A note about plotting results of a Fourier transform: Because this book deals only with discrete sampled time series (as opposed to continuous and more theoretical signals), frequencies are also sampled discretely. As such, it is more appropriate to plot frequencies as dots or bars, without connecting them with lines.

Lines give the impression that there is a continuous and linear transition between successive frequency bins, when in fact this is not necessarily the case. Nonetheless, line plots are useful for visual inspection, particularly when directly comparing multiple results in the same plot. Comparing results of the Fourier transform from different time series is difficult and sometimes impossible with discrete bars.

Thus, caution must be taken when interpreting line plots. In Figure 4.2, for example, there appears to be a transition between 3 Hz and the frequencies before and after (the “carrot” shape). This is a false impression created by discrete sampling; the amplitude actually drops immediately to near-zero with no carrot-shaped decay.

4.4 | The Fourier transform with multiple sine waves and with noise

One of the advantages of the Fourier analysis is that it isolates overlapping sine waves that might be difficult to isolate in the time domain. The code below will perform a Fourier analysis on summed sine waves that were created in Chapter 3.

```
srate=1000; t=0:1/srate:5; n=length(t);
```

```
a=[10 2 5 8]; f=[3 1 6 12];
swave = zeros(size(t));
for i=1:length(a)
  swave = swave + a(i)*sin(2*pi*f(i)*t);
end

% Fourier transform
swaveX = fft(swave)/n;
hz = linspace(0,srate/2,floor(n/2)+1);

% plot
subplot(211), plot(t,swave)
xlabel('Time (s)'), ylabel('amplitude')

subplot(212)
plot(hz,2*abs(swaveX(1:length(hz))))
set(gca,'xlim',[0 max(f)*1.3]);
xlabel('Frequencies (Hz)'), ylabel('amplitude')
```

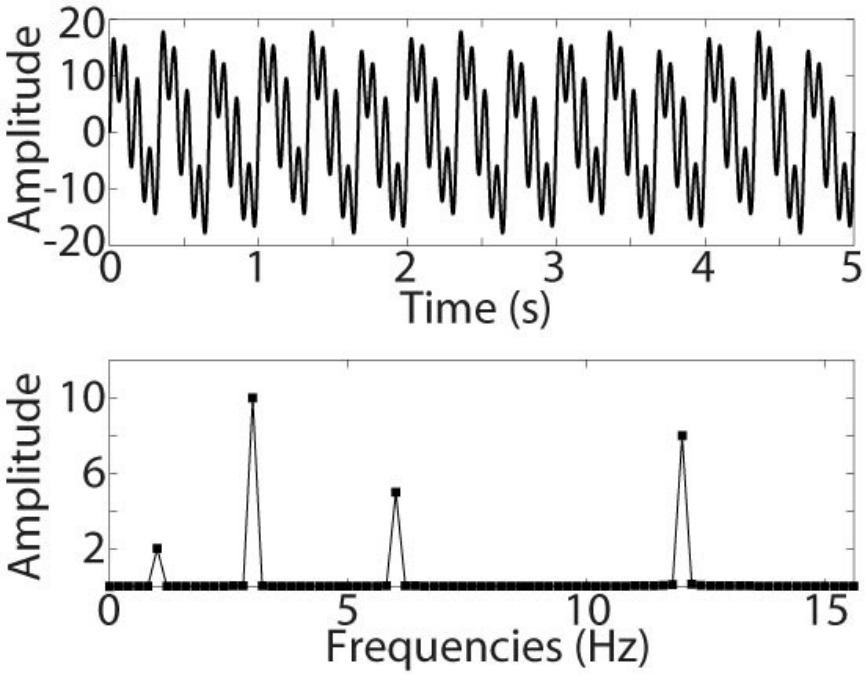

Figure 4.3 | Sine wave components are easier to distinguish in frequency (bottom) compared to time (top).

The lower plot shows four peaks corresponding to the frequencies of the four sine waves. Notice that the frequencies on the x-axis and the amplitudes on the y-axis perfectly match those specified when creating the summed sine wave. It is clearly easier to see the constituent sine waves in the frequency domain compared to the time domain. This becomes more striking when random noise is added to the time-domain signal.

```
swaveN = swave + randn(size(swave))*20;
swaveNX = fft(swaveN)/n;
subplot(211), plot(t,swaveN)
xlabel('Time (s)'), ylabel('amplitude')

subplot(212)
plot(hz,2*abs(swaveNX(1:length(hz))))
```

```
set(gca,'xlim',[0 max(f)*1.3]);
xlabel('Frequencies (Hz)'), ylabel('amplitude')
```

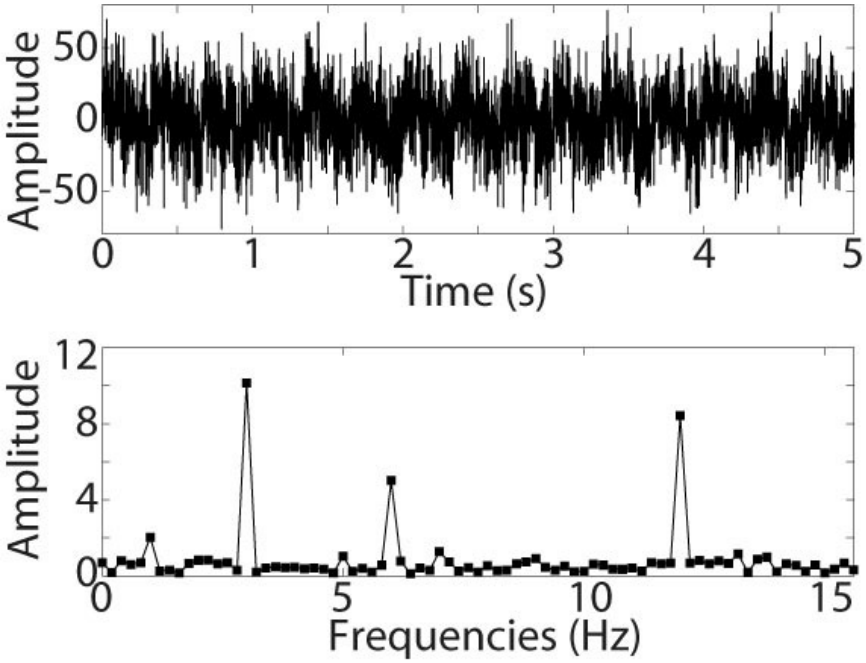

Figure 4.4 | Figure 4.3 with noise.

However, the Fourier transform is not impervious to noise. Try changing the noise scaling from 20 to 100. Are the sine wave components still detectable?

4.5 | Extracting information about specific frequencies

The spectral plots shown so far provide useful qualitative information about the frequency content of time series data. However, it is also often useful to extract quantitative information from specific frequencies.

Frequencies in indices do not necessarily correspond to frequencies in hertz. For example, the 10 Hz component from the above signal cannot be extracted by typing **`swaveX(10)`**. That is, the tenth element in the **`swaveX`** vector does not correspond to 10 Hz. In the example above, the tenth element corresponds to 1.8 Hz (to obtain this result, type **`hz(10)`**).

Instead, it is necessary to search through the frequencies label vector **`hz`** to find the closest match to the desired frequency. This can be achieved in one of two ways.

```
[junk,tenHzidx] = min(abs(hz-10));
tenHzidx = dsearchn(hz',10);
```

Both approaches return the number 51, meaning that the 51st element in the **`hz`** (and, thus, in **`swaveX`**) vector corresponds to 10 Hz.

Now the results can be extracted from specific frequencies for further analysis.

```
frex_idx = sort(dsearchn(hz',f'));
requested_frequences = 2*abs(swaveX(frex_idx));
```

```
bar(requested_frequences)
xlabel('Frequencies (Hz)'), ylabel('Amplitude')
set(gca,'xtick',1:length(frex_idx),...
        'xticklabel',cellstr(num2str(round(hz(frex_idx))')))
```

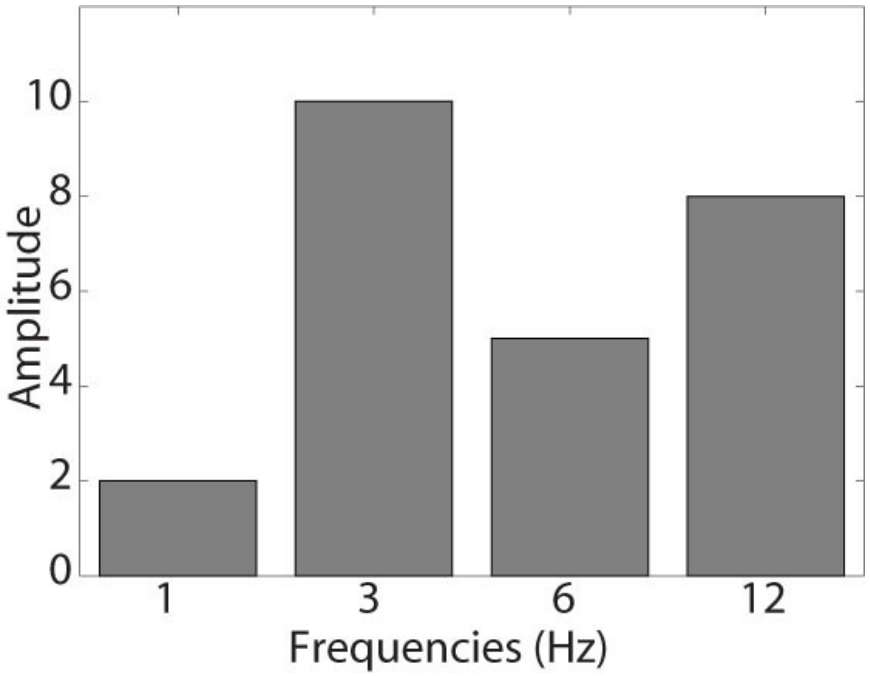

Figure 4.5 | Bars of power.

4.6 | The Fourier transform with non-stationary sinusoidal signals

Consider a sine wave with constant frequency but varying amplitude.

```
srate=1000; t=0:1/srate:5; n=length(t);
f = 3; % frequency in Hz
swave  = linspace(1,10,n).*sin(2*pi*f*t);
swaveX = fft(swave)/length(t);
hz = linspace(0,srate/2,floor(n/2)+1);
subplot(211), plot(t,swave)
xlabel('Time'), ylabel('amplitude')
subplot(212)
plot(hz,2*abs(swaveX(1:length(hz))))
xlabel('Frequency (Hz)'), ylabel('amplitude')
```

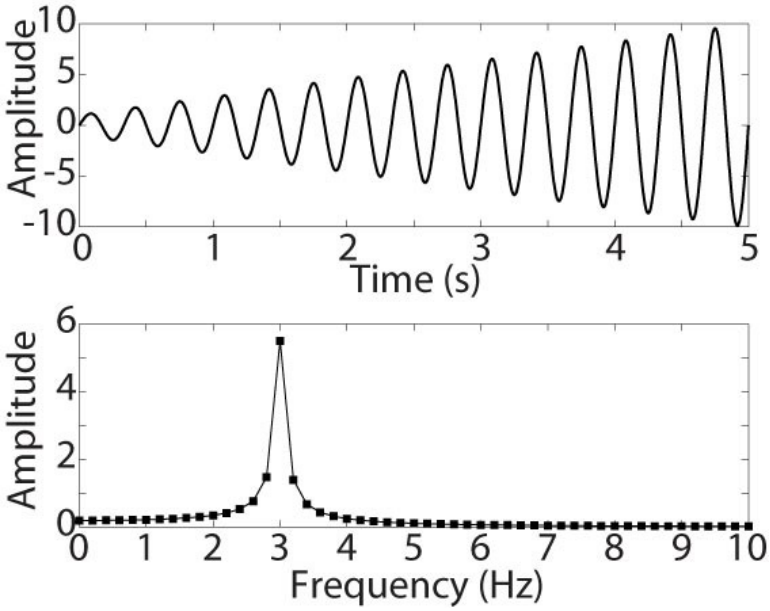

Figure 4.6 | Non-stationarities widen spectral representations, part I.

Next consider a time series with varying amplitudes and varying frequencies.

```
a = [10 2 5 8];
f = [3 1 6 12];

tchunks = round(linspace(1,n,length(a)+1));

swave = 0;
for i=1:length(a)
  swave = cat(2,swave,a(i)* ...
    sin(2*pi*f(i)*t(tchunks(i):tchunks(i+1)-1)));
end

swaveX = fft(swave)/n;
hz = linspace(0,srate/2,floor(n/2)+1);
subplot(211), plot(t,swave)
xlabel('Time'), ylabel('amplitude')
subplot(212)
plot(hz,2*abs(swaveX(1:length(hz))))
xlabel('Frequency (Hz)'), ylabel('amplitude')
```

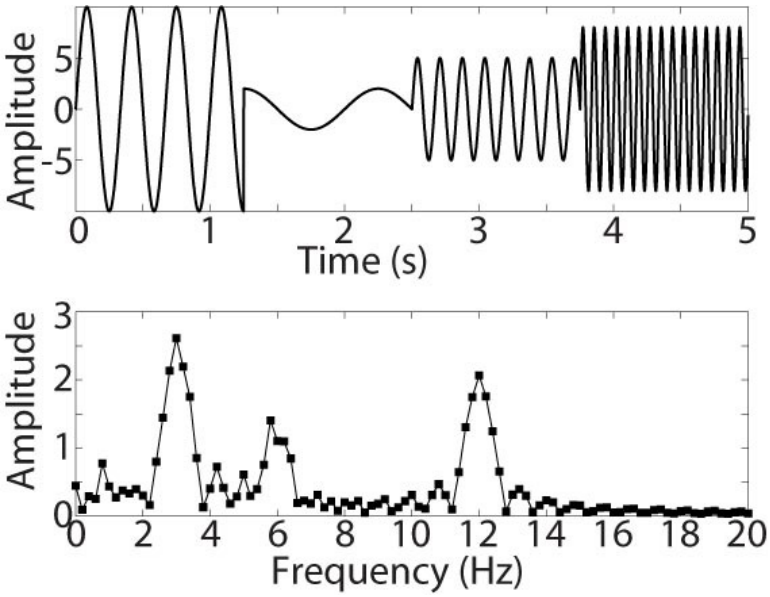

Figure 4.7 | Non-stationarities widen spectral representations, part II.

The power spectrum shows the correct peak frequencies, although they are somewhat smeared out and with side-band ripples. Strikingly, the increased power at frequencies that were not specified when creating the data appears similar to the power spectrum with noise added (Figure 4.4), although no noise was added to the time series here.

Another striking feature of this figure is that the amplitudes are smaller than what was specified when creating the signal, and from inspecting the spectral plot it cannot be inferred that the sine waves were present at different times. In fact, one might look at this plot and think that all four sine waves were presented simultaneously, and random noise was added afterwards.

The temporal information is contained in the phase of the Fourier coefficients. However, inspecting the phase information also does not easily reveal when the different sine waves were present (this can be inspected by replacing **`2*abs`** with **`angle`** in the lower subplot). The difficulty of inferring time-varying information with the Fourier transform is a primary motivation for performing time-frequency analyses.

Finally, consider a chirp signal that contains a sine wave of constant amplitude but time-varying changes in frequency.

```
f = [2 10];
ff = linspace(f(1),f(2)*mean(f)/f(2),n);
swave = sin(2*pi.*ff.*t);

swaveX = fft(swave)/n;
hz = linspace(0,srate/2,floor(n/2));
subplot(211), plot(t,swave)
xlabel('Time'), ylabel('amplitude')
subplot(212)
plot(hz,2*abs(swaveX(1:length(hz))))
xlabel('Frequency (Hz)'), ylabel('amplitude')
```

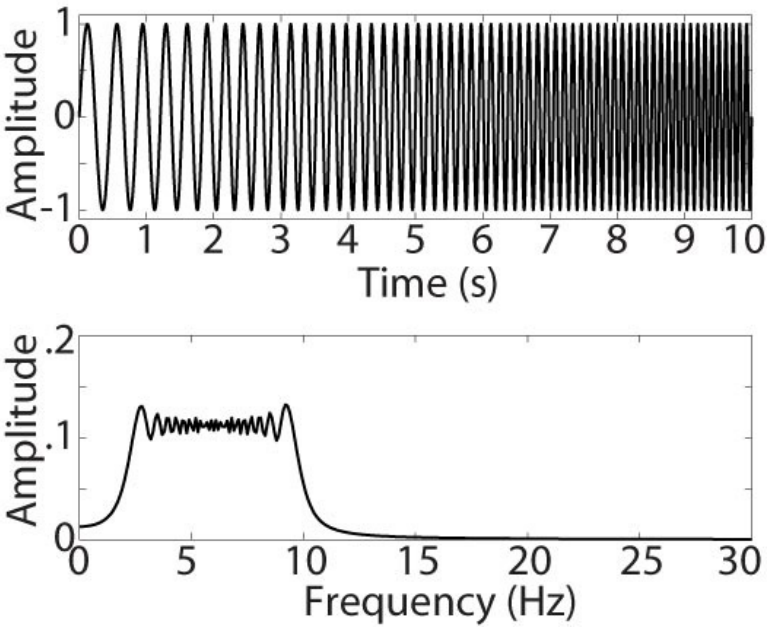

Figure 4.8 | Chirp in time and in frequency

Again, in this case it is not straight-forward to imagine what the time series looks like purely by inspecting the Fourier amplitude spectrum.

This is because one assumption of the Fourier analysis is stationarity, and stationarity is violated in this example. This does not make the Fourier coefficients invalid or inaccurate, but it does make them difficult to interpret in an easy manner. That is, it is difficult to look at the power spectrum and imagine what the time-domain data might look like. This is yet another motivation for performing time-frequency analyses rather than frequency analyses.

There are two important lessons here in comparing the power spectral plots in Figures 4.2, 4.6, 4.7, and 4.8, concerning how to interpret power spectral plots. First, when the frequency peak has a sharp point (e.g., Figure 4.2), the sinusoidal component in the data is frequency-stationary, whereas when the frequency “peak” is more of a flat plateau (e.g., Figure 4.8), there are frequency non-stationarities. Second, when the frequency peak has narrow slopes down from the peak frequency—that is, when the power spectrum has very sharp pointy features—there is little or no change in the amplitude of the sine wave over time. In contrast, when the frequency peak has gentle slopes down from the peak, there are amplitude non-stationarities.

4.7 | The Fourier transform with non-sinusoidal signals

The Fourier transform is based on comparing perfect sine waves with the time-domain data, so it is sensible that the Fourier transform produces easily interpretable results for time series comprising sine waves.

The Fourier transform, however, is not limited to signals that are sinusoidal. It can accurately describe any time series in the frequency domain, even if that time series does not comprise sine waves or other periodic features. However, for non-sinusoidal time series, the frequency-domain representation may be less easily visually interpretable. The example below illustrates this using a times series of repeating boxes.

```
srate=100; t=0:1/srate:11; n=length(t);
boxes = double(.02+(mod(.25+t,2)>1.5));

boxesX = fft(boxes)/n;
hz = linspace(0,srate/2,floor(n/2));
subplot(211), plot(t,boxes)
xlabel('Time'), ylabel('amplitude')

subplot(212)
plot(hz,2*abs(boxesX(1:length(hz))))
xlabel('Frequency (Hz)'), ylabel('amplitude')
```

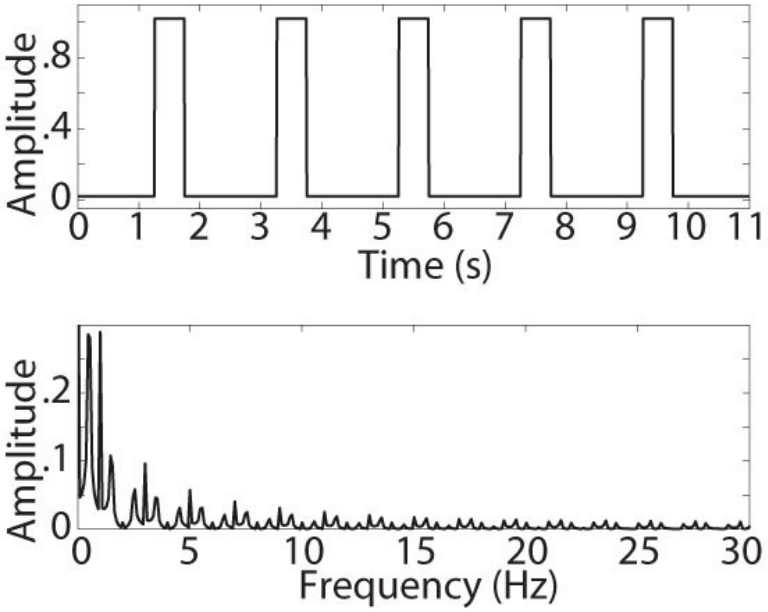

Figure 4.9 | Non-sinusoidal signals have multipeaked spectral forms.

Note that the power spectrum does not look as "clean" as the spectra for sine wave-based time series. Nonetheless, the Fourier coefficients are a perfect frequency-domain representation of the square pulse time series. This can be shown by computing the inverse Fourier transform, which will be demonstrated in Chapter 4.14, and utilized in the following chapters.

4.8 | Edge artifacts and the Fourier transform, and tapering the time series

As shown above in the box example, sharp edges in a time-domain signal can be represented perfectly by a Fourier transform, but have a "messy" spectral representation. This is because many smooth sine waves must be summed to produce a straight line.

```
x=(linspace(0,1,1000)>.5)+0;
% the +0 converts boolean to numeric
subplot(211), plot(x)
set(gca,'ylim',[-.1 1.1])
xlabel('Time (a.u.)')
subplot(212), plot(abs(fft(x))),
set(gca,'xlim',[0 200],'ylim',[0 100])
xlabel('Frequency (a.u.)')
```

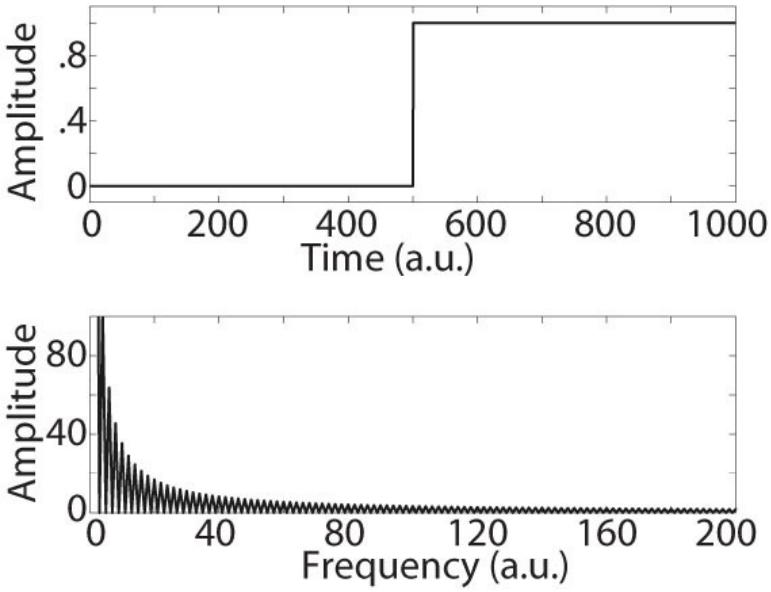

Figure 4.10 | Edge artifacts.

In addition to sudden jumps within a time series, edge artifacts can also occur at the beginning and at the end of a time series, when there is a theoretical jump between zero and the value at the first time point, and between the value at the last time point and zero.

The code below will show the Fourier spectra of two sinusoidal responses with edge artifacts of different magnitudes, which were created by a phase offset in the sine wave (sine to cosine).

```
srate=1000; t=0:1/srate:10; n=length(t);

x1 = sin(2*pi*2*t + pi/2);
x2 = sin(2*pi*2*t);

subplot(211), plot(t,x1)
hold on, plot(t,x2,'r')
xlabel('Time'), ylabel('amplitude')

hz = linspace(0,srate/2,floor(n/2)+1);
x1X = fft(x1)/n;
x2X = fft(x2)/n;
subplot(212),
plot(hz,2*abs(x1X(1:length(hz))),'b.-'),hold on
plot(hz,2*abs(x2X(1:length(hz))),'r.-')
xlabel('Frequency (Hz)'), ylabel('amplitude')
set(gca,'xlim',[0 10],'ylim',[0 .001])
```

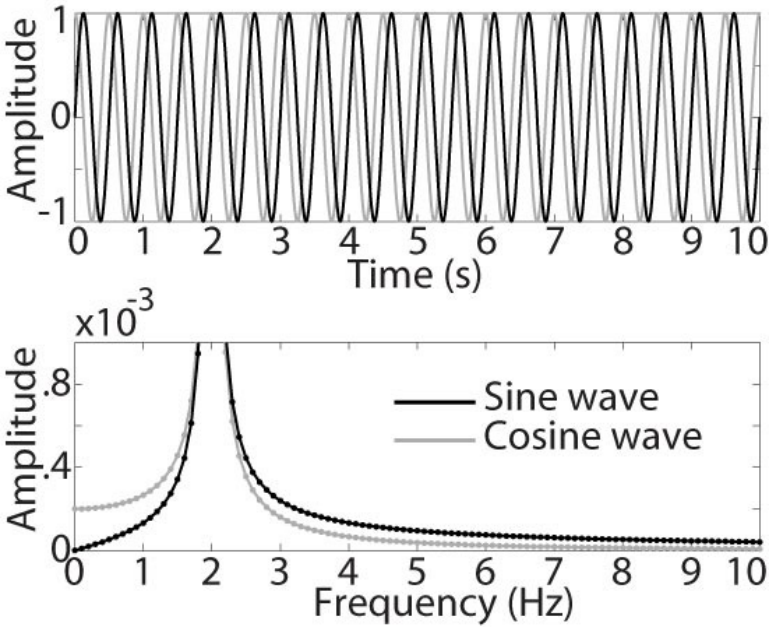

Figure 4.11 | Edge artifacts in cosine

This may seem like a small artifact, but it is also a small edge. In real time series the edge artifacts can be severe enough to impede interpretation.

A solution to this problem is to apply a taper to the time series. A taper is a smooth envelope that dampens the time series near the beginning and the end, thereby effectively removing edge artifacts. There are many tapers that can be applied; a Hann window will be show here (other tapers include a Hamming, Gaussian, Welch, and Blackman, and generally provide similar results; this is demonstrated in the online code).

The Matlab Signal Processing toolbox contains a function called **`hann`** that will return an N-point Hann taper. However, the Hann taper is easy and fast to compute in one line of code, so the special toolbox function is not necessary.

```
hannwin = .5*(1-cos(2*pi*linspace(0,1,n)));
subplot(311), plot(t,x1)
subplot(312), plot(t,hannwin)
subplot(313), plot(t,x1.*hannwin)
```

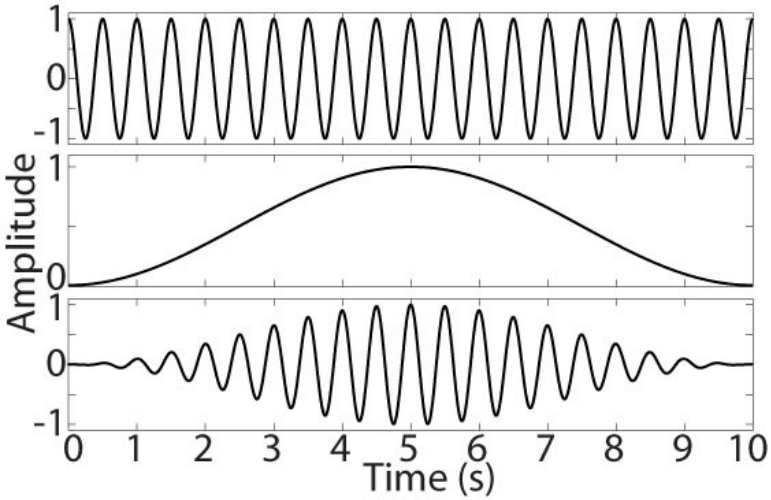

Figure 4.12 | Sine wave (top), taper (middle), and the tapered sine wave (bottom).

Note that applying the taper also attenuates valid signal. This potential loss of signal must be balanced with the attenuation of artifacts introduced by edges.

```
hz = linspace(0,srate/2,floor(n/2)+1);
x1X = fft(x1)/n;
x2X = fft(x1.*hannwin)/n;
plot(hz,2*abs(x1X(1:length(hz)))), hold on
plot(hz,2*abs(x2X(1:length(hz))),'r')
set(gca,'xlim',[0 10],'ylim',[0 1])
```

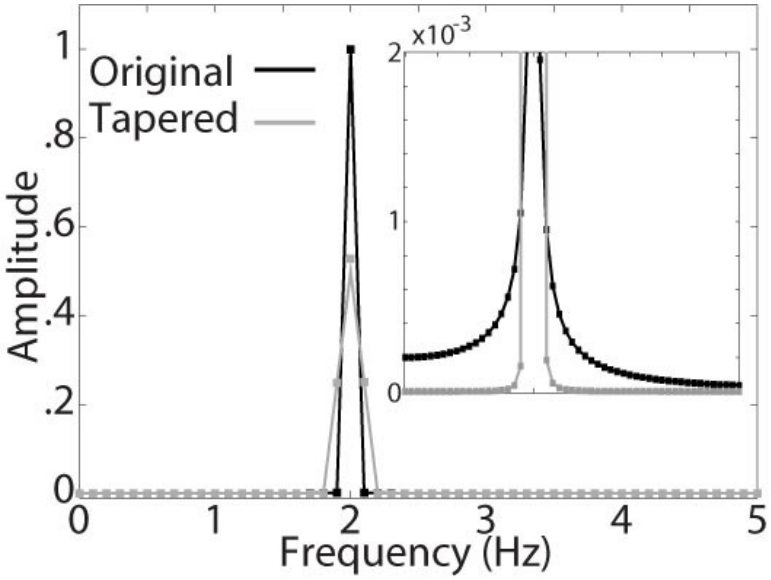

Figure 4.13 | Tapered sine wave reduces edge artifacts. Inset plot shows increased y-axis scaling for comparison.

The tapered sine wave has decreased peak power, due to the attenuation of valid data at the beginning and at the end of the time series. However, the tapered sine wave also has very low side-bands, with power dropping quickly to zero (see inset in Figure 4.13). The non-tapered sine wave, which has a minor edge artifact, has larger power over a broader frequency range.

Whether the time series should be tapered depends on the purpose of computing the Fourier transform. If the goal is to measure peak power at a selected frequency as accurately as possible, applying a taper will reduce the accuracy of the peak measurement. On the other hand, if the shape of the frequency response is more important, then tapering will increase the accuracy of the results by dampening edge artifacts.

In most applications, tapering does more good than harm. Thus, tapering should be the default procedure unless there is a specific reason not to taper (an illustration of when tapering might not be preferred is illustrated in Chapter 10.1). When in doubt, taper.

4.9 | Zero-padding and frequency resolution

As explained in Chapter 4.3, the frequency resolution of the Fourier transform (the number of unique frequencies) is determined by the number of time points in the time series. It is possible to increase the frequency resolution by adding more time points. This can be done by adding extra zeros at the end of the time series, also known as zero-

padding. In Matlab, zero-padding can be achieved via the second input of the **`fft`** function, the N. If N is greater than the number of time points in the time series, zeros are added before the Fourier transform is computed (if N is less than the number of time points in the data, the time series is truncated).

```
n=50; x=(linspace(0,1,n)>.5)+0;
zeropadfactor = 2;
subplot(211), plot(x)
set(gca,'ylim',[-.1 1.1])
xlabel('Time (a.u.)')
X1=fft(x,n)/n;
hz1=linspace(0,n/2,floor(n/2)+1);

subplot(212)
plot(hz1,2*abs(X1(1:length(hz1))))
X2=fft(x,zeropadfactor*n)/n;
hz2=linspace(0,n/2,floor(zeropadfactor*n/2)+1);
hold on,plot(hz2,2*abs(X2(1:length(hz2))),'r')
set(gca,'xlim',[0 20])
xlabel('Frequency (a.u.)')
```

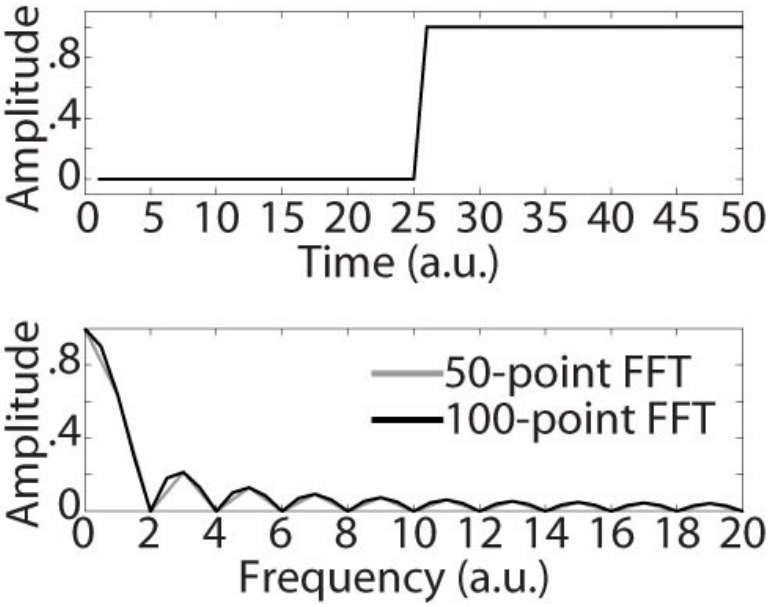

Figure 4.14 | Same FFT, different N.

In both cases above, the frequencies start at zero and end at n/2 (the Nyquist frequency). Thus, the range of frequencies does not change; only the number of frequency points (that is, the frequency resolution) between zero (DC) and the Nyquist changes.

The maximum frequency resulting from a Fourier transform is always one-half of the sampling rate. In the code above this was specified as the number of points, but in real applications, make sure the Nyquist is defined as half the sampling rate, not half the number of time points. This is an easy mistake that can have a significant negative impact when interpreting the results.

When scaling the result of the Fourier transform, make sure to divide by the number of original data points, not the number of data points including the zero padding. Dividing by N including padded zeros will dampen the amplitude of the result of the Fourier analysis.

This is explored further in the exercises at the end of this chapter.

There are three primary reasons to zero-pad a time series before computing its Fourier transform. The first is during convolution via frequency-domain multiplication. In this case, the frequency spectra of two signals are multiplied frequency-by-frequency; if the two signals have different lengths, one must be zero-padded to be the same length as the other. The reason why this is the case will be explained in Chapter 6.

A second reason to zero-pad a time series is to obtain a specific frequency. Because the frequency resolution of a Fourier transform is determined by the number of time points, it is possible that a frequency important for the analysis cannot be extracted from the time series. Zero-padding, by arbitrarily increasing the length of the data, can be used to obtain an arbitrary frequency (as long as that frequency is between 0 and the Nyquist frequency). Note that zero-padding does not increase the frequency *precision* of the Fourier transform, only the frequency *resolution*. To increase both the precision and the resolution, it is necessary to increase the amount of data in the time series, rather than adding zeros.

Finally, a third reason to zero-pad a time series is to speed up the processing time of the fast-Fourier-transform (FFT). Most FFT algorithms are maximally efficient if the input time series has a length corresponding to a power of two (that is, 2 to the power of a positive integer). An FFT will be faster if the time series is 256 points long compared to if it is 250 points long. For short time series, the difference in computation time is negligible. However, if the time series contains thousands, millions, or more time points, zero-padding to obtain a power-of-2 length can save minutes to hours of computation time.

To automatically determine the next power-of-2 for a time series of length **`N`**, use the following Matlab expression: **`2^nexpow2(N)`**.

4.10 | Aliasing and subsampling

If the sampling rate is too low relative to the speed of the changes in the time series, aliasing may occur. The figure below illustrates how aliasing occurs when a “continuous” (here simulated as a high sampling-rate sine wave) signal is subsampled.

```
srate=1000; t=0:1/srate:1;
f = 30; % Hz
srates = [15 20 50 200]; % Hz
% "continuous" sine wave
d = sin(2*pi*f*t);

for i=1:4
  subplot(2,2,i)
  plot(t,d), hold on
  samples = round(1:1000/srates(i):length(t));
  plot(t(samples),d(samples),'r-','linew',2)
end
```

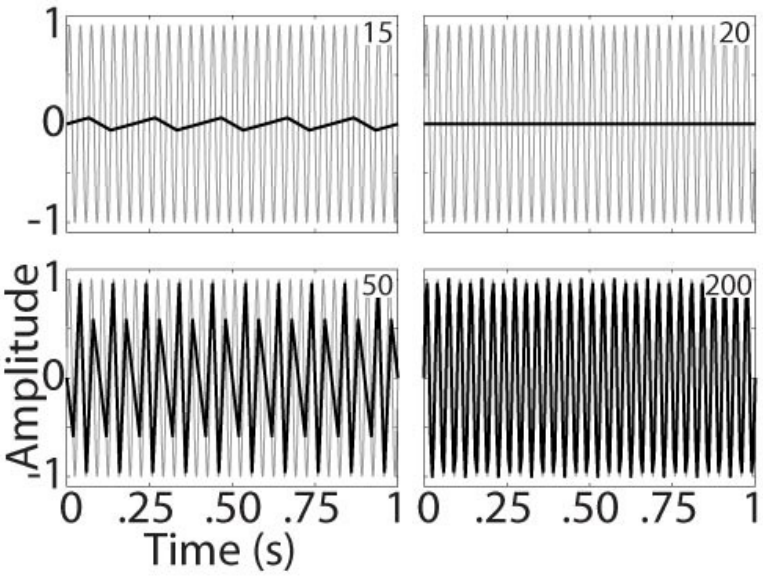

Figure 4.15 | Subsampling produces aliasing, which can lead to improper results. In each panel, the gray line is the "continuous" 30-Hz signal and the black line is the result of sampling. The number in the upper right of each plot refers to the sampling rate in Hz of the black line.

To avoid aliasing, the sampling rate must be at least twice as high as the fastest frequency in the time series (the Nyquist cut-off). To avoid subsampling, it is best to sample 5-10 times higher than the fastest frequency in the signal. This will ensure a high signal-to-noise ratio estimation of the high-frequency activity. Thus, if the fastest frequency is 200 Hz, sampling the time series at 400 Hz is the minimum requirement, and sampling at 2000 Hz is preferable.

4.11 | Repeated measurements

In many applications including scientific experiments, data contain noise. There are several strategies to separate signal from noise, and the optimal strategy depends on the source and the characteristics of the noise. Some sources of noise can easily be filtered out; an example is electrical line noise, which can be removed by applying a band-stop filter at 50 or 60 Hz.

If noise is unpredictable or broadband, one of the easiest and most robust techniques for attenuating noise while preserving signal is to take repeated measurements of the same system. The idea is that the recorded data contain both signal and noise; the signal is similar across repeated measurements while the noise is random on each measurement. Thus, averaging several measurements together will cancel out some of the random noise while preserving the signal.

In most cases, it is advisable to compute the frequency representation of each measurement separately, and then average the spectral results over measurements. This is preferable because minor phase differences across repeated measurements will cause a loss of signal in time-domain averaging, but will not affect the time-frequency power results. This concept is explored further in Chapter 6.

In the code below, 40 measurements of the same system (each measurement is called a trial) will be generated.

```
nTrials=40; srate=1000;
t=0:1/srate:5; n=length(t);
a=[2 3 4 2]; f=[1 3 6 12];
data=zeros(size(t));
for i=1:length(a)
  data = data + a(i)*sin(2*pi*f(i)*t);
end
% create trials with noise
dataWnoise = bsxfun(@plus,data, ...
                30*randn(nTrials,n));

hz = linspace(0,srate/2,floor(n/2)+1);
dataPow = zeros(nTrials,length(hz));
hanwin = .5*(1-cos(2*pi*linspace(0,1,n)));
for triali=1:nTrials
  temp = fft(hanwin.*dataWnoise(triali,:))/n;
  dataPow(triali,:) = 2*abs(temp(1:length(hz)));
end
subplot(211), plot(t,mean(dataWnoise))
subplot(212), plot(hz,dataPow), hold on
plot(hz,mean(dataPow),'k','linewidth',5)
set(gca,'xlim',[0 20])
```

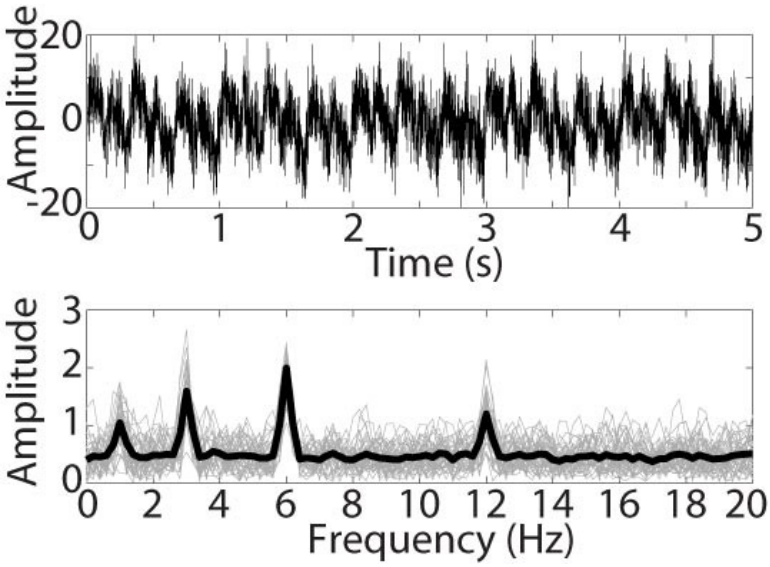

Figure 4.16 | Averaging improves results. Each gray line is a trial, and the thick black line is the average over all trials.

This example also illustrates an advantage of frequency-domain analyses over simple time-domain averaging, particularly in the presence of noise. Indeed, the sine wave components of the original time series are only somewhat visible in the trial-averaged plot, but are clearly identifiable in the frequency-domain plot.

4.12 | The Fourier transform and signal-to-noise ratio

Signal-to-noise ratio (SNR) is an indicator of the quality of the data. Ideally, the signal should be large relative to the noise. In practice, it is often difficult to know what is signal and what is noise. In these cases, the SNR must be estimated from the data. One method to

estimate SNR when there are repeated measurements is to divide the mean power by the standard deviation power, separately for each frequency. It is sometimes helpful to remove the power offset by detrending the power spectrum before computing SNR.

```
snr = mean(detrend(dataPow')')./ ...
      std(detrend(dataPow')');
plot(hz,snr)
set(gca,'xlim',[0 20])
```

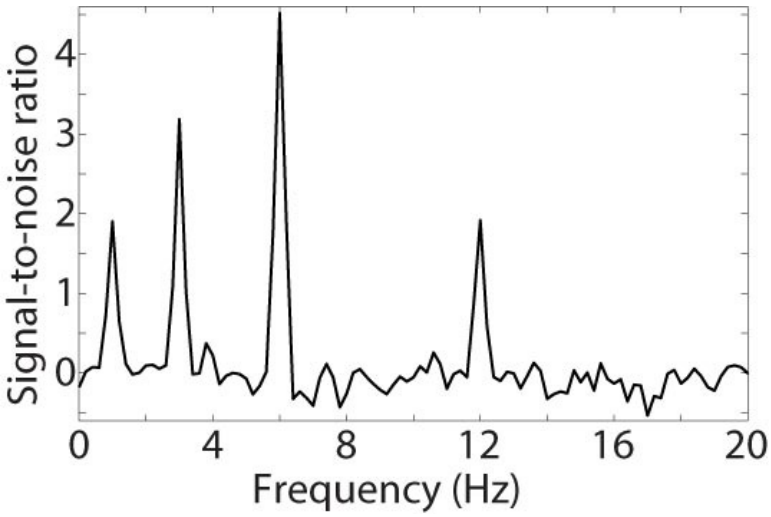

Figure 4.17 | Empirical SNR (mean divided by standard deviation).

4.13 | The multitaper method

When the time series data are tapered before computing the Fourier transform, only one taper is used (here the Hann taper). The multitaper method is an extension of the Fourier transform, in which the Fourier transform is computed several times, each time tapering the data using a different taper. The tapers are taken from the "Slepian sequence," in which each taper is orthogonal to each other taper.

The multitaper method may be useful in situations of low SNR. However, it estimates spectral peaks that are wider than the frequencies present in the original time series (this is called spectral leakage or frequency smearing). Thus, if isolating closely spaced frequencies is important, the multitaper method may not be a preferable option.

The code below will perform a multitaper analysis. It requires the Matlab Signal Processing toolbox to compute the tapers (function **`dpss`**, or discrete prolate spheroidal sequences, also called Slepian sequences). This function is not implemented in Octave as of summer 2014.

```
srate=1000; t=0:1/srate:5; n=length(t);

% how much noise to add
noisefactor = 20;

% create multi-frequency signal
a=[2 3 4 2]; f=[1 3 6 12];
data=zeros(size(t));
```

```
for i=1:length(a)
  data = data + a(i)*sin(2*pi*f(i)*t);
end
data=data+noisefactor*randn(size(data));

% define Slepian tapers
tapers = dpss(n,3)';

% initialize multitaper power matrix
mtPow = zeros(floor(n/2)+1,1);
hz = linspace(0,srate/2,floor(n/2)+1);

% loop through tapers
for tapi = 1:size(tapers,1)-1
  % scale taper
  temptaper = tapers(tapi,:)./ ...
              max(tapers(tapi,:));
  % FFT and add to power
  x = abs(fft(data.*temptaper)/n).^2;
  mtPow = mtPow + x(1:length(hz))';
end
% divide by the n tapers to average
mtPow = mtPow./tapi;

% 'normal' power spectra
hann = .5*(1-cos(2*pi*(1:n)/(n-1)));
x = abs(fft(data.*hann)/n).^2;
regPow = x(1:length(hz));

% plot
plot(hz,mtPow,'.-'), hold on
plot(hz,regPow,'r.-')
```

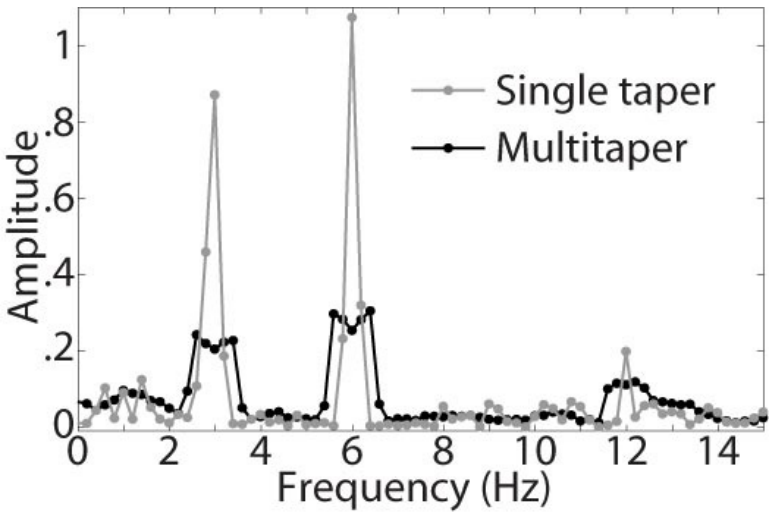

Figure 4.18 | Single and multitaper.

4.14 | The inverse Fourier transform, slow and fast

The Fourier transform contains all of the information of the time series from which it was computed. This loss-less representation can be demonstrated by applying the inverse

Fourier transform to obtain the original time series. The inverse Fourier transform involves summing a series of sine waves that are scaled by the frequency-corresponding Fourier coefficients.

```
xTime = randn(20,1);
xFreq = fft(xTime)/length(xTime);
t = (0:length(xTime)-1)'/length(xTime);

recon_data = zeros(size(xTime));
for fi=1:length(xTime)
  sine_wave = xFreq(fi)*exp(1i*2*pi*(fi-1).*t);
  recon_data = recon_data + sine_wave;
end
plot(xTime,'.-'), hold on
plot(real(recon_data),'ro')
```

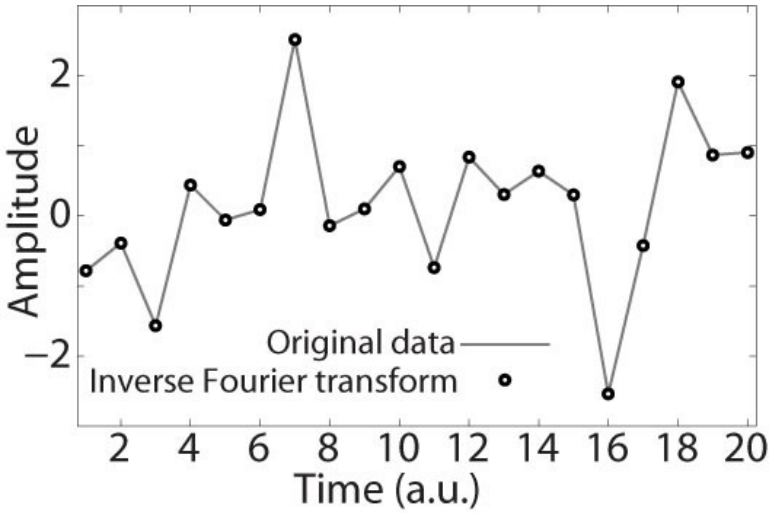

Figure 4.19 | Back to the beginning.

In practice, it is faster and simpler to use the fast inverse Fourier transform, which is equivalent to the implementation above.

```
recon_data = ifft(xFreq);
```

4.15 | Exercises

1) One method of scrambling a signal is to shuffle the phases of the Fourier coefficients and then reconstruct the time-domain data. In this exercise, generate a linear chirp and compute its inverse after shuffling the phases (do not alter the power values). Plot the time domain signals and their spectra. Is the scrambled time series recognizable as having been a chirp?

```
% first, the basics:
srate= 1000;
t    = 0:1/srate:6;
N    = length(t);
f    = [1 5]; % in hz
```

```
ff = linspace(f(1),f(2)*mean(f)/f(2),N);
data = sin(2*pi.*ff.*t);

% second, compute Fourier transform
dataX = fft(data);

% third, shuffle phases (here, shifted by 10)
phases = angle([dataX(10:end) dataX(1:9)]);
shuffdata = abs(dataX).*exp(1i*phases);

% fourth, reconstruct the signal
newdata = ifft(shuffdata);

% fifth, plot the results
subplot(211)
plot(t,data), hold on
plot(t,real(newdata),'r')

subplot(212)
hz = linspace(0,srate/2,floor(N/2)+1);
plot(hz,2*abs(dataX(1:length(hz))/N),'o'), hold on
plot(hz,2*abs(shuffdata(1:length(hz))/N),'r')
set(gca,'xlim',[0 20])
```

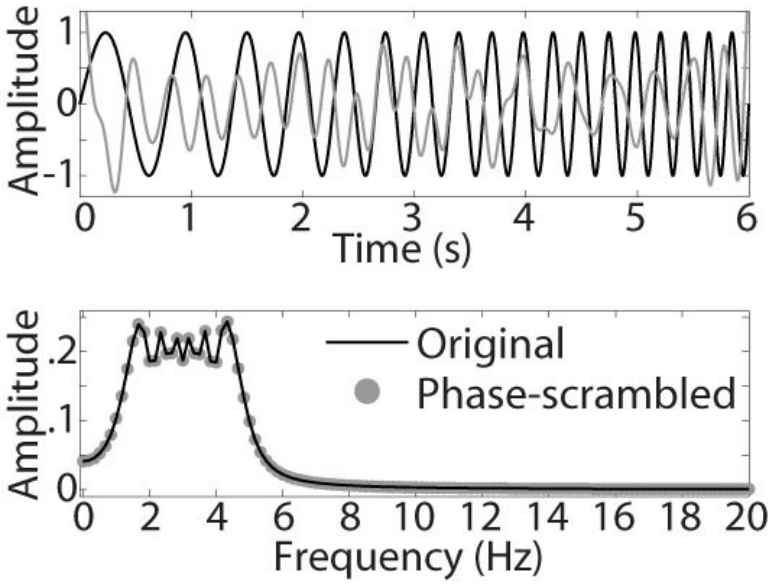

Figure 4.20 | Same power, shuffled phases.

By the way, the above example also illustrates a simple method of decryption: If a time-domain signal is sent with shuffled phases, only a receiver with the correct reverse-distortion algorithm (in this case, the algorithm is simply to shift the phases 10 points forwards) will be able to transform the time series to its original signal.

2) Reproduce Figure 4.16 (Chapter 4.11) four additional times, setting the amplitude of the random noise to be 10 or 100, and the number of trials to be 100 or 1000. Are the individual peak frequencies still clearly visible in the spectral plot? What does your answer indicate about collecting more data with a lot of noise, vs. less data with less noise?

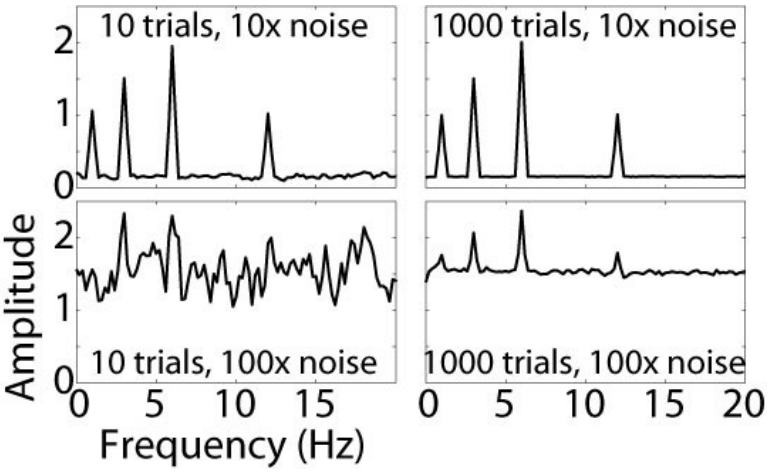

Figure 4.21 | Trials and noise.

3) Generate a 2-second sine wave of 10 Hz. Compute its Fourier transform and plot the resulting power spectrum (do not apply a taper). Next, re-compute the Fourier transform with 2 additional seconds of zero-padded data. Instead of scaling the Fourier coefficients by the number of time points in the original sine wave, however, scale the coefficients by the number of time points including the zero padding (thus, the number of time points in 4 seconds). What do you notice about the amplitude between the two analyses? What happens to the amplitude if there are 4 seconds of zero padding, or 1 second of zero padding? Explain why these differences occur.

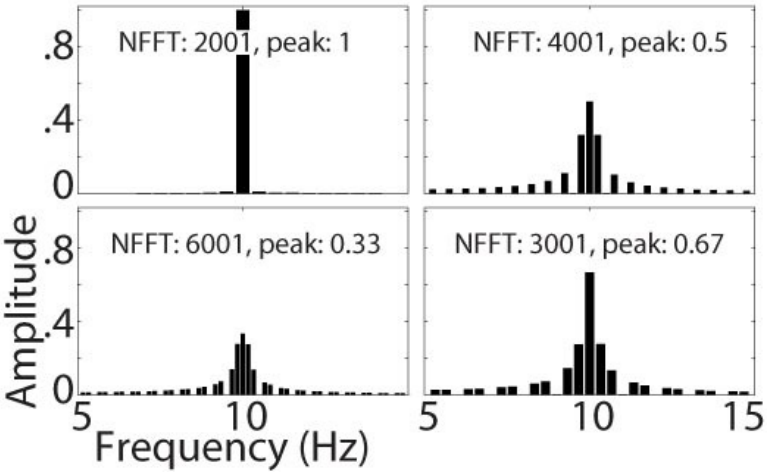

Figure 4.22 | FFT amplitude with different N scalings.

Chapter 5: The short-time Fourier transform

Chapter 4 illustrated that although the Fourier transform can perfectly represent a time series using sine waves of various frequencies, amplitudes, and phases, the time-varying changes in the frequency structure are difficult to visualize in the spectral plot. The next four chapters will introduce various methods to visualize and quantify time-varying changes in the frequency characteristics of a time series.

Many time-frequency analyses result in some loss of signal, which is to say, it is not possible to reconstruct the original time series perfectly based on the results of a time-frequency analysis. This contrasts with the Fourier transform, which contains all the information in a time series and from which the original time series can be perfectly reconstructed. Often, this is not important because the gains in qualitative and quantitative understanding of the characteristics of the time series are more important than being able to reconstruct the time series perfectly. That is, some signal may be lost, but information is gained.

5.1 | The short-time Fourier transform

The idea is simple, intuitive, and effective: Compute the Fourier transform in short, successive time windows, rather than once over the entire time series. It is useful to have the time windows overlap to improve visibility of the plot. This is demonstrated below using a time series comprising time-varying changes in frequency. The time series is similar to that used in Figure 4.7.

```
srate=1000; t=0:1/srate:5; n=length(t);
f = [30 3 6 12];
tchunks = round(linspace(1,n,length(f)+1));
data = 0;
for i=1:length(f)
  data = cat(2,data,...
    sin(2*pi*f(i)*t(tchunks(i):tchunks(i+1)-1) ));
end
subplot(211), plot(t,data)
```

Now that a time series of changes in frequency is created, the FFT will be computed as shown in Chapter 4, except that it is here computed in successive windows of specified width. Before starting the analysis, the time window width and center time points for computing the Fourier transform must be specified.

```
fftWidth_ms = 1000;
fftWidth=round(fftWidth_ms/(1000/srate)/2);
Ntimesteps = 10; % number of time widths
centimes = round(linspace(fftWidth+1, ...
            n-fftWidth,Ntimesteps));
```

After being converted from time in milliseconds (ms) to time in indices (in this case, they happen to be the same because the sampling rate is 1000 Hz, but adding the conversion is good practice for situations with non-1000 Hz sampling), the variable **`fftWidth`** is reduced by half. This is because the Fourier transform will be computed surrounding each center time point—in other words, **`fftWidth`**/2 before, and **`fftWidth`**/2 after, each center time point.

If the center time points are closer together than the windows, for example, if the center time points are spaced 100 ms apart and the windows are 100 ms wide, there will be overlap between successive windows. As mentioned above, time window overlap helps to smooth the time-frequency result and thus make the time-varying changes in frequency characteristics easier to see.

The variable **`centimes`** (center time points) is computed automatically based on a desired number of center time points, which above was set to ten. Note that the data in each time window should be tapered before computing the Fourier transform. Tapering attenuates edge artifacts, as discussed in Chapter 4.

```
hz=linspace(0,srate/2,fftWidth-1);
tf=zeros(length(hz),length(centimes));
hwin = .5*(1-cos(2*pi*(1:fftWidth*2)/ ...
          (fftWidth*2-1))); % Hann taper
for ti=1:length(centimes)
  x = fft(hwin.*data(centimes(ti)-fftWidth:...
        centimes(ti)+fftWidth-1))/fftWidth*2;
  tf(:,ti) = 2*abs(x(1:length(hz)));
end
subplot(212)
contourf(t(centimes),hz,tf,1)
set(gca,'ylim',[0 50],'clim',[0 1])
xlabel('Time (s)'), ylabel('Frequency (Hz)')
```

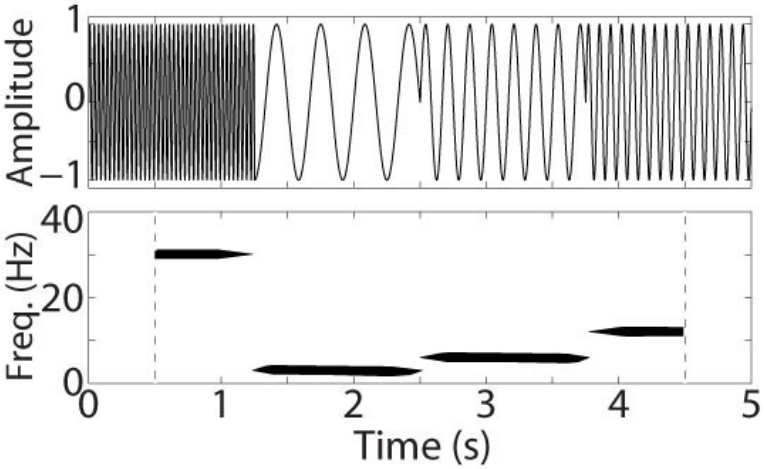

Figure 5.1 | A time varying signal and its time-frequency representation.

The lower panel in Figure 5.1 is called a “time-frequency plot.” Because the time-frequency plot shows amplitude (the square root of power), it is also called a “time-frequency power(/amplitude) plot.”

Unlike the spectral plots shown previously in this book, frequency here is on the y-axis,

with lower frequencies towards the bottom and higher frequencies towards the top. Time is plotted on the x-axis, the same as in the time-domain plot in the upper panel.

Visually, the time-frequency plot seems to match the time series fairly well, including the onset/offset times and frequencies of the constituent sine waves. And from this plot, it is clear that the frequency structure of the signal changes over time (recall that this was not clear when computing the "static" Fourier transform in Figure 4.7).

One striking difference, however, is that the time-frequency plot cuts off the time series at 0.5 and 4.5 seconds (see vertical dotted lines in Figure 5.1). This occurs because the short-time Fourier transform is computed on data surrounding each center time point, and there is no data before time 0. Thus, the first time point in which it is possible to estimate the frequency characteristics is after the signal has already begun. If there were a pre-/post-signal buffer time of at least 500 ms, it would have been possible to have the time-frequency result start and end with the signal. This was one of the reasons why a buffer period is useful, as described in Chapter 3.8.

If a certain frequency is of *a priori* interest, it can be plotted separately from the other frequencies.

```
freq2plot = dsearchn(hz',6);
plot(t(centimes),tf(freq2plot,:),'-o')
```

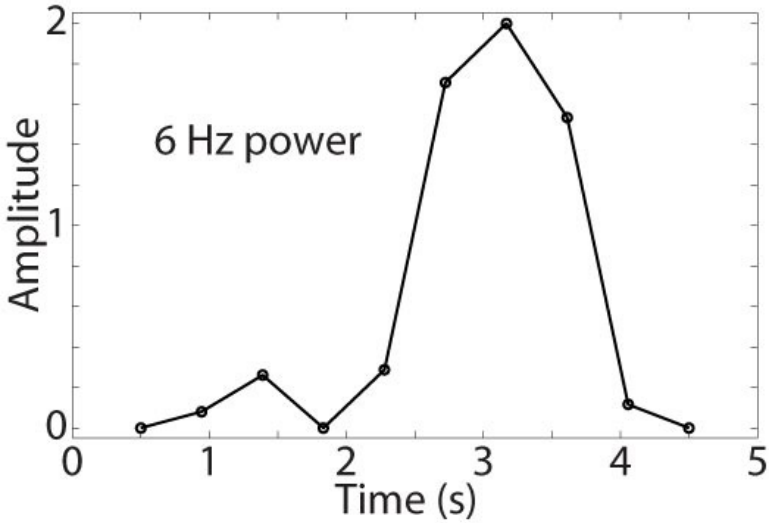

Figure 5.2 | A time slice of frequency.

5.2 | Temporal leakage in the short-time Fourier transform

According to the code at the start of this chapter, the 6 Hz sine wave should be present and with constant amplitude from 2.5 to 3.75 seconds. Thus, there appears to be a mismatch between the time-frequency result shown in Figure 5.2 and the original time series data.

This illustrates temporal leakage (also called temporal smearing), which results from the windowing procedure. Specifically, a window centered at 2.27 seconds will include part of the 6-Hz sine wave, and thus will show some non-zero power at 6 Hz.

Temporal smearing is unavoidable in time-frequency analyses. It can be minimized by taking smaller time windows, although this comes at the expense of decreased frequency

resolution, as explained in the next section.

5.3 | Temporal precision and resolution in the short-time Fourier transform

Recall the distinction in Chapter 3.7 between precision and resolution. For the short-time Fourier transform, a high temporal resolution (that is, a high sampling rate) is useful to extract frequency components from the data. On the other hand, temporal smearing, combined with the fact that changes in frequency characteristics are necessarily slower than the temporal resolution of the data, reduces the temporal precision of the result of the short-time Fourier transform. The temporal precision of the results is determined in part by the characteristics of the time series, and in part by the size of the window (wider windows decrease temporal precision because they average over more time points).

Because the temporal precision of a time-frequency analysis is decreased, it is reasonable to reduce the temporal resolution of the result of the time-frequency analysis. For example, in Figures 5.1 and 5.2, the temporal resolution of the short-time Fourier transform is 2 Hz (10 time points in 5 seconds).

Although some information is lost, the main features of the results remain. You can investigate the effect of the temporal resolution of the short-time Fourier transform on the quality of the result, by changing values for the **`Ntimesteps`** parameter. For example, plot the 6-Hz sine component with 10 vs. 100 time steps.

In other words, the temporal resolution of the result of the short-time FFT is determined by the number of time steps (here, the variable **`centimes`**), while the temporal precision is determined in part by the window width and in part by the speed with which the frequency characteristics change relative to each frequency of the Fourier transform result.

Temporally down-sampling the results is not necessary, but is often useful: It reduces computation time and reduces the amount of data that must be stored in a buffer or exported to a file for subsequent analyses.

The amount of down-sampling that is appropriate depends on the characteristics of the data and cannot be predicted before starting the analysis. In practice, it is best to inspect the results and determine how much down-sampling is appropriate for each data set.

5.4 | Frequency precision and resolution in the short-time Fourier transform

The frequency resolution of a Fourier transform depends entirely on the number of time points in the data (see Chapter 4.3). For this reason, the frequency resolution of the short-time Fourier transform decreases when the size of the time window decreases.

As with temporal down-sampling, it may not be necessary to store the results of all frequencies in the time-frequency analysis. If activity in a certain range of frequencies can be expected, it may be advantageous to save only some frequencies. This is shown in the code below.

```
% keep only 30 evenly spaced frequencies
freqs2keep = linspace(0,hz(end),30);
```

```
freqsidx = dsearchn(hz',freqs2keep');
hanwin = .5*(1-cos(2*pi*(1:fftWidth*2) ...
            /(fftWidth*2-1)));

tf=zeros(length(freqs2keep),length(centimes));
for ti=1:length(centimes)
  temp = data(centimes(ti)-fftWidth…
         :centimes(ti)+fftWidth-1);
  x = fft(hanwin.*temp)/fftWidth*2;
  tf(:,ti) = 2*abs(x(freqsidx));
end
```

Similar to temporal down-sampling, the number and range of requested frequencies depends on the characteristics of the data, and thus is best determined for each dataset individually.

For example, after inspecting the full frequency range of the data shown in Figure 5.1, it seems that the **`freqs2keep`** variable can be set to **`linspace(1,40,30)`**, in other words, 30 frequencies between 1 Hz and 40 Hz will be kept in the results. It is generally a good idea to have some frequencies below, and some frequencies above, the main results; this allows confirmation of the frequency range of the signal.

Note that the requested frequencies are not exact; they are approximations. For example, when extracting frequencies between 1 Hz and 40 Hz, the second requested frequency is 2.3448 Hz. This exact frequency cannot be extracted from the data due to limited frequency resolution (because of the limited time window). Instead, this requested frequency is estimated as 2.008 Hz, the closest frequency extracted from the data to the requested frequency. As discussed earlier, longer time windows will provide better frequency resolution, but at the expense of reduced temporal precision of the results.

5.5 | Adjusting the window size for different frequency ranges

A trade-off emerges when selecting the window size: It is not possible to maximize both temporal precision and frequency resolution simultaneously. This is the Heisenberg uncertainty principle applied to time-frequency analysis.

One solution to this trade-off is to change the size of the time window as frequencies increase. Thus, at lower frequencies, frequency resolution is maximized at the expense of decreased temporal precision, while at higher frequencies, temporal precision is maximized at the expense of decreased frequency resolution. This approach is often acceptable because lower-frequency activity usually changes more slowly compared to higher-frequency activity. The general idea is illustrated in Figure 5.3.

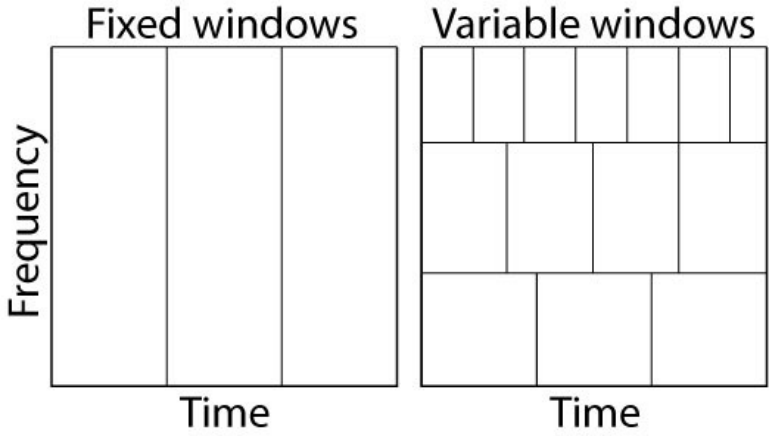

Figure 5.3 | Constant vs. variable time-frequency windows. Variable window sizes allow for a flexible trade-off between temporal precision and frequency resolution, although this comes at the expense of an increase in computation time. Variable window sizes are often preferred in time-frequency analyses.

The code below illustrates the procedure of generating an arbitrary number of time-frequency window sizes.

```
NtimeWidths = 5;
fftWidth_ms = linspace(1300,500,NtimeWidths);
fftWidth=round(fftWidth_ms./(1000/srate)/2);
Ntimesteps = 10; % number of time widths
centimes = round(linspace(max(fftWidth)+1,...
           length(t)-max(fftWidth),Ntimesteps));
```

The time windows become smaller with increasing frequency, thus increasing temporal precision at the expense of frequency resolution.

```
% frequencies to keep, and per bin
f2keep = linspace(1,50,40);
freqsPerBin = ceil((f2keep./max(f2keep)) ...
                *NtimeWidths);

tf=zeros(length(f2keep),length(centimes));
for ti=1:length(centimes)
  for fi=1:NtimeWidths

    % find appropriate frequencies in this bin
    hz=linspace(0,srate/2,fftWidth(fi)-1);
    freqsidx = dsearchn(hz',...
          f2keep(freqsPerBin==fi)');

    % compute windowed FFT
    hanwin=.5*(1-cos(2*pi*(1:fftWidth(fi)*2)...
              /(fftWidth(fi)*2-1)));
    temp = data(centimes(ti)-fftWidth(fi)...
              :centimes(ti)+fftWidth(fi)-1);
    x = fft(hanwin.*temp)/fftWidth(fi)*2;

    % put data in TF matrix
    tf(freqsPerBin==fi,ti)=2*abs(x(freqsidx));
```

```
  end
end
```

Changing the time window for different frequency ranges becomes time-consuming because of the increased number of FFTs and associated calculations that must be performed. In this sense, complex wavelet convolution (next chapter) is advantageous over the short-time Fourier transform because it can achieve similar results using fewer computations.

5.6 | Other short-time Fourier transform possibilities

This chapter does not provide an exhaustive list of analysis possibilities using the short-time Fourier transform. Additional possibilities, including those discussed in Chapter 4 such as signal-to-noise ratio and the multitaper method, can also be applied to the short-time Fourier transform.

If repeated measurements of the system are taken ("trials" as discussed in Chapter 4.11), the short-time FFT should be computed separately for each trial, and then the resulting time-frequency plots should be averaged together. This is preferable over averaging first in the time domain and then performing one short-time Fourier transform, because minor temporal jitters or phase differences across trials can cause time-domain cancellation, thus decreasing the accuracy of the time-frequency analysis. This phenomenon will be illustrated in exercise 1 of Chapter 6.

5.7 | Wigner-Ville distribution

The Wigner-Ville method of time-frequency analysis is not formally a short-time Fourier transform-based technique, but it is presented in this chapter because it involves the Fourier transform and it is not presented in enough depth to warrant its own chapter.

The Wigner-Ville method involves looping through each time point of the time series. At each time point, the data from N points forward in time are multiplied by the data from N points backwards in time, creating an autocorrelation distribution matrix. A Fourier transform is then applied to this matrix. Because the number of points multiplied at each step increases from 1 to the total number of time points (N), each row of the Fourier transform corresponds to a frequency of the data.

```
% for signal 'data' of length 'n'
for ti=1:n
  % determine points to use
  tmax = min([ti-1,n-ti,round(n/2)-1]);
  pnts = -tmax:tmax;
  idx  = rem(n+pnts,n)+1;

  % multiply forward and backward points
  wig(idx,ti) = data(ti+pnts) ...
                .*data(ti-pnts);
end
```

```
% take Fourier transform
wig = 2*abs(fft(wig)/size(wig,1));
```

The Wigner-Ville method has high frequency and temporal precision.

However, as shown below, its two main limitations are that it is strongly affected by noise, and that it produces “cross-terms” for multi-component signals that act as artifacts. There are adjustments to the Wigner-Ville method that can help filter out these cross-terms, although these are not discussed here. The figure below shows results of the Wigner-Ville analysis and the short-time Fourier analysis applied to the same data. The results use the code presented earlier in this chapter, and can be inspected in more detail in the online Matlab code.

In short, although the Wigner-Ville method has some advantages (it was initially developed to study quantum mechanics), it is generally not superior to other time-frequency methods presented in the book in the presence of noise and/or multifrequency signals.

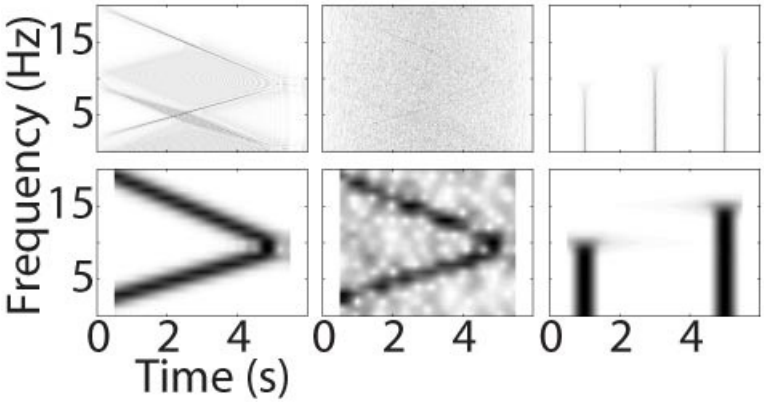

Figure 5.4 | Comparison of result of Wigner-Ville (top row) and short-time Fourier transform (bottom row) on three simulated time series: A double-chirp without noise (left), the same chirp but with noise (noise factor=3) (middle), and two sincs (right). See online Matlab code for implementation and additional notes.

5.8 | Exercises

1) Generate a 5-second chirp that spans from 1 Hz to 40 Hz, with a 100 Hz sampling rate. Perform a time-frequency analysis using the short-time Fourier transform with four different settings for the window width parameter: 50, 100, 200, and 500 ms. Plot all four time-frequency power plots using the same color scale. Clearly, the analyses with window widths of <100 ms do not accurately represent the signal. Why is this the case, and what is the important lesson about the short-time FFT from this exercise?

```
% first, create chirp
srate = 100;
t     = 0:1/srate:5;
n     = length(t);
f     = [1 40];
ff    = linspace(f(1),f(2)*mean(f)/f(2),n);
signal= sin(2*pi.*ff.*t);
```

```
% list of window widths (in ms)
FFTwidths  = [50 100 200 500];
Ntimesteps = 20; % number of time widths

for widi=1:length(FFTwidths)
  % define window width, convert to time steps
  fftWidth_ms = FFTwidths(widi);
  fftWidth = round(fftWidth_ms/(1000/srate)/2);
  centimes = round(linspace(fftWidth+1,...
                n-fftWidth,Ntimesteps));

  % initialize parameters
  hz = linspace(0,srate/2,fftWidth-1);
  tf = zeros(length(hz),length(centimes));
  hanwin = .5*(1-cos(2*pi*(1:fftWidth*2)/ ...
           (fftWidth*2-1)));

  % loop through center points and compute FFT
  for ti=1:length(centimes)
    temp = signal(centimes(ti)-fftWidth: ...
            centimes(ti)+fftWidth-1);
    x = fft(hanwin.*temp)/fftWidth*2;
    tf(:,ti) = 2*abs(x(1:length(hz)));
  end
  % and plot
  subplot(2,2,widi)
  contourf(t(centimes),hz,tf,1)
end
```

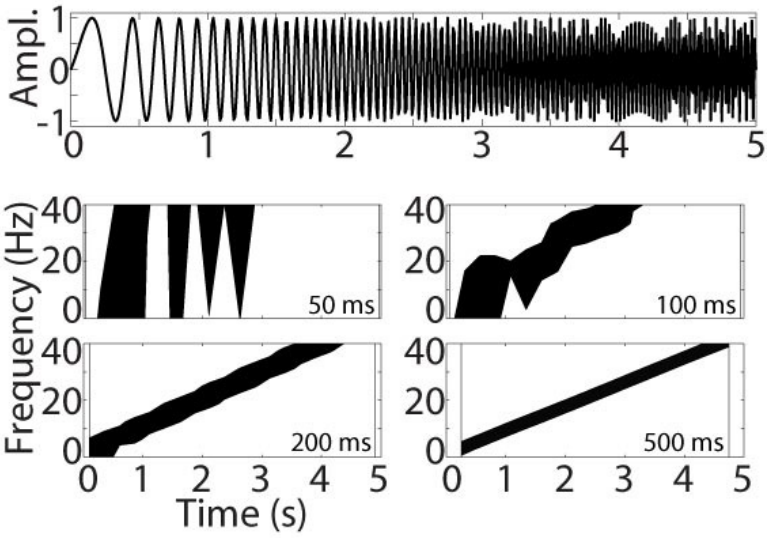

Figure 5.5 | FFT window widths must be carefully selected (lower-right number in each plot indicates time width).

To understand why the results with 50 ms and 100 ms window width so poorly reflected the signal characteristics, consider that when each FFT is based on 50 ms of data, the lowest-frequency signal that would have at least one full cycle within 50 ms is 40 Hz (1000/(50/2)). For a window width of 100 ms, the lowest frequency that can be reconstructed is 20 Hz.

In other words, the frequency precision of the signal is greater than the frequency resolution of the analysis.

The important lesson here is that there is a practical lower-limit to how short the time windows can be in a short-time Fourier transform analysis. This lower limit is set by the lowest-frequency component that can be expected in the signal.

2) Re-compute Figure 5.1, but adapt the time-frequency analysis so that it captures the dynamics in the first and last 500 ms.

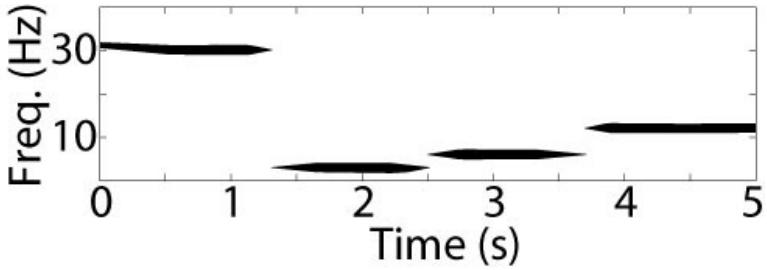

Figure 5.6 | Same as Figure 5.1 but this time with time-frequency estimates before .5 seconds and after 4.5 seconds.

3) Generate a 9-second time series with 4.5 seconds at one frequency and 4.5 seconds at another frequency. Compute the time-frequency representation of the time series, and also the 'static' Fourier transform of the entire time series as shown in Chapter 4. From the time-frequency representation, sum the amplitude values over all time points to obtain a power spectrum plot like that of the 'static' Fourier transform. How do the amplitudes compare between the two analyses?

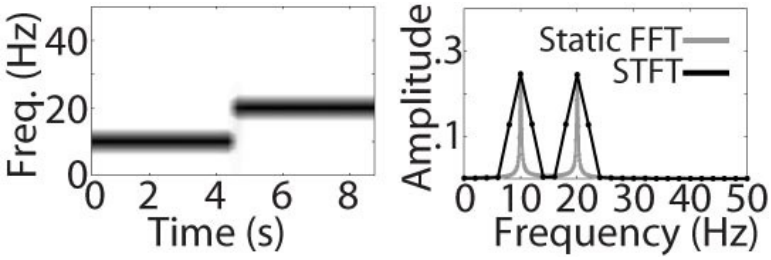

Figure 5.7 | Static (FFT) and short-time (STFT) Fourier transforms. The STFT was computed using a constant 500 ms time window.

Chapter 6: Morlet wavelet convolution

Morlet wavelet convolution is a time-frequency analysis method that produces similar results as the short-time Fourier transform, and has some advantages including decreased computation time, increased adaptability concerning the time-frequency resolution trade-off as a function of frequency, and increased temporal resolution.

6.1 Wavelets and Morlet wavelets

A wavelet is a short, wave-like time series that starts and ends at or very close to zero. There are many types of wavelets that are useful for specific applications. The figure below illustrates three different wavelets.

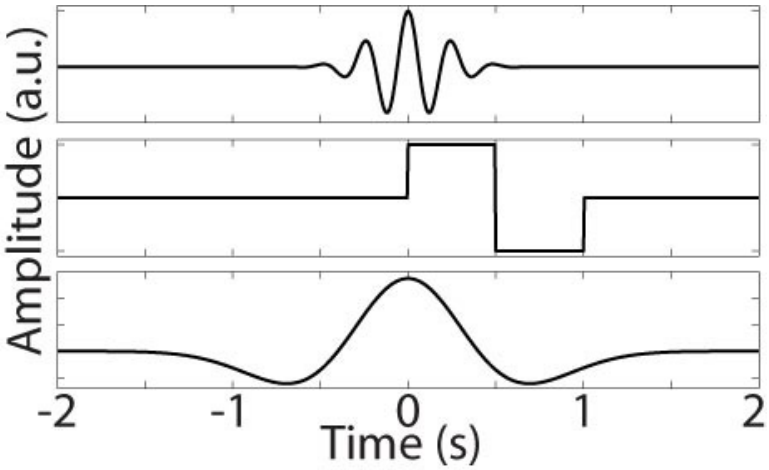

Figure 6.1 | There are many wavelets in the world, including the Morlet (top), the Haar (middle), and the Mexican hat (bottom).

This chapter will focus on the Morlet wavelet, which is created by tapering a sine wave with a Gaussian. Morlet wavelets are often used in time-frequency analyses because they are temporally symmetric, allow flexible control of the trade-off between temporal and frequency precisions, and do not introduce edge artifacts because they have a Gaussian shape in the frequency domain. Nonetheless, the procedures and code in this chapter could be applied to any wavelet, not only the Morlet wavelet.

```
t=-1:.01:1; f=10;
sinewave = cos(2*pi*f*t);
w = 2*( 5/(2*pi*f) )^2;
gaussian = exp( (-t.^2)/w );
mwavelet = sinewave .* gaussian;
plot(t,mwavelet);
```

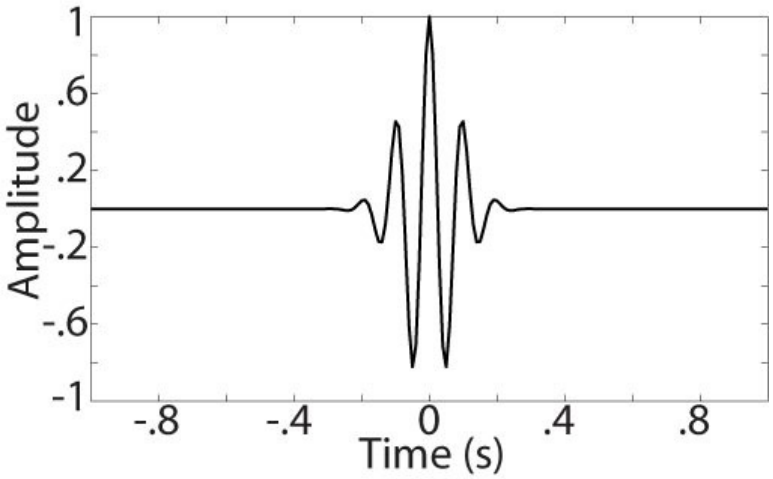

Figure 6.2 | Morlet wavelet.

The Gaussian is created using a width parameter w. This parameter (more specifically, the variable that above is hard-coded as "5") defines the time-frequency precision trade-off, and will be discussed in more detail in Chapter 6.7.

Because power and phase will need to be estimated, a *complex* Morlet wavelet is necessary, similar to how a complex sine wave is necessary for the Fourier transform, as was described in Chapter 4. A complex Morlet wavelet has three dimensions: real, imaginary, and time. The real part corresponds to a cosine wave and the imaginary part corresponds to a sine wave.

```
t=-1:.01:1; f=10;
csw = exp(1i*2*pi*f*t);
mwavelet = csw .* gaussian;
plot3(t,real(mwavelet),imag(mwavelet),'-o');
xlabel('Time (s)'), ylabel('real part'),
zlabel('imaginary part')
rotate3d % unnecessary in Octave
```

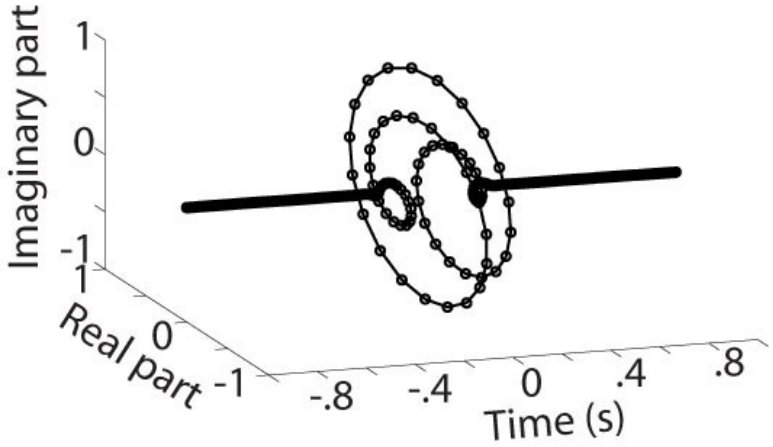

Figure 6.3 | Complex Morlet wavelet in 3D.

6.2 Convolution in the time domain

The purpose of convolution is to compare two time series over time (convolution can also be computed over space, but this book will focus on time).

The basis of convolution is the dot-product—the sum over point-wise multiplications between two vectors—which was introduced in Chapter 4 as the basis of the Fourier transform.

Convolution is achieved by repeatedly computing dot-products between the kernel (which is flipped backwards) and the time series, each time shifting the kernel over by one unit of time (see Figure 6.4). The time series of dot-products is the result of convolution.

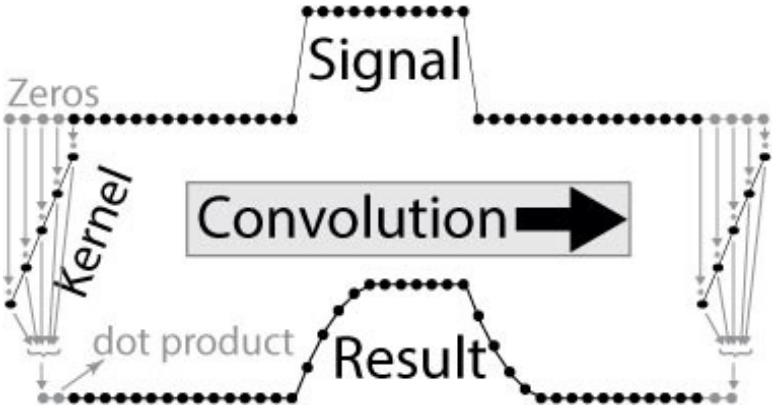

Figure 6.4 | Overview of convolution in the time domain between a time series (top) and a kernel (slanted line). Each value of the result is the dot-product between the kernel and the time series. The result (bottom) reflects the "smearing" of the kernel over the time series. Grey dots indicate padded zeros.

Convolution begins with the kernel aligned to the left-most point of the time series, and convolution ends with the kernel aligned to the right-most point of the time series.

This has two implications. First, the time series must be zero-padded on either end to accommodate the extra multiplications. Second, the length of the result of convolution will be K+S-1, where K and S are the lengths of the kernel and the time series. After convolution is finished, the result must be cut at the beginning and at the end by one-half of the length of the wavelet. The code below performs convolution between a chirp (linear increase in frequency from 2 to 8 Hz over 6 seconds) and a 6 Hz Morlet wavelet.

```
srate=1000; f=[2 8];
t=0:1/srate:6; n=length(t);
chirpTS = sin(2*pi.* ...
      linspace(f(1),f(2)*mean(f)/f(2),n).*t);

% create complex Morlet wavelet
wtime = -2:1/srate:2; % wavelet time
w = 2*( 4/(2*pi*5) )^2;
cmw = exp(1i*2*pi*5.*wtime) ...
       .* exp( (-wtime.^2)/w );
% half of the length of the wavelet
halfwavL = floor(length(wtime)/2);

% zero-pad chirp
chirpTSpad = [zeros(1,halfwavL) chirpTS…
              zeros(1,halfwavL)];

% run convolution
convres = zeros(size(chirpTSpad));
```

```
for i=halfwavL+1:length(chirpTS)+halfwavL-1
  % at each time point, compute dot product
  convres(i)=sum(chirpTSpad( ...
          i-halfwavL:i+halfwavL) .* cmw );
end
% cut off edges
convres = convres(halfwavL:end-halfwavL-1);

subplot(211), plot(t,chirpTS)
subplot(212), plot(t,abs(convres))
```

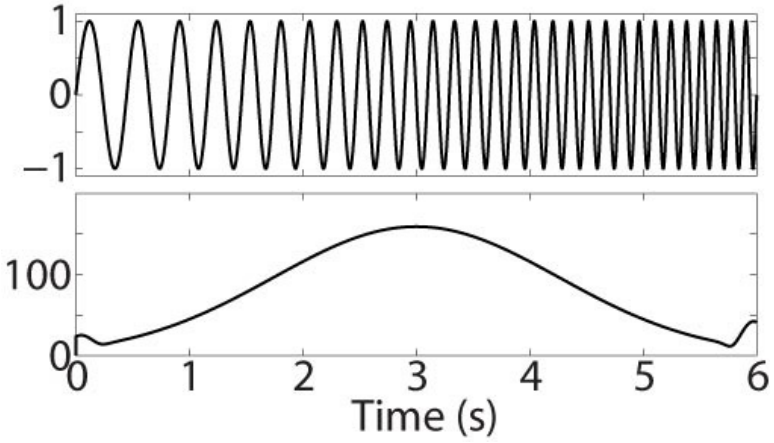

Figure 6.5 | Result of convolution between a 5 Hz wavelet and a 2-8 Hz chirp. Note the difference in amplitude between the original time series and the result of convolution.

The amplitude of the result of convolution between a 5 Hz wavelet and the chirp builds up, peaks at around 3 seconds, and then decays. This is a sensible result, considering that the middle frequency of the 6-second chirp is 5 Hz, that the transition from 2 Hz to 8 Hz is smooth, and that there is temporal leakage in time-frequency analyses.

The amplitude of the signal is 1 while the peak amplitude of the result of convolution is around 150. This is because the amplitude of the Morlet wavelet was not scaled in the example above. This will be discussed in Chapter 6.4.

6.3 The convolution theorem: Fast convolution via frequency-domain multiplication

The convolution theorem states that convolution in the time domain is equivalent to multiplication in the frequency domain (the reverse is also true: convolution in the frequency domain is equivalent to multiplication in the time domain, although this is not useful for time-frequency analyses).

The impact of the convolution theorem for wavelet convolution is that convolution can be performed by computing the Fourier transforms of the kernel and the wavelet, multiplying their frequency spectra, and computing the inverse Fourier transform.

This makes wavelet convolution similar to band-pass filtering as will be described in Chapter 7, except that the shape of the filter is defined by the frequency representation of the wavelet, rather than by an arbitrary (e.g., plateau-shaped) function. A Morlet wavelet has a Gaussian shape in the frequency domain. This is useful for time-frequency analyses

because there are no frequency-domain edge artifacts.

Performing convolution via frequency-domain multiplication makes complex wavelet convolution a useful and fast time-frequency analysis method. The code below performs the same convolution as in the previous section, but using frequency-domain multiplication.

```
Lconv = length(t)+length(wtime)-1;
convres2 = ifft( fft(chirpTS,Lconv).* ...
                 fft(cmw,Lconv) );
convres2 = convres2(halfwavL:end-halfwavL-1);
```

The result above is not plotted because it overlaps perfectly with the lower panel of Figure 6.5. This can be further verified with the Matlab **`conv`** function.

```
convres3 = conv(chirpTS,cmw);
convres3 = convres3(halfwavL:end-halfwavL-1);
```

In Matlab (not Octave), the above two lines can be condensed into one:

```
convres3 = conv(chirpTS,cmw,'same');
```

6.4 Amplitude scaling complex Morlet wavelet convolution results

As seen in Figure 6.5, the amplitude of the result of convolution between a complex Morlet wavelet and a time series did not match the amplitude of the time series. The amplitude of the convolution result is a function of the frequency, the width of the wavelet, and the sampling rate of the data.

To obtain a convolution result in the same amplitude units as the original time series data, the amplitude of the Morlet wavelet must be scaled. This scaling can be done either in the time domain or in the frequency domain. Time-domain scaling is tricky, because there is no simple amplitude scaling function that will work for all frequencies, Gaussian widths, and sampling rates. Fortunately, frequency-domain amplitude scaling is simple and effective. It involves scaling the frequency representation of the wavelet to a maximum of 1.0 before multiplying the spectra of the wavelet and time series together. This is illustrated in the code below.

```
Lconv = length(t)+length(wtime)-1;
cmwX = fft(cmw,Lconv);
cmwX = cmwX./max(cmwX);
convres4 = ifft( fft(chirpTS,Lconv).* cmwX );
convres4 = convres4(halfwavL:end-halfwavL-1);

plot(t,chirpTS), hold on
plot(t,2*abs(convres4),'r')
```

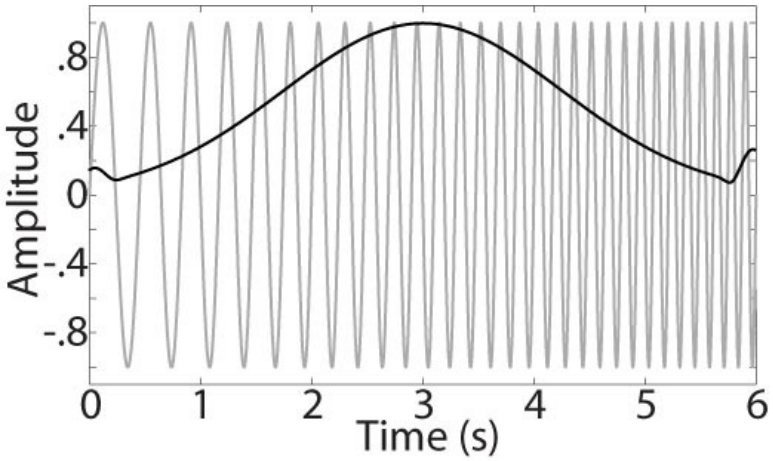

Figure 6.6 | Convolution result with appropriate amplitude scaling.

Note that amplitude-scaling the result does not change the temporal characteristics of the result; it merely changes the y-axis scaling.

6.5 Time-frequency analysis using complex Morlet wavelet convolution

Time-frequency analysis using complex Morlet wavelet convolution involves creating a "family" of wavelets with different frequencies, and convolving each wavelet with the data. The frequencies can be specified *a priori* if the range of frequencies in the data is known, or the frequencies can span the total possible range.

The total possible range of wavelet frequencies is limited by two factors. At the upper end, it is not possible to estimate frequencies higher than the Nyquist frequency, for the same reason that the upper frequency limit of the Fourier transform is the Nyquist frequency. At the lower end, there must be sufficient time in the time series to estimate at least one cycle, and preferably several cycles. That is, if the time series is one second long, it is not possible to estimate activity at 0.5 Hz, because the time series contains only one half of a cycle. Although technically, a 1 Hz response could be estimated in a one-second time series, it is preferential to limit the lower wavelet frequency bound to 2-3 cycles.

In the code below, the range of frequencies is based on what is known about the time series.

```
nfrex = 30;
frex = logspace(log10(2),log10(30),nfrex);
tf = zeros(nfrex,length(chirpTS));
chirpTSX = fft(chirpTS,Lconv);

for fi=1:nfrex
  w = 2*( 5/(2*pi*frex(fi)) )^2;
  cmwX = fft(exp(1i*2*pi*frex(fi).*wtime) ...
      .* exp( (-wtime.^2)/w ), Lconv);
  cmwX = cmwX./max(cmwX); % scale to 1
  convres = ifft( chirpTSX .* cmwX );
  tf(fi,:) = 2*abs(convres(halfwavL: ...
                  end-halfwavL-1));
end
contourf(t,frex,tf,40,'linecolor','none')
```

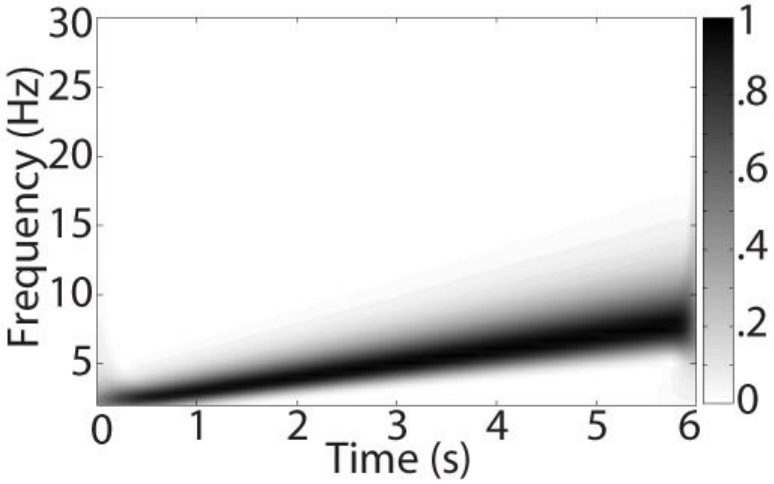

Figure 6.7 | Result of convolution between a family of complex Morlet wavelets and a 2-8 Hz chirp.

Note that it is not necessary to re-compute the Fourier transform of the time series at each frequency. This is one of the ways in which wavelet convolution can reduce computation time for time-frequency analyses, compared to the short-time Fourier transform. It is also a reason why performing the convolution using custom-written code can be faster and more efficient than using the Matlab **`conv`** function, which would involve redundantly re-computing the Fourier transform of the time series many times.

6.6 Temporally downsampling the results

As discussed in Chapter 5.2, the relatively slow changes in frequency characteristics compared to the sampling rate, in combination with temporal leakage, means that the temporal precision of the result of the convolution is generally lower compared to its temporal resolution. This in turn means that the result of convolution can often be temporally down-sampled without losing significant information. In the example below, rather than specifying a total number of data points to save, as was done with the short-time Fourier analysis, the requested time points are specified to range from 1 to 4.5 seconds in steps of 200 ms.

```
% define time points to plot
t2plot = dsearchn(t',(1:.2:4.5)');
% and the frequency to plot
f2plot = dsearchn(frex',6);
plot(t,tf(f2plot,:)), hold on
plot(t(t2plot),tf(f2plot,t2plot),'ro-')
```

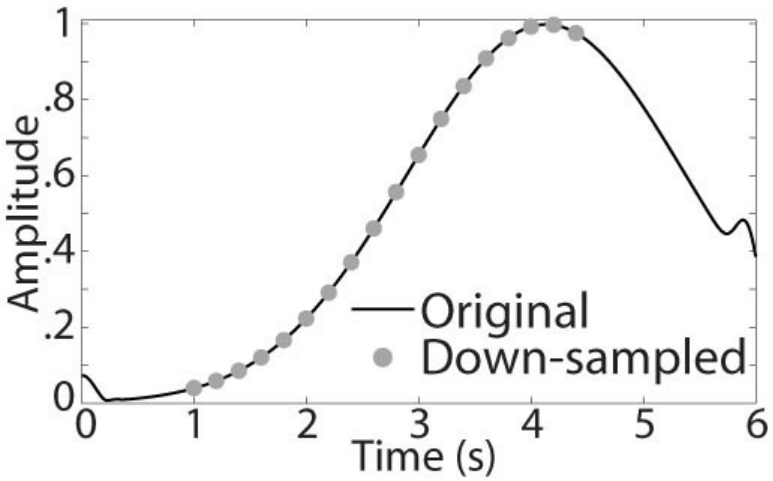

Figure 6.8 | Temporally downsampling results is often a good idea, but must be done carefully.

When down-sampling, it is important to inspect the results to check whether information has been lost. In the example above, ignoring the results after 4.5 seconds is inappropriate because it does not accurately portray the time course of the 6 Hz activity (the same is true for cutting off activity after 1 second when plotting 3 Hz power, which can be seen in the online Matlab code). Thus, down-sampling and selective temporal sampling should be done cautiously.

6.7 The Gaussian width parameter and the time-frequency trade-off

As discussed in Chapter 3.3 and mentioned above in Chapter 6.1, the width of a Gaussian function is defined by a standard deviation equation, which in turn is defined by a parameter frequently termed the "number of cycles." It is often abbreviated as *n*, although in the Matlab code in this book, **`ncyc`** is used (number of cycles) because **`n`** is used to indicate the number of time points in a time series. When **`ncyc`** is larger, the Gaussian is wider. When used to taper sine waves during the creation of Morlet wavelets, **`ncyc`** is scaled by **`2*pi*f`**, where **`f`** is the frequency of the sine wave.

This parameter has consequences for the results, and should be carefully selected. As seen in Figure 6.9 below, a wider Gaussian has improved frequency precision at the expense of poorer temporal precision, and vice-versa for the narrower Gaussian. This is because a wider Gaussian will contain more sine wave cycles and will thus more precisely measure narrow frequency activity. At the extreme, a sine wave tapered by a Gaussian with an infinite number of cycles is a pure sine wave, and the convolution becomes one step of a Fourier transform.

If the dynamics in the signal change rapidly, a narrower Gaussian (smaller **`ncyc`** parameter) will help determine *when* the changes take place, although the exact frequencies of those changes are less precisely known. On the other hand, if the dynamics change relatively slowly but frequency precision is important, a wider Gaussian (larger **`ncyc`** parameter) will help determine *at which frequency* the changes are occurring, but it will be difficult to know with certainty precisely when those changes occur. A reasonable range that balances both temporal and frequency precision is between 5 and 20.

```
freq2use = 5;
w = 2*( 7/(2*pi*freq2use) )^2;
cmw7 = exp(1i*2*pi*freq2use.*wtime) ...
         .* exp( (-wtime.^2)/w );
convres = ifft( chirpTSX .* fft(cmw7,Lconv) );
pow7 = abs(convres(halfwavL:end-halfwavL-1));

w = 2*( 3/(2*pi*freq2use) )^2;
cmw3 = exp(1i*2*pi*freq2use.*wtime) ...
         .* exp( (-wtime.^2)/w );
convres = ifft( chirpTSX .* fft(cmw3,Lconv) );
pow3 = abs(convres(halfwavL:end-halfwavL-1));

subplot(221)
plot(wtime,real(cmw7)), hold on
plot(wtime,real(cmw3),'r')

subplot(222)
hz = linspace(0,srate/2, ...
         floor(length(wtime)/2+1));
x7=2*abs(fft(cmw7)); x3=2*abs(fft(cmw3));
plot(hz,x7(1:length(hz))), hold on
plot(hz,x3(1:length(hz)),'r')
set(gca,'xlim',[0 20])

subplot(212)
plot(t,pow7), hold on
plot(t,pow3,'r')
```

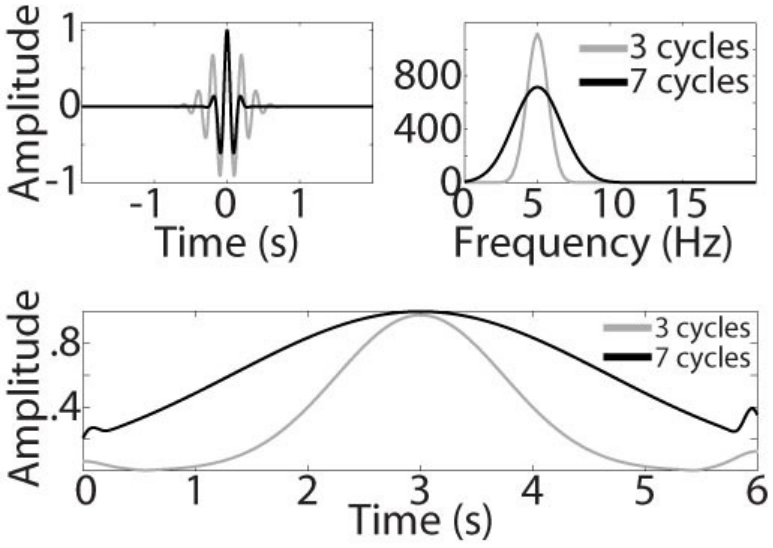

Figure 6.9 | Upper left panel shows two wavelets with Gaussian widths of 3 (gray) and 7 (black). Upper right panel shows their power spectra. Lower panel shows the result of convolution with the chirp.

In the example above, **`ncyc`** was fixed to a specific value. It is also possible to allow **`ncyc`** to vary over a range as a function of frequency.

```
nfrex = 9;
frex = logspace(log10(2),log10(30),nfrex);
ncyc = logspace(log10(3),log10(12),nfrex);
```

```
cmw_fam = zeros(nfrex,length(wtime));
for fi=1:nfrex
  w = 2*( ncyc(fi)/(2*pi*frex(fi)) )^2;
  cmw_fam(fi,:)=exp(1i*2*pi*frex(fi).*wtime)...
      .* exp( (-wtime.^2)/w );
end

for i=1:9
  subplot(3,3,i)
  plot(wtime,real(cmw_fam(i,:)))
  set(gca,'xlim',[-1 1])
end
```

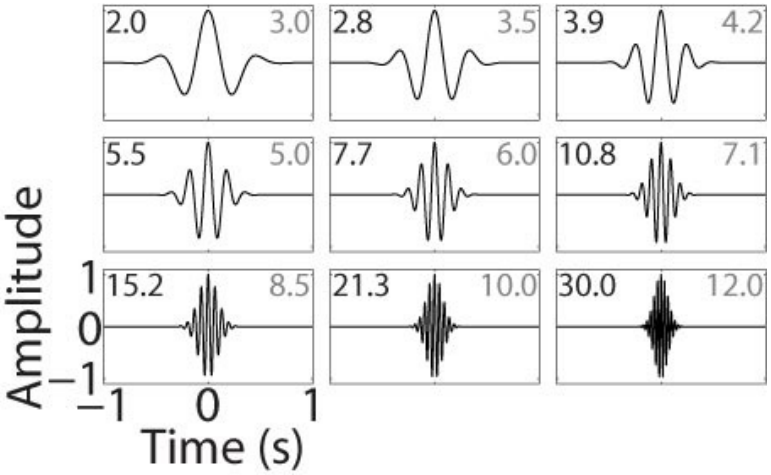

Figure 6.10 | Wavelets of different frequencies. Black numbers in the top left of each panel indicate frequencies in Hz; gray numbers in the top right indicate wavelet cycles (parameter **`ncyc`***).*

As is often the case in time-frequency analyses, there is no optimal choice for this parameter; a good strategy is to explore the data to determine the **`ncyc`** or range of **`ncyc`** that is best suited for those data. The choice is this parameter should also depend on expectations about whether temporal precision or frequency precision is more important in the time-frequency result.

6.8 Baseline normalizations

So far in this book, time-frequency plots have been shown in the amplitude scale of the original time series data. Although this is often useful because it facilitates interpretations of the results, there are three situations in which the time-frequency results should not be interpreted in the original scale, and thus should be normalized.

One situation is when the signal of interest reflects small changes summed on top of large-amplitude background activity. A second situation is when there are signals from different sources that have different scales, and the signals must be compared (an example is electrical vs. magnetic fields). A third situation is when there are multiple frequency components in a time series, and each component has a very different amplitude scaling.

This latter situation is illustrated below. A time series is created comprising five sine waves of different frequencies and amplitudes. The amplitudes of the different frequency components vary >10 orders of magnitude, and are thus not comparable with each other in

a single plot. A normalization is thus necessary in order to inspect changes in all frequencies in the same plot.

There are several normalizations that can be applied. One is called a decibel (dB), and is often used in signal processing to quantify the change in power of a signal, typically relative to a "baseline" or reference period. DB is defined as $10log_{10}(a/b)$, where a indicates the activity during the time period of interest and b indicates the average activity during the baseline period. In this case, a baseline period will be taken as the average activity of 2 seconds before the relevant signal is introduced.

```
% create signal
f = [6 14 25 40 70];
% note the widely varying amplitudes
a = [.001234 1.234 1234 123400 12340000];
% relevant amplitude modulations
m = [-.1 .1 -.2 .2 -.3];
signal = zeros(size(t));

for i=1:length(f)
  % compute 'base' signal
  signal = signal + a(i).*sin(2*pi*f(i).*t);
  % compute time-limited modulation
  extrasignal = m(i)*a(i)*sin(2*pi*f(i).*t) ...
                .* exp( -(t-7).^2 );
  % add modulation to signal
  signal = signal + extrasignal;
end

%-> wavelet convolution excluded for space <-%

% plot results
subplot(221), plot(t,tf)

% compute dB relative to baseline
bidx = dsearchn(t',[2 4]');
baseMean = mean(tf(:,bidx(1):bidx(2)),2);
db = 10*log10( bsxfun(@rdivide,tf,baseMean) );

subplot(224), plot(t,db)
```

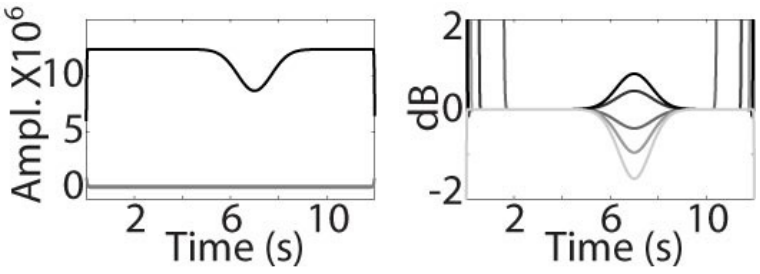

Figure 6.11 | Signals in different frequencies cannot be directly compared in one plot if their amplitudes vary over several orders of magnitude (left plot). Converting to dB allows simultaneous visual inspection of all signals, regardless of their original amplitude scaling (right plot). The near-vertical lines at the beginning and at the end of the time series are edge artifacts and should not be included in the baseline period. These edge

artifacts are contained inside the "buffer zones" and thus will not negatively influence the interpretation of the results between 2 and 10 seconds.

In addition to dB, several other normalizations are appropriate, including percent change or division by a scalar. If there is no appropriate "baseline" period, the average activity over all time points, or a separate but comparable time series, might be appropriate alternatives.

Note that because dB involves dividing by power, it is only appropriate for situations in which the baseline period contains non-zero power.

6.9 Exercises

1) First, generate 20 repeated measurements ("trials") of a time series, with each trial having a random phase between 0 and 2pi. Second, compute time-frequency power using Morlet wavelet convolution. Perform the convolution on each trial separately, and then average the time-frequency results together. Third, compute another time-frequency analysis on the trial-averaged time series (that is, first average the time series over trials in the time domain, and then perform convolution). Fourth, produce a figure with four subplots showing all trials together, the trial-average response, the time-frequency power from all trials, and the time-frequency power from the trial-average. All axis (x, y, color) limits should be the same for both analyses. Discuss differences in the time and time-frequency domain representations of the two sets of analyses.

```
%%% Solution to exercise 1
% The time series will have two overlapping
% sine waves, one with increasing power
% and one with decreasing power.

% zeroth, the basics:
srate     = 1000;
time      = 0:1/srate:6;
sinefreqs = [5 11]; % in hz
numTrials = 20;
wtime     = -2:1/srate:2;
nfrex     = 9;
frex      = logspace(log10(2),log10(30),nfrex);
ncyc      = logspace(log10(3),log10(12),nfrex);
Lconv     = length(time)+length(wtime)-1;
halfwavL= floor(length(wtime)/2);

% initialize matrices
signal = zeros(numTrials,length(time));
tf     = zeros(2,nfrex,length(time));

% first, create the signal
ampInc = linspace(0,1,length(time));
ampDec = linspace(1,0,length(time));

for ti=1:numTrials
  sine1 = ampInc.*sin(2*pi*sinefreqs(1)*time…
```

```
                        + rand*2*pi);
  sine2 = ampDec.*sin(2*pi*sinefreqs(2)*time…
                        + rand*2*pi);
  signal(ti,:) =  sine1+sine2;
end

% second, perform time-frequency decomposition
signalX1 = fft(signal,Lconv,2);
signalX2 = fft(mean(signal),Lconv);

for fi=1:nfrex
  % create wavelet
  w     = 2*( ncyc(fi)/(2*pi*frex(fi)) )^2;
  cmwX  = fft(exp(1i*2*pi*frex(fi).*wtime)...
              .* exp( (-wtime.^2)/w ), Lconv);
  cmwX  = cmwX./max(cmwX);

  % bsxfun allows simultaneous convolution
  convres    = ifft( ...
      bsxfun(@times,signalX1,cmwX) ,[],2);
  temppower  = 2*abs(convres(:,halfwavL: ...
      end-halfwavL-1));
  tf(1,fi,:) = mean(temppower,1);

  % convolution of trial average (third)
  convres    = ifft( signalX2.*cmwX );
  tf(2,fi,:) = 2*abs(convres(:,halfwavL: ...
                 end-halfwavL-1));
end

% fourth, plot the average and TF power
subplot(223)
contourf(time,frex,squeeze(tf(1,:,:)), ...
           40,'linecolor','none')
subplot(224)
contourf(time,frex,squeeze(tf(2,:,:)), ...
           40,'linecolor','none')
```

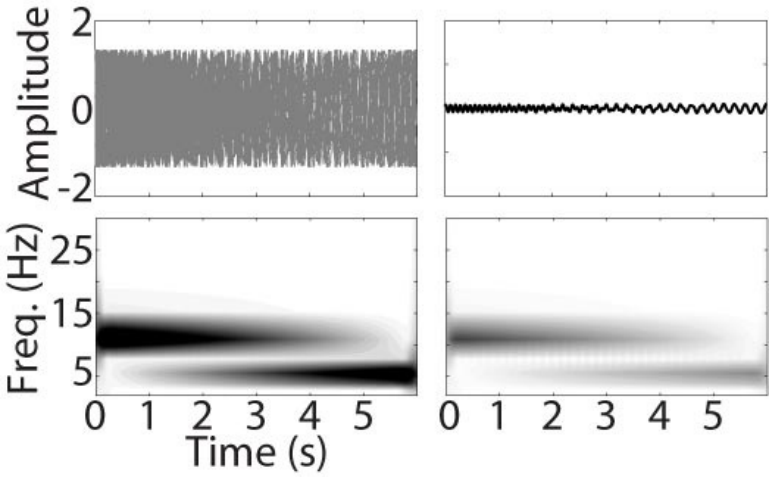

Figure 6.12 | When there are phase jitters across repeated measurements, computing time-frequency power is best done separately for each trial, and then the results can be averaged together (left-hand plots). Averaging in the time domain first can dampen or eliminate the dynamics (right-hand plots). Both time-frequency plots have the same color

scaling ([0 .6]).

---

2) Generate a 10-second time series that comprises the following two chirps: One lasting from 2 to 7 seconds and increasing in frequency from 10 Hz to 25 Hz while also increasing in amplitude, and one lasting from 5 to 10 seconds and decreasing in frequency from 45 Hz to 30 Hz while also decreasing in amplitude. Perform a time-frequency analysis of this time series using complex Morlet wavelet convolution. Justify your choice of parameter selection (number and range of wavelet frequencies, and their Gaussian widths).

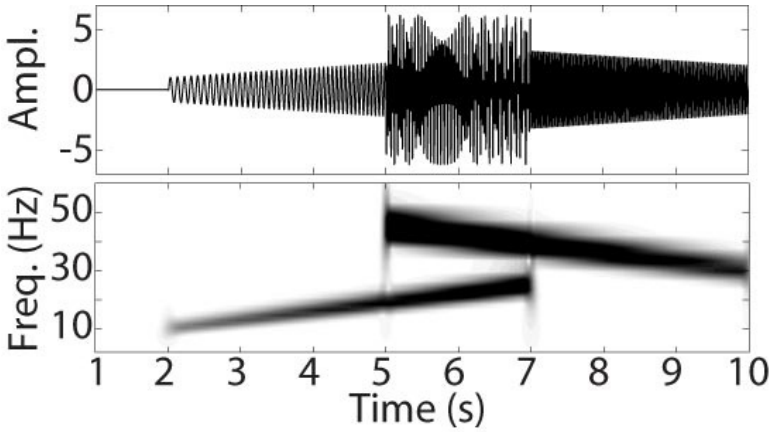

Figure 6.13 | Wavelet convolution with overlapping chirps.

3) Take the signal used in exercise 2, and add pink noise with 1/f characteristics. The maximum amplitude of the pink noise should be around twice the average amplitude of the noise-less signal. Re-do the time-frequency analysis. Does the noise have a stronger effect on the lower frequency or the higher frequency chirp, or roughly the same effect? Is it easier to understand the characteristics of the time series in the time or in the time-frequency domain?

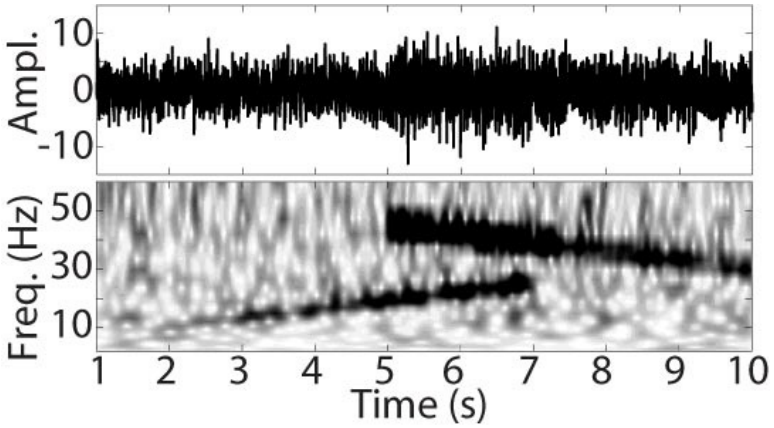

Figure 6.14 | Noisy chirps.

4) Re-do exercises 1 (analysis of single trials and then average, not the analysis of the trial-average) and 2 above (without noise), except using the short-time Fourier transform instead of Morlet wavelets. Plot the results of the two analyses (wavelet, short-time Fourier transform) next to each other. Based on visual inspection, are there any noticeable qualitative differences between the two approaches? Does one seem better than the other, and does this depend on the signal characteristics?

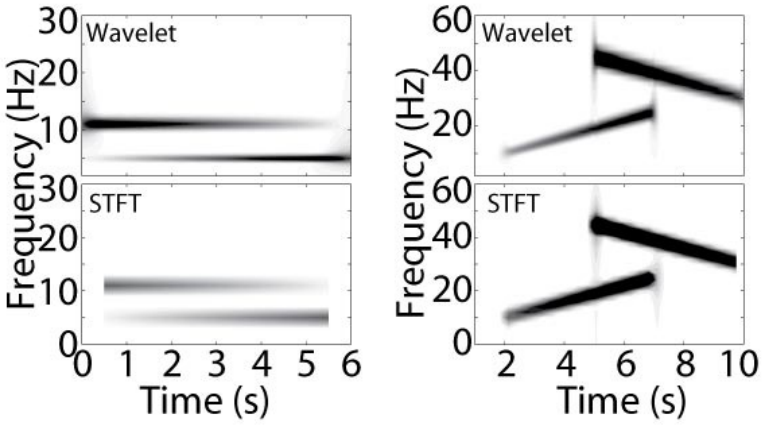

Figure 6.15 | Depending on parameter selection, Morlet wavelet convolution and short-time Fourier transform (STFT) can produce nearly identical results.

Chapter 7: The filter-Hilbert method

Band-pass filtering and then applying the Hilbert transform (the “filter-Hilbert method”) is another time-frequency analysis method that can produce similar results as the short-time Fourier transform and complex Morlet wavelet convolution. The main advantage of the filter-Hilbert method is that the frequency response of the filter can be specified (in contrast, for example, the frequency response of a Morlet wavelet is always Gaussian).

7.1 The Hilbert transform

The Hilbert transform is a way to convert a real-value-only time series to its complex representation (one that contains both a real part and an imaginary part), which in turn allows power and phase values to be extracted. The Hilbert transform involves computing a Fourier transform, rotating the positive frequencies (those between zero and the Nyquist) by -90^0 in complex space, rotating the negative frequencies (those above the Nyquist) by 90^0, summing the rotated frequencies back onto the original frequencies, and then computing the inverse Fourier transform. In practice, this can be implemented by doubling the positive frequencies, zeroing-out the negative frequencies, and computing the inverse Fourier transform. This produces a times series of complex numbers.

```
n=20; d=randn(n,1); dx=fft(d);
posF = 2:floor(n/2);   % positive frequencies
negF = floor(n/2)+2:n; % negative frequencies
dx(posF)=dx(posF)*2;
dx(negF)=0;
hilbertd = ifft(dx);
```

The Matlab Signal Processing toolbox, and the Octave Signal package (free but not included with the typical Octave installation), contain a function called **`hilbert`** that will compute the Hilbert transform. You can also create your own function using the code above.

The Hilbert transform does not affect the real part of a time series. This can be verified by plotting the real part of the Hilbert transform on top of the original time series.

```
plot(d), hold on
plot(real(hilbertd),'ro')
```

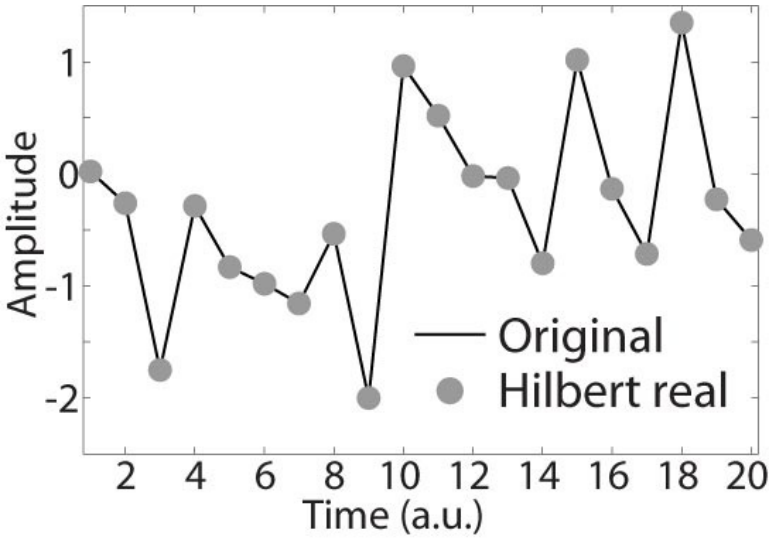

Figure 7.1 | The real part of the result of the Hilbert transform is the original time series

---

Power/amplitude and phase values can be extracted from the (complex) result of the Hilbert transform in the same way that the result of complex Morlet wavelet convolution or the short-time Fourier transform is treated, e.g., by using the functions **`abs`** or **`angle`**.

If the time series comprises one dominant frequency with little or no noise, the Hilbert transform can be applied to the raw time series data.

However, for multi-frequency time series, the Hilbert transform may be uninterpretable without first band-pass filtering. This is because the power and phase information from the Hilbert transform will reflect the broadband features of the time series, which are largely dominated by whatever frequency has the most power at each time point.

```
t=0:.01:5;
signal = zeros(size(t));
a=[10 2 5 8]; f=[3 1 6 12];
for i=1:length(a)
  signal = signal + a(i)*sin(2*pi*f(i)*t);
end
hilsine = hilbert(signal);
subplot(311), plot(t,signal)
subplot(312), plot(t,abs(hilsine).^2)
subplot(313), plot(t,angle(hilsine))
```

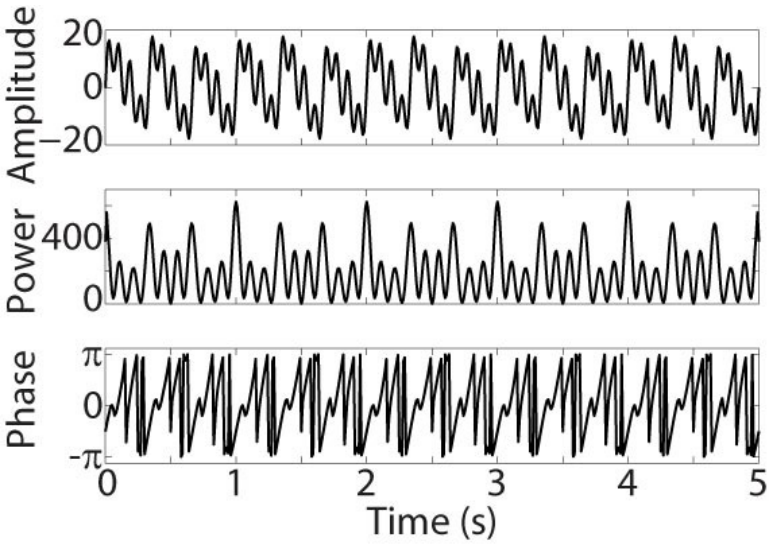

Figure 7.2 | Information extracted from the Hilbert transform: real part (original time series), power, and phase.

The estimated power and phase angles are not easily interpretable, in that they do not seem to characterize any of the constituent sine waves. Thus, it is often useful to filter the data before applying the Hilbert transform.

7.2 Introduction to filtering: Basic frequency-domain manipulations of time series data

The ability of the inverse Fourier transform to reconstruct a time series means that frequency-domain manipulations can be applied to time-domain data in order to modify the properties of that time series. A simple manipulation is to dampen the amplitude of a sine wave.

```
t=0:.001:1; sinewave=3*sin(2*pi*5*t);
plot(t,sinewave), hold on
sinewaveDamp=real(ifft( fft(sinewave)*.5) );
plot(t,sinewaveDamp,'r')
```

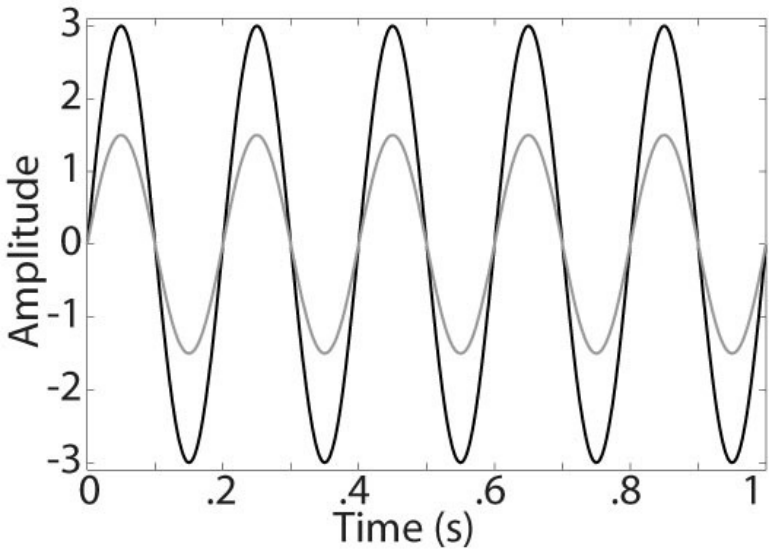

Figure 7.3 | A 3-unit amplitude sine wave (black) is attenuated after scaling the Fourier coefficients by 50%.

The spectral representation of the sine wave was dampened by 50%, and, not surprisingly, the peak of the reconstructed time domain sine wave is 1.5 (compare with the original peak amplitude of 3).

In the code below, a sum of two sine waves (5 Hz and 10 Hz) is created, and the 10 Hz sine wave is attenuated by down-scaling the corresponding frequencies in the frequency domain and then computing the inverse Fourier transform to get back to the time domain.

```
srate=1000; t=0:1/srate:1; n=length(t);
signal=3*sin(2*pi*5*t) + 4*sin(2*pi*10*t);
subplot(211), plot(t,signal), hold on
```

```
x=fft(signal)/n;
hz=linspace(0,srate/2,floor(n/2)+1);
subplot(212)
plot(hz,2*abs(x(1:length(hz))),'-*'), hold on

% frequencies to attenuate
hzidx=dsearchn(hz',[8 14]');
x(hzidx(1):hzidx(2)) = .1*x(hzidx(1):hzidx(2));
subplot(211)
 plot(t,real(ifft(x)*n),'r')
subplot(212)
 plot(hz,2*abs(x(1:length(hz))),'r-o')
set(gca,'xlim',[0 15])
```

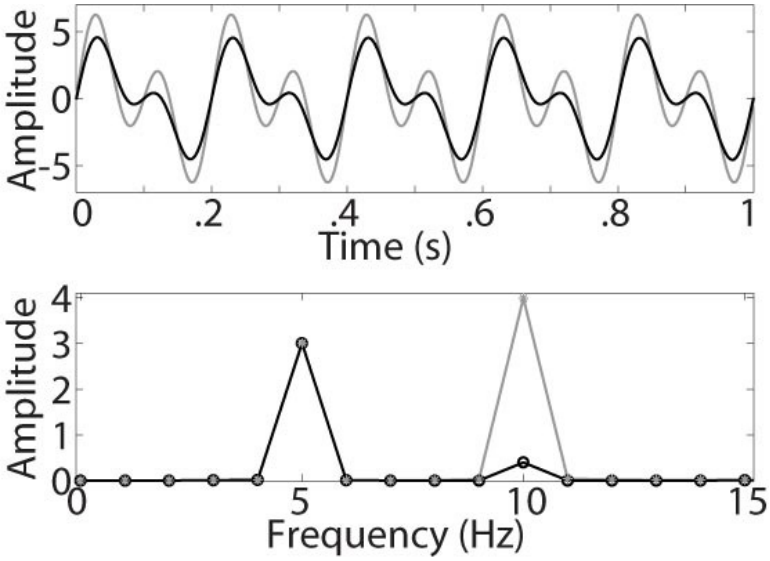

Figure 7.4 | Frequency-selective dampening of a multi-sine signal.

Note that because the Fourier coefficients were scaled down (**`x=fft()/n`**), the coefficients need to be scaled back up during the inverse Fourier transform. Also note that the frequency-domain attenuation was applied not only to the peak frequency, but also to several surrounding frequencies. This is because the finite sine wave in combination with some edge artifacts cannot be represented by a perfect impulse at only one frequency (try examining the spectral plot when the y-axis scaling is changed to [0 .1]). Thus, the side bands are also attenuated.

7.3 Finite vs. infinite impulse response filters

FIR and IIR (finite or infinite impulse response) describe the filter response when the input is an impulse—a single non-zero number. The difference is that the response of an FIR filter ends at some point (thus, a finite response) whereas the response of an IIR filter is infinite.

Whether to use an FIR or IIR filter depends on the application. If the goal of filtering is to obtain a time-domain signal in a specified frequency range, FIR and IIR filters will generally provide very similar results, and thus the choice of FIR vs. IIR can be based on experience or comfort implementing a specific type of filter.

If the goal of filtering is to obtain phase values for which subsequent analyses will be

performed, such as computing instantaneous frequency, phase synchronization, or phase stability, an FIR filter should be preferred. FIR filters are more stable, whereas IIR filters can introduce some phase disturbances if the filter is not well constructed. FIR filters take slightly longer to implement due to their higher filter order, but in practice this is not a major limitation for modern computers.

7.4 Causal vs. non-causal filtering

Applying a filter to a time series introduces a phase shift in the data, with the amount of the phase shift determined by the length of the filter kernel. These phase shifts are an artifact and should be corrected when possible.

The phase shift can be corrected by filtering the data, then flipping the time series backwards and applying the same filter, and finally flipping the data forward again. The principle behind this flipping is that the phase distortions introduced by filtering are re-introduced in reversed-time, thus reverse-distorting the distortions. The result is a filtered time series without phase distortions. This is known as a non-causal filter.

Non-causal filters can only be applied after the data have been collected, because in real-time filtering one does not have access to data from the future. In real-time filtering, if the kernel is fairly short, the phase delay might be negligible. Only non-causal filters will be discussed in this book.

Non-causal filtering can be implemented using the function **`filtfilt`**, which is contained in the Matlab Signal Processing toolbox and the Octave Signal package.

7.5 Frequency edge artifacts in the time domain

In Chapter 4 it was shown that sharp edges in a time-domain signal have a “messy” spectral representation. The same is true the other way around: Sharp edges in the frequency domain can introduce “ripple” artifacts in the time domain. This is illustrated below by adding sharp edges to the spectral representation of a sine wave, and then plotting the inverse-Fourier-reconstructed time-domain signal.

```
srate=1000; t=0:1/srate:5;
sinewave=sin(2*pi*2*t);
subplot(211), plot(t,sinewave), hold on

x=fft(sinewave)/length(t);
hz=linspace(0,srate/2,floor(length(t)/2+1));
subplot(212)
plot(hz,2*abs(x(1:length(hz))),'-*'), hold on

hzidx=dsearchn(hz',[8 9]');
x(hzidx(1):hzidx(2)) = .5;
subplot(211)
 plot(t,real(ifft(x)*length(t)),'r')
set(gca,'ylim',[-5 5])
subplot(212)
 plot(hz,2*abs(x(1:length(hz))),'r-o')
```

```
set(gca,'xlim',[0 15])
```

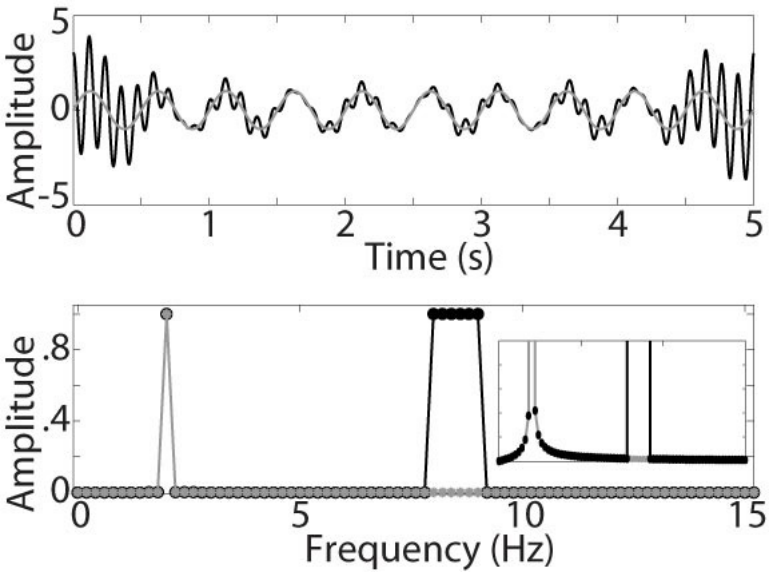

Figure 7.5 | Sharp edges in the frequency domain (here, the addition of an 8-9 Hz plateau) can cause ripples in the time domain. The sharp edges are more visible in the inset plot, which shows the same data with focused y-axis scaling ([0 .005])

7.6 High-pass and low-pass filters

A time series can be filtered by computing its Fourier representation, dampening the unwanted frequencies, and then computing the inverse Fourier transform.

The code below illustrates a high-pass filter. “High-pass” means that only frequencies above a certain cut-off remain in the data, in other words, higher frequencies are passed through the filter while lower frequencies are attenuated.

To avoid ripple artifacts in the time domain as shown above, the filter should be designed to have a smooth transition in the frequency domain. The shape of the filter in the example below is a sigmoid centered at 7 Hz. When the Fourier spectrum is multiplied by the sigmoid, the amplitudes of lower frequencies are attenuated while the amplitudes of higher frequencies are preserved.

```
srate=1000; t=0:1/srate:3; n=length(t);
frex = logspace(log10(.5),log10(20),10);
amps = 10*rand(size(frex));
data = zeros(size(t));
for fi=1:length(frex)
    data=data+amps(fi)*sin(2*pi*frex(fi)*t);
end
subplot(211), plot(t,data), hold on

x=fft(data);
hz=linspace(0,srate,n);
subplot(212), plot(hz,2*abs(x)/n,'-*')
hold on, set(gca,'xlim',[0 25])
filterkernel = (1./(1+exp(-hz+7)));
x = x.*filterkernel;
subplot(211), plot(t,real(ifft(x)),'r')
```

```
subplot(212), plot(hz,2*abs(x/n),'r-o')
```

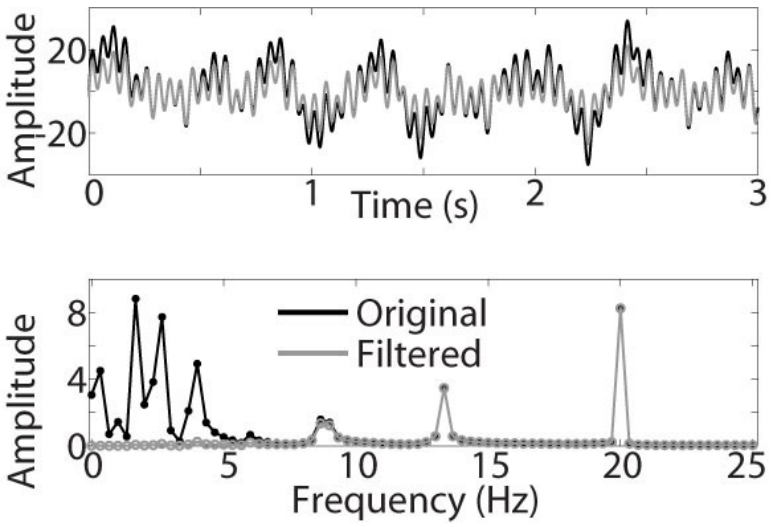

Figure 7.6 | A high-pass filter attenuates low-frequency activity.

Note that the frequencies are computed differently here compared to previously (variable **`hz`**). Previously, this variable was defined only up to the Nyquist frequency whereas now it is defined up to the number of points in the time series. This is a programming trick to evaluate the sigmoidal function over the entire range of frequencies. The frequency labels are correct up until the Nyquist frequency, but are invalid above that. Care must be taken during plotting to avoid confusion.

A similar procedure underlies low-pass filtering.

```
subplot(211), plot(t,data), hold on
x=fft(data); hz=linspace(0,srate,n);
subplot(212), plot(hz,2*abs(x)/n,'-*')
hold on, set(gca,'xlim',[0 25])
filterkernel = (1-1./(1+exp(-hz+7)));
x = x.*filterkernel;
subplot(211), plot(t,real(ifft(x)),'r')
subplot(212), plot(hz,2*abs(x/n),'r-o')
```

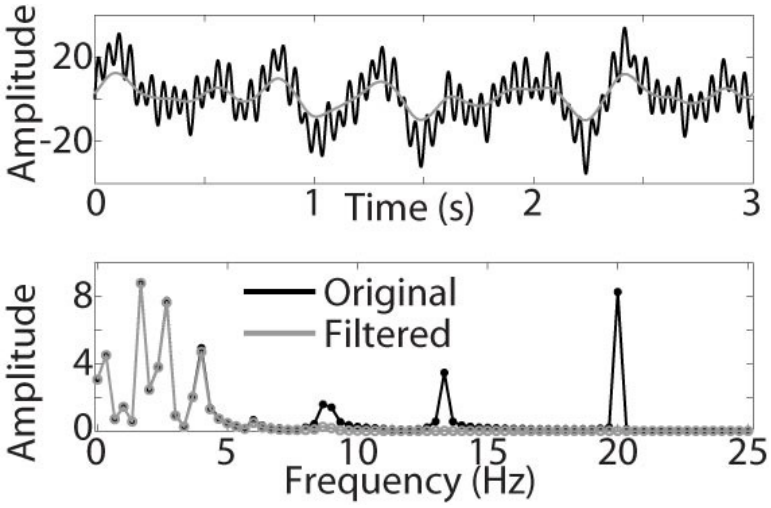

Figure 7.7 | A low-pass filter attenuates high-frequency activity.

7.7 Plateau-shaped band-pass filters

The idea of a band-pass filter is to combine a low-pass with a high-pass filter, in order to isolate one frequency range while attenuating both lower and higher frequencies. Plateau-shaped filters are popular band-pass filters.

A perfect plateau-shaped filter would contain edge artifacts at the upper and lower bounds, as seen in Figure 7.5. Therefore it is useful to include “transition zones” that smooth the edges of the plateau-shaped filter. Transition zones are typically set to between 5% and 20% of the frequency edges, depending on the desired frequency specificity.

To ensure that edge artifacts are attenuated, the desired plateau shape can be provided as an input to the function **`firls`** (finite impulse response, least squares). **`firls`** will design a time-domain kernel with a frequency response that matches the plateau but with smoothed transition zones that attenuate edge artifacts.

The “order” of the filter defines the number of time points in the filter kernel (the kernel is order+1 points long). It is specified according to the lowest frequency of the band-pass filter: The filter kernel must be at least longer than one cycle of the lowest frequency.

In practice it is good to set the order to be 3-8 times the number of time points in one cycle of the lowest frequency. A filter kernel with a higher order will increase the precision of the frequency response of the filter, though it will also slightly increase computation time. Note that Matlab will produce an error if the order is less than three times the length of the time series.

The example below requires the Matlab Signal Processing toolbox or the Octave Signal package.

```
srate=1000; nyquist=srate/2;
band = [4 8]; % Hz
twid = 0.2; % transition zone of 20%
filtO = round(3*(srate/band(1)));
freqs = [ 0 (1-twid)*band(1) band(1) ...
          band(2) (1+twid)*band(2) nyquist…
         ]/nyquist;
idealresponse = [ 0 0 1 1 0 0 ];
filterweights = firls(filtO,freqs,idealresponse);
filtered_data = filtfilt(filterweights,1,data);

subplot(211), plot(freqs*nyquist,idealresponse)
filterx = fft(filterweights);
hz=linspace(0,nyquist,floor(filtO/2)+1);
hold on
plot(hz,abs(filterx(1:length(hz))),'r-o')
set(gca,'xlim',[0 50])

subplot(212), plot(filterweights)
```

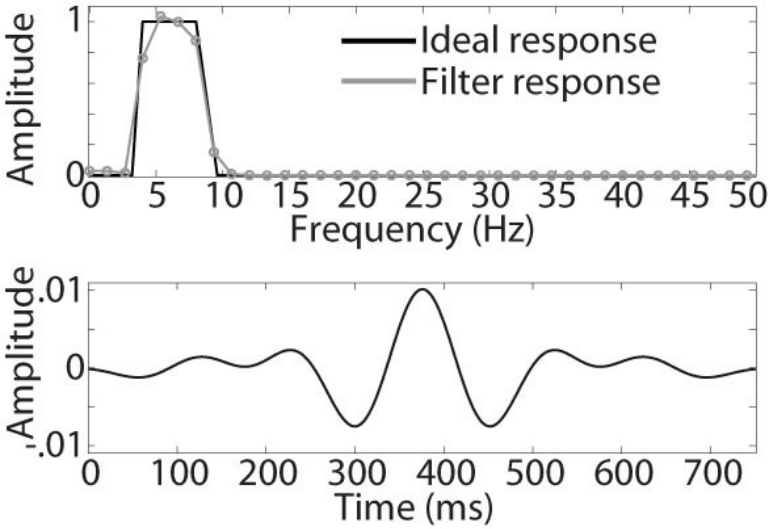

Figure 7.8 | The ideal and the real.

The upper plot shows both the "ideal" plateau-shaped filter (with 20% transition zones) as well as the spectral profile of the actual filter that was constructed by **`firls`**. The actual filter is smoothed to avoid introducing edge artifacts. The lower plot shows the time-domain filter kernel.

Band-pass filter kernels generally have a wavelet-like appearance. This is not surprising considering the frequency shape of the actual filter kernel, which appears to be something in between a plateau and a Gaussian. Indeed, if the ideal response is specified to be a perfect Gaussian, the time-domain filter kernel will be a Morlet wavelet.

Once the filter kernel is obtained, there are two ways to use the kernel to filter the data. First, the filter kernel can be used in convolution as described in the previous chapter. Because the kernel is real-valued (not complex), the Hilbert transform must be applied to the result of convolution in order to obtain phase and power information.

```
Lconv = length(filterweights)+length(data)-1;
halfwavL = floor(length(filterweights)/2);
convres = real(ifft( fft(data,Lconv).* ...
           fft(filterweights,Lconv) ,Lconv));
convres = convres(halfwavL:end-halfwavL-1);
```

The second approach is to use the function **`filtfilt`**, which applies the kernel to the data using a zero-phase-shift (non-causal) filter as described in Chapter 7.4.

```
filtdat = filtfilt(filterweights,1,data);
plot(t,data), hold on
plot(t,convres,'r')
plot(t,filtdat,'k')
```

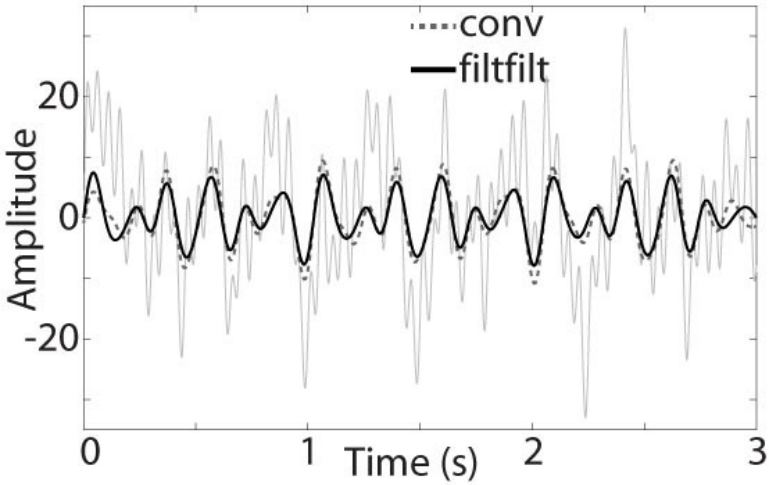

Figure 7.9 | Band-pass filtered signal, via convolution and Matlab function ***filtfilt****.*

The results of convolution and the **filtfilt** function are similar but not identical. Minor discrepancies result from kernel scaling that is computed and applied by **filtfilt**. These discrepancies are generally not a significant concern. Indeed, changing the wavelet width or the band-pass FIR filter transition width will cause similar or even larger discrepancies.

7.8 Exercises

1) Generate a time series that increases from 5 Hz to 15 Hz over 4 seconds, with randomly varying amplitude. Convolve this time series with a 10-Hz complex Morlet wavelet, and, separately, apply the filter-Hilbert method using a plateau-shaped band-pass filter centered at 10 Hz. From both results, extract the filtered time series and the power, and plot them simultaneously. Do the results from these two methods look similar or different, and how could they be made to look more similar or more different?

```
% First define the basics
srate=1000; nyquist=srate/2;
t=0:1/srate:4;
peakfreq = 10; % Hz

% Second, create chirp. This formulation
% differs from that in Chapter 3, and is
% explained in more detail in Chapter 8.
freqTS   = linspace(5,15,length(t));
centfreq = mean(freqTS);
k        = (centfreq/srate)*2*pi/centfreq;
pow = abs(interp1(linspace(t(1),t(end),10), ...
            10*rand(1,10),t,'spline'));
signal = pow.*sin(2*pi.*centfreq.*t + ...
           k*cumsum(freqTS-centfreq));

% Third, wavelet convolution
wtime = -2:1/srate:2;
Lconv = length(t)+length(wtime)-1;
halfwavL = floor(length(wtime)/2);
% compute and normalize wavelet
```

```
w    = 2*( 6/(2*pi*peakfreq) )^2;
cmwX = fft(exp(1i*2*pi*peakfreq.*wtime) .* ...
               exp( (-wtime.^2)/w ), Lconv);
cmwX = cmwX./max(cmwX);
% run convolution
convres = ifft( fft(signal,Lconv) .* cmwX );
convres = convres(halfwavL:end-halfwavL-1);

% Fourth, band-pass filter
band   = [peakfreq-.5 peakfreq+.5];
twid   = 0.15;
filtO  = round(3*(srate/band(1)));
freqs  = [0 (1-twid)*band(1) band(1) band(2)...
          (1+twid)*band(2) nyquist ]/nyquist;
ires = [ 0 0 1 1 0 0 ];
fweights = firls(filtO,freqs,ires);
filtdat = filtfilt(fweights,1,signal);

% Fifth, plot results
subplot(311)
 plot(t,signal)
subplot(312)
 plot(t,real(convres)), hold on
 plot(t,filtdat,'r')
subplot(313)
 plot(t,2*abs(convres)), hold on
 plot(t,2*abs(hilbert(filtdat)),'r')
```

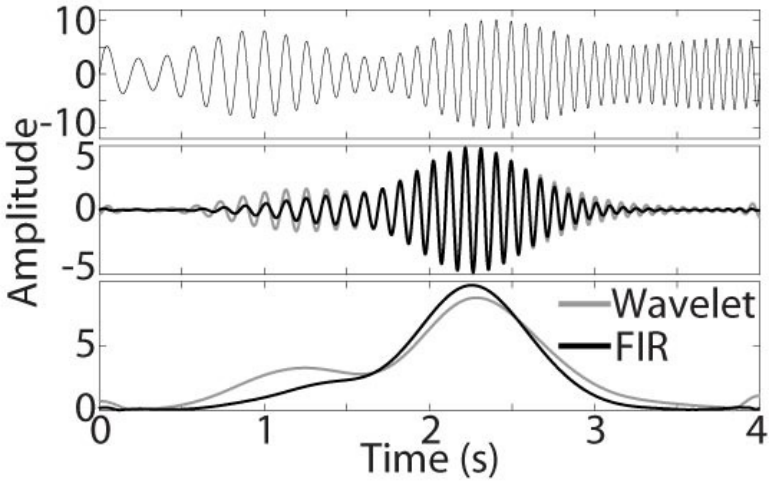

Figure 7.10 | The top panel shows a chirp signal, the middle panel shows the 10-Hz filtered signals using wavelet (gray line) and FIR band-pass filtering (black line), and the bottom panel shows 10 Hz amplitude.

---

2) Reproduce the time-frequency power plot in Figure 5.1 (a signal comprising concatenated sine waves at 30, 3, 6, and 12 Hz over 5 seconds with a sampling rate of 1000 Hz), but using the filter-Hilbert method. To complete this exercise, it is necessary to filter the signal in multiple overlapping frequency bands. How do the results compare to those shown in Figure 5.1? Is it now possible to estimate power before .5 seconds and after 4.5 seconds without zero-padding or reflection? Why or why not?

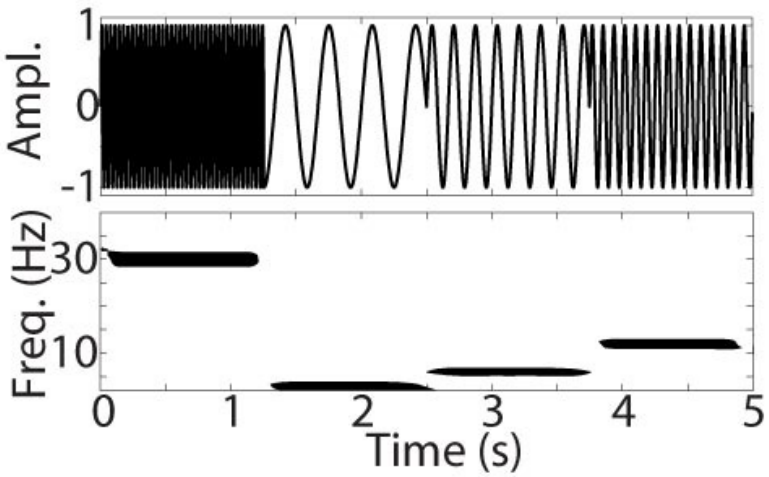

Figure 7.11 | Time-frequency power estimated from the filter-Hilbert method (compare with Figure 5.1).

3) With Morlet wavelet convolution, the trade-off between temporal precision and frequency resolution is defined by the width of the Gaussian that tapers the sine wave; in band-pass filtering, this trade-off is defined by the width of the filter in the frequency domain (variable **`twid`** in the code earlier in this chapter).

Explore this parameter by performing time-frequency decomposition on a chirp (ranging from 20-80 Hz over 6 seconds with a sampling rate of 1000 Hz) using four sets of parameters: Constant narrow-band filters, constant wide-band filters, filters with widths that increase as a function of frequency, and filters with widths that decrease as a function of frequency. Note that there is a limit as to how wide the filters can be at lower frequencies (it is not possible to have a filter with a 3 Hz center and a 10-Hz width). Are there noticeable differences amongst these settings, and in which circumstances would it be best to use each parameter setting?

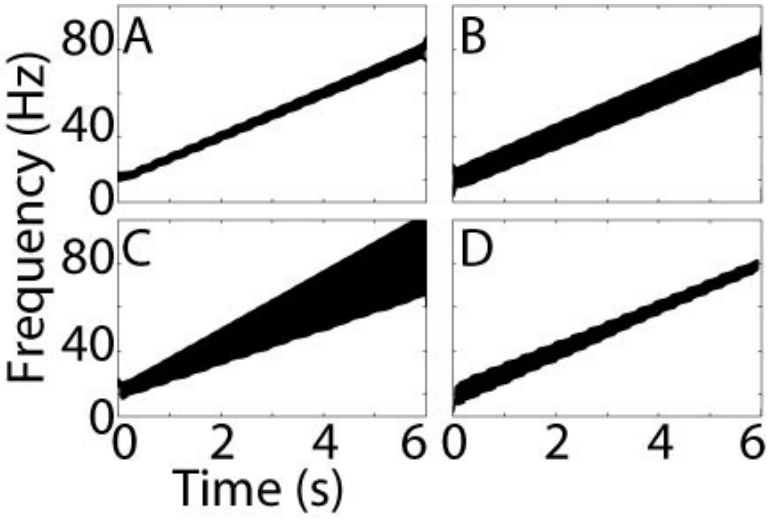

Figure 7.12 | Time-frequency power via the filter-Hilbert method using four different parameter settings for the width of the band-pass filter: constant narrow (panel A), constant wide (panel B), increasing (panel C), decreasing (panel D).

Chapter 8: Instantaneous frequency in non-frequency-stationary signals

Frequency and time-frequency analyses assume that the oscillator being measured has a stable—that is, stationary—frequency structure over time, at least in the time window of the analysis (for example, the non-zero portions of the Gaussian in Morlet wavelet convolution). In many systems, however, oscillator frequencies can change quickly over time. Indeed, time-varying changes in frequency can be a method to convey information, such as in FM radio.

It is important to realize that an oscillation frequency cannot be precisely measured at a single instant in time, similar to how heart rate cannot be known by measuring the electrocardiogram for only one time point. Thus, instantaneous frequency at each time point must be estimated based on surrounding time points. Because lower frequency oscillations are by definition slower (that is, the same number of cycles takes up more time as frequencies decrease), instantaneous frequency is more temporally precise at higher frequencies. This is one of the reasons why FM radios use the MHz range, rather than the Hz range.

8.1 Generating signals with arbitrary time-varying changes in frequency

Before learning how to extract instantaneous frequencies, it is useful first to learn how to create time series with a frequency structure that varies over time. Linear chirps (a time series with linearly increasing or decreasing frequency over time) were introduced in Chapter 3.2. The code below is a more general implementation that will create a time series with time-varying frequencies of any arbitrary shape.

```
srate=1000; t=0:1/srate:5;
freqTS = linspace(1,20,length(t));
centfreq = mean(freqTS);
k = (centfreq/srate)*2*pi/centfreq;
y = sin(2*pi.*centfreq.*t + ...
        k*cumsum(freqTS-centfreq));
plot(t,y)
```

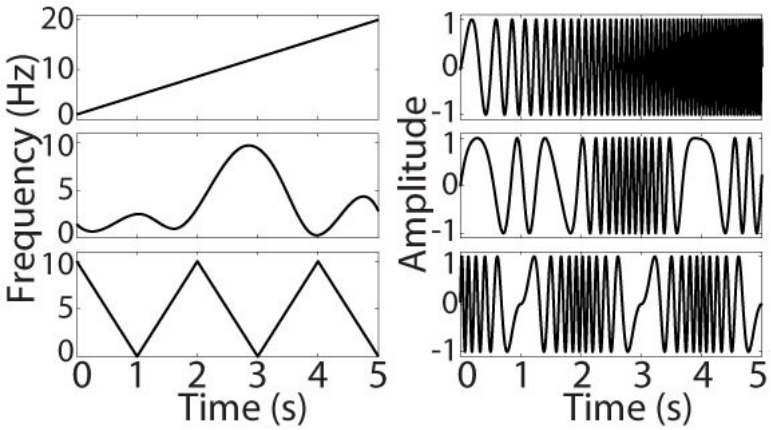

Figure 8.1 | Diversity of arbitrary frequency responses.

The above figure illustrates three possibilities of generating sinusoidal time series with non-stationary frequencies. The first is a linear chirp. The second and third examples are of an interpolated random number vector and a triangle. Producing these signals involves changing only the **`freqTS`** variable, which is shown in the online Matlab code.

8.2 Estimating instantaneous frequency

Frequency can be defined as the derivative of the phase angle time series. This is sometimes referred to as instantaneous angular velocity, because it is the speed with which the phase angle vector spins around the polar plane at the smallest time measurement possible (from one time point to the next, which depends on the sampling rate). Because instantaneous frequency is computed only based on the phase values, it is independent of any ongoing dynamics in the power of the signal (with the exception that at zero power, phase values are not defined, and therefore instantaneous frequency cannot be computed).

The phase angle time series can be extracted from a complex time series (e.g., the result of convolution with a complex wavelet or the filter-Hilbert method) via the Matlab function **`angle`**. Thereafter, the first derivative of the phase angle time series is extracted. The phase angles must be unwrapped first. Unwrapping phase angles means that each time the angles transition from -pi to +pi (as the vector moves past the positive real axis in a polar plane), 2pi is added to the phase angle time series.

The derivative of the unwrapped phase angle time series is the estimate of instantaneous frequency. To scale the result to units of hertz, the derivative is multiplied by the data sampling rate and divided by 2pi.

```
srate=1000; t=0:1/srate:5; n=length(t);
f = [4 6];
ff = linspace(f(1),f(2)*mean(f)/f(2),n);
signal = sin(2*pi.*ff.*t);
phases = angle(hilbert(signal));
angVeloc = diff(unwrap(phases));
instFreq1 = srate*angVeloc/(2*pi);

subplot(211), plot(t,signal)
subplot(212), plot(t(1:end-1),instFreq1)
```

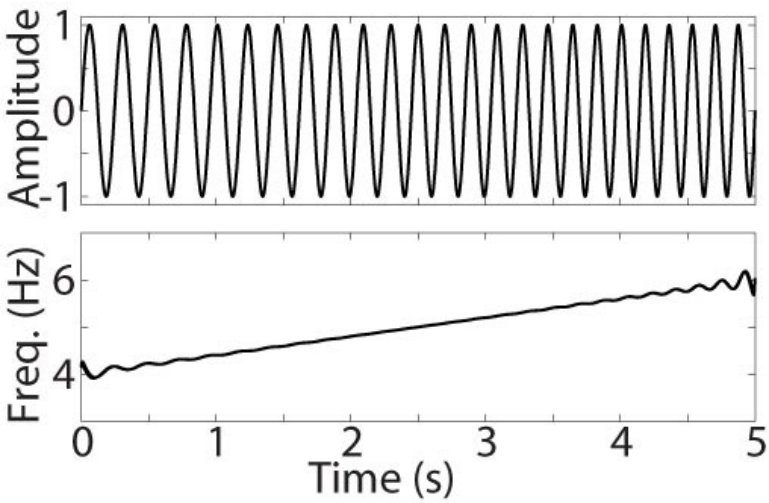

Figure 8.2 | Sliding frequencies.

Note the presence of edge artifacts in the reconstructed instantaneous frequency time series. These could be attenuated or avoided if there were buffer zones before and after the chirp.

Because the instantaneous frequency is based on the temporal derivative (each point is the difference between itself and the previous point), the time series of instantaneous frequency is one point shorter than the length of the original time series. This can cause annoyances or errors during subsequent analyses. It is thus useful to add an extra data point at the end so that the time series of instantaneous frequencies retains the same length, as was discussed in Chapter 3.6 concerning using the derivative to make a time series more stationary. To accomplish this, the final data point can be repeated.

```
instFreq1(n) = instFreq1(n-1);
```

8.3 Band-limited instantaneous frequency

The code presented in the previous section works well only if the time series is strongly dominated by one frequency and if there is little or no noise. This is shown below by estimating instantaneous frequencies after adding background noise (a time series comprising four sine waves) to the signal created earlier.

```
a=[10 2 5 8]/10; f=[3 1 6 12];
background = zeros(size(t));
for i=1:length(a)
  background = background + ...
           a(i)*sin(2*pi*f(i)*t);
end
data = signal + background;
instFreq2 = srate*diff(unwrap(angle(hilbert(...
                  data))))/(2*pi);
subplot(211), plot(t,data)
subplot(212), plot(t(1:end-1),instFreq2)
```

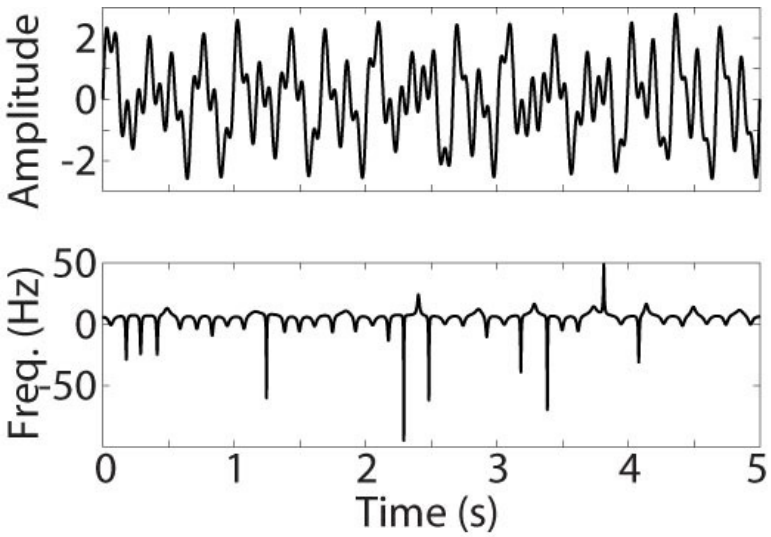

Figure 8.3 | Overlapping frequencies make instantaneous frequency difficult to interpret.

The estimated instantaneous frequencies in Figure 8.3 do not accurately reflect the frequency content of the sine wave. Indeed, the estimated instantaneous frequencies contain frequencies much higher than those present in the data (>12 Hz), as well as uninterpretable brief negative frequencies. Inspecting the distribution of instantaneous frequencies (using **`hist(instFreq,100)`**) shows that the most frequent instantaneous frequency is around 7 Hz, which also does not make sense considering the actual frequency content of the original time series.

Visually inspecting the phase angle time series in Figure 8.4 reveals the cause of this poor performance. The phase angle time series defined by the broadband (non-filtered) time series is non-monotonic and contains many sudden increases and decreases in the phase angle time series.

```
plot(t,angle(hilbert(data)))
```

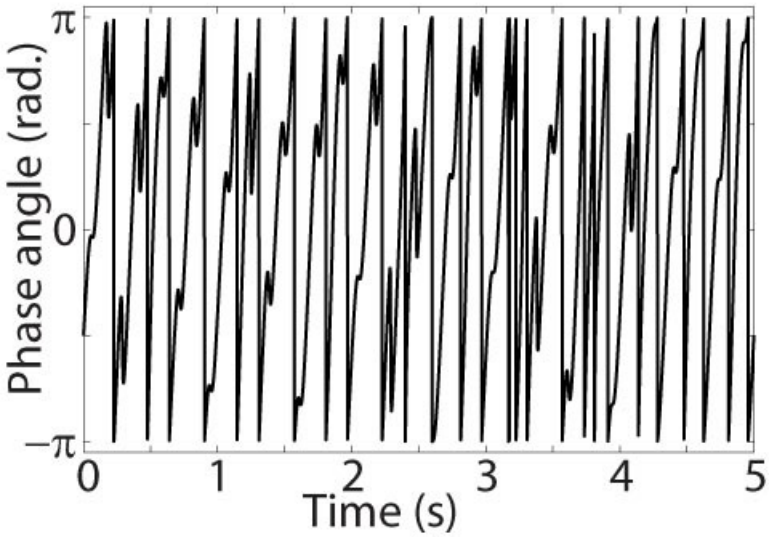

Figure 8.4 | Non-monotonic phases produce uninterpretable instantaneous frequencies.

In multi-frequency signals, or in uni-frequency time series that contain noise, instantaneous frequencies can be interpreted only after isolating a specific frequency range by band-pass filtering. The code below uses complex Morlet wavelet convolution to isolate one narrow frequency range in the data. It is also possible to use a plateau-shaped filter as illustrated in Chapter 7.

```
wtime = -2:1/srate:2;
wavefreq = 5; % Hz
w = 2*( 5/(2*pi*wavefreq) )^2;
cmw = exp(1i*2*pi*wavefreq.*wtime) ...
         .* exp( (-wtime.^2)/w );
halfwavL = floor(length(wtime)/2);
Lconv = length(t)+length(wtime)-1;

convres = ifft(fft(data,Lconv).*fft(cmw,Lconv));
convres = convres(halfwavL:end-halfwavL-1);
```

```
instFreq3 = diff(unwrap(angle(convres)));
instFreq3(n) = instFreq3(n-1);

plot(t,instFreq1), hold on,
plot(t,srate*instFreq3/(2*pi),'r')
```

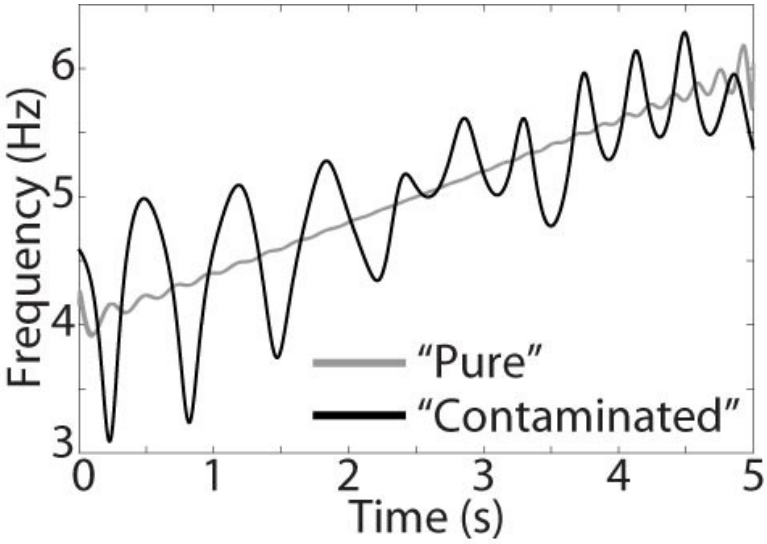

Figure 8.5 | Band-pass filtering improves estimation of instantaneous frequencies in the presence of "contamination" by other overlapping data components (the "pure" instantaneous frequency time series is reproduced here from Figure 8.2).

The estimated instantaneous frequency time series of the filtered data more closely reflects the true instantaneous frequencies, although there are residual effects of the background activity. A wider Gaussian when creating the wavelet will help reduce these artifacts by increasing frequency specificity; try changing the number of cycles in the Morlet wavelet from 5 to 10.

It may initially seem strange to isolate one narrow frequency range and then estimate instantaneous frequencies within that range. However, frequency non-stationarities that are small relative to the "carrier" frequency around which those non-stationarities fluctuate are common. Indeed, FM radio is based on the principle of time-varying changes in frequency within a narrow frequency band.

Nonetheless, the procedure of estimating instantaneous frequencies after applying a narrow-band filter is valid only for signals with frequency non-stationarities that are within the range of the band-pass filter.

8.4 Estimating instantaneous frequency in the presence of broadband noise

Noise can cause considerable problems for computing an accurate time series of instantaneous frequencies. This is particularly the case for broadband noise. Below is an illustration of how even a small amount of noise can prove disastrous for estimating instantaneous frequencies without any filtering.

```
data = signal + randn(size(signal));
phases = angle(hilbert(data));
angVeloc = diff(unwrap(phases));
```

```
instFreq = srate*angVeloc/(2*pi);
instFreq(end+1) = instFreq(end);

subplot(311), plot(t,data)
subplot(312), plot(t,phases)
subplot(313), plot(t,instFreq)
```

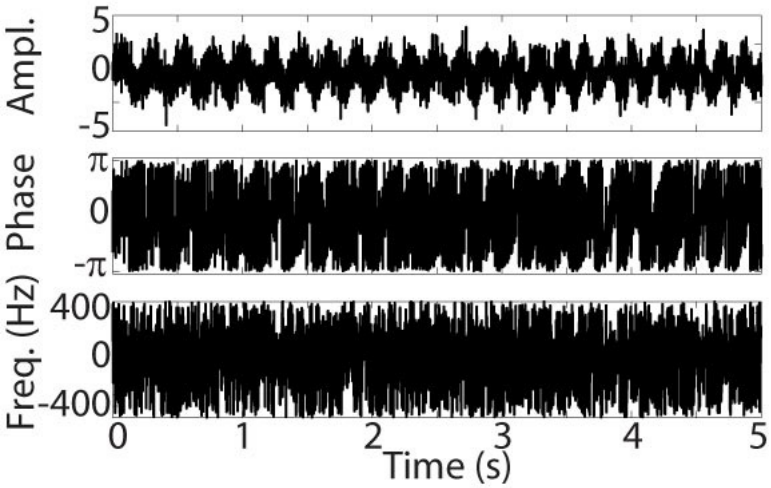

Figure 8.6 | Noise can cause uninterpretable instantaneous frequencies.

Although the sine wave is easily recognizable with the added noise (top plot), the estimated instantaneous frequencies are uninterpretable due to the large spikes as well as negative frequencies (lower plot).

The problem lies with the phase angle times series (middle plot), which contains considerable noise. Although the magnitude of noise is fairly small, the jumps are larger than the smooth changes in the sine wave, and thus create disproportionately large derivative values, which in turn create unrealistically and uninterpretably large jumps to positive and negative frequencies.

Results like this indicate either that instantaneous frequency should not be computed on this time series, or that filtering is necessary before interpreting the result.

In this example, using the same wavelet convolution that was used for Figure 8.5 works well. The results now match the instantaneous frequency of the original noiseless chirp fairly well, as seen in the figure below.

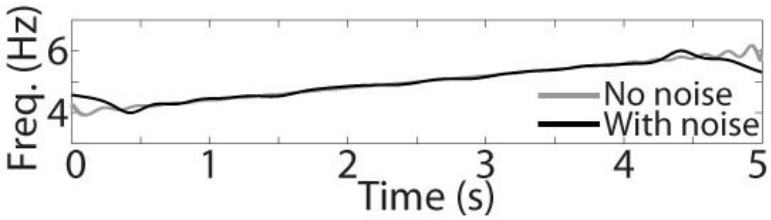

Figure 8.7 | A simple filter (in this case, a 5-Hz complex Morlet wavelet) can effectively reduce the noise that otherwise makes instantaneous frequency uninterpretable.

Estimating instantaneous frequency using the method shown above can be successfully performed only when the characteristics of the signal are known, such as how many dominant frequencies are present in the time series.

If the characteristics of the signal are not known *a priori*, a good approach is to start with a general time-frequency analysis method such as the short-time Fourier transform or

complex Morlet wavelet convolution, and then to compute instantaneous frequencies on the known signal components.

8.5 Empirical mode decomposition

Empirical mode decomposition (EMD) is a nonlinear time-frequency decomposition method that does not require *a priori* knowledge of the frequency structure of the data, nor does it assume frequency stationarity, nor does it involve "template-matching" procedures such as the Fourier transform, convolution, or filtering.

The goal of EMD is to identify "intrinsic modes" that characterize the data. The modes are oscillatory (that is, they have temporally successive and roughly temporally equidistant peaks and troughs) but do not necessarily have a constant frequency over time. The oscillatory structure of the time series need not be known in advance, but can be "discovered" by the algorithm.

The code below generates a time series comprising several non-stationary frequency components.

```
signal=zeros(size(t));
for i=1:4
  f = rand(1,2)*10 + i*10;
  ff = linspace(f(1),f(2)*mean(f)/f(2),n);
  signal = signal + sin(2*pi.*ff.*t);
end

hz=linspace(0,srate/2,floor(n/2)+1);
x=fft(signal)/n;
subplot(211), plot(t,signal)
subplot(212), plot(hz,2*abs(x(1:length(hz))))
set(gca,'xlim',[0 60])
```

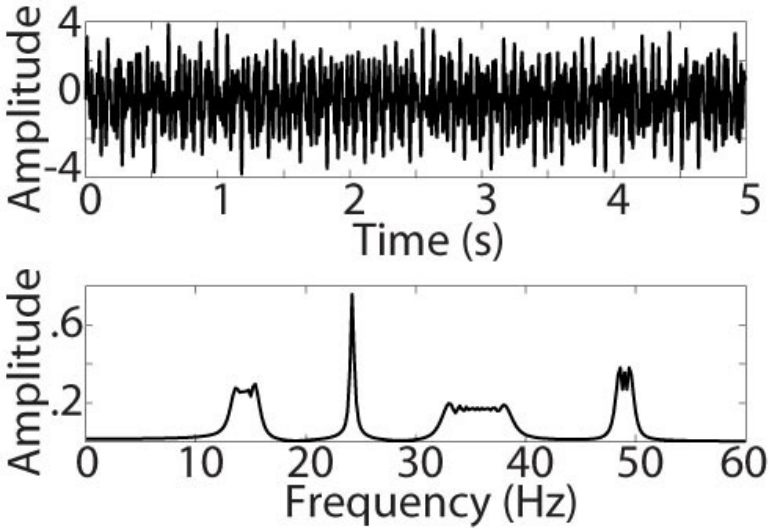

Figure 8.8 | A time series and its power spectrum, awaiting empirical mode decomposition.

To perform EMD, the following procedure is applied. First, the local minima and maxima of the time series are identified. Next, upper and lower envelopes are created by

interpolating across the local maxima (upper envelope) and local minima (lower envelope). These envelopes are then averaged together and subtracted from the original time series, creating a residual. If the difference between the time series and the residual is below a threshold, the residual is considered an intrinsic mode; it is then subtracted from the time series, and the process described above begins again in order to identify the next intrinsic mode. If the difference is above the threshold, the residual is subjected to the local minima/maxima procedure without subtracting the original time series. This "sifting" procedure continues until the threshold is reached.

A Matlab implementation of EMD can be inspected in the online code.

The figure below shows the spectral representation of the first four intrinsic modes resulting from EMD applied to the multi-chirp time series created above. EMD did a fairly good job at reconstructing the frequencies in the original time series, although this is not always the case. Other tests using different frequencies and amplitudes resulted in moderate or sometimes poor reconstructions. Furthermore, in some cases a single mode will contain multiple frequencies. That is, there is no necessary one-to-one mapping between EMD-derived intrinsic mode and the original sine wave component of the time series.

```
imfs = emdx(signal,4);
f = fft(imfs,[],2)/n;
plot(hz,abs(f(:,1:length(hz))).^2)
set(gca,'xlim',[0 60])
```

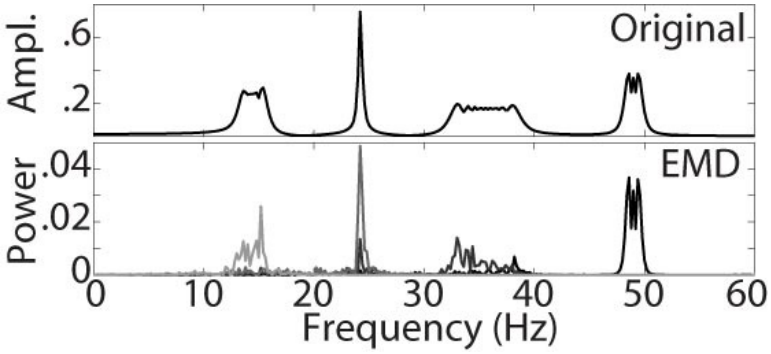

Figure 8.9 | The first four intrinsic modes of the EMD recover the original frequency structure, though not perfectly.

After intrinsic modes are extracted, the Hilbert transform can be applied to the time series of the intrinsic modes to obtain power and phase. From the phase values, instantaneous frequency can be computed as described earlier in this chapter. This procedure (EMD and Hilbert transform) is also known as the Hilbert-Huang transform.

The figure below shows the instantaneous frequencies estimated from the first four intrinsic modes. For comparison, the middle plot shows a time-frequency decomposition of the time series computed via complex Morlet wavelet convolution. The bottom plot shows instantaneous frequencies computed by first filtering the time series according to the center peaks of the Fourier transform. Code is not provided for this figure because reproducing it is one of the exercises at the end of this chapter.

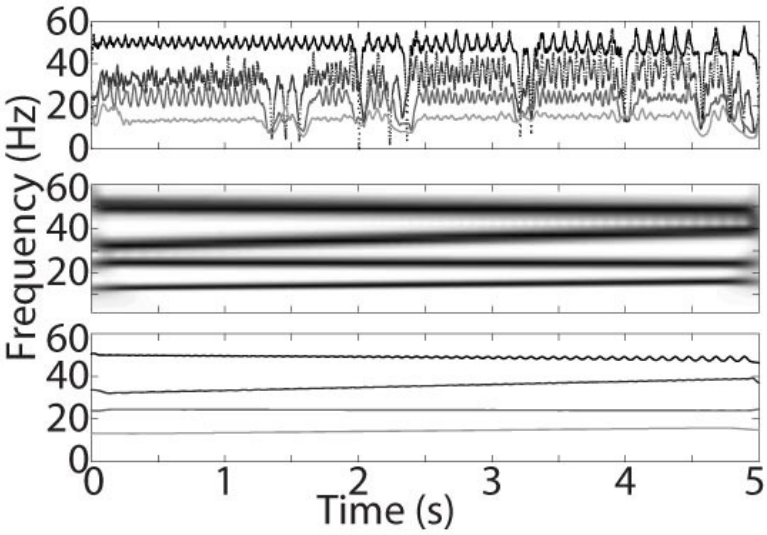

Figure 8.10 | Comparison of time-frequency decomposition via EMD (top) and complex Morlet wavelet convolution (middle). The bottom plot shows instantaneous frequencies after band-pass filtering around the spectral peaks of the time series.

In this case, estimating instantaneous frequencies from the band-pass filtered time series provided cleaner and more accurate results compared to estimating instantaneous frequencies from the intrinsic modes returned by EMD.

8.6 Exercises

1) Create a signal with two frequency non-stationarities, one between 10 and 13 Hz, and one between 30 and 35 Hz. Compute instantaneous frequency on the broadband time series. Next, apply appropriate band-pass filtering to isolate the two frequency ranges, and compute instantaneous frequencies again in these two frequency windows. Are the results from the broadband analysis interpretable?

Next, perform a time-frequency analysis on these data via complex Morlet wavelet convolution and plot the resulting time-frequency power. Finally, at each time point in the time-frequency result, extract and plot the frequency with the maximum power, separately in the ranges of 2-25 Hz and 25-40 Hz. What do the results indicate about the importance of band-pass filtering when analyzing a non-stationary multi-frequency time series? What is the relationship between instantaneous frequency as computed by the phase angle derivative, and the frequency with the maximum power from a wavelet convolution?

Code for this assignment is not printed here due to its length, but it can be inspected in the online Matlab code.

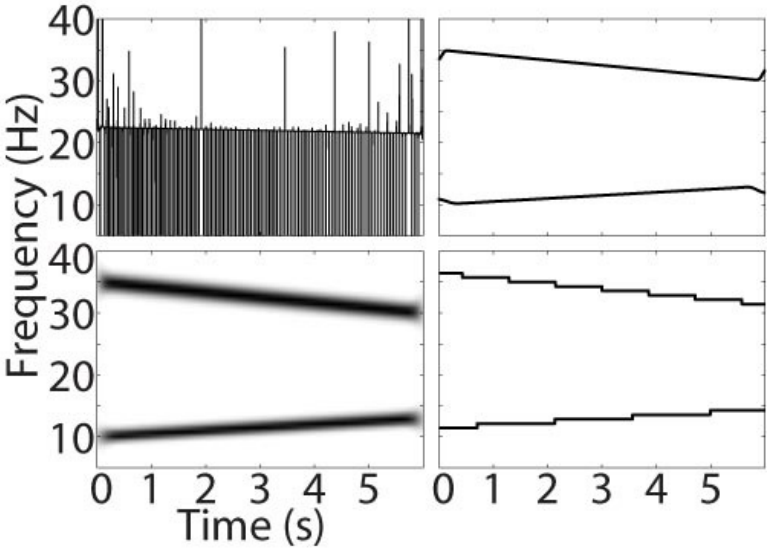

Figure 8.11 | The top left panel shows instantaneous frequencies computed from the broadband time series. The top right panel shows instantaneous frequencies computed from the band-pass filtered time series (convolution with a 11.5 Hz or a 32.5 Hz wavelet). The bottom left panel shows a time-frequency plot of the time series, created with complex Morlet wavelet convolution. The bottom right panel shows the frequencies from the time-frequency plot with maximal power in the ranges 2-25 Hz and 25-40 Hz.

2) Create Figure 8.10, starting from the code that generated Figure 8.9.

3) In Chapter 3, “pre-whitening” (computing the first temporal derivative) was discussed as a strategy for making time series data more stationary, and for acting as a low-pass filter. Compute instantaneous frequency on six time series, before and after pre-whitening: Three shown in Figure 8.1, and the same three time series but 20 Hz higher. Thus, there will be 12 results in total. What is the effect of pre-whitening on the accuracy of the estimated instantaneous frequencies, and does this depend on the frequency range?

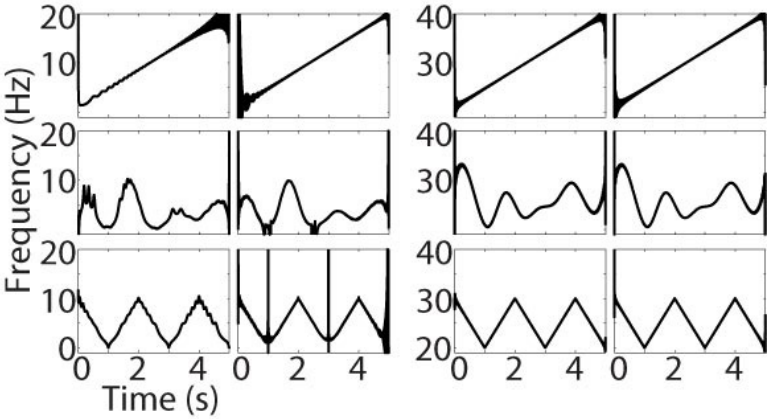

Figure 8.12 | Pre-whitening mainly affects estimation of very low-frequency dynamics. The left 6 plots show low-frequency simulations, and the right 6 plots show simulations 20 Hz higher. Within each subgroup of plots, the left column shows the instantaneous frequencies from the original time series, and the right column shows the instantaneous frequencies from the pre-whitened time series.

Chapter 9: Time series stationarity

Stationarity generally means that properties of a time series do not change significantly over time. There are several forms of stationarity, some of which are discussed in this chapter.

Stationarity can be achieved locally or globally. “Local” and “global” are relative terms, but generally, local stationarity refers to stationarity in a time window within the time series, while global stationarity refers to stationarity over the duration of the available time series. It is possible for there to be some local non-stationarities without significant global non-stationarity. Likewise, there can be global non-stationarities that change slowly enough such that local stationarity is preserved.

There are several reasons to analyze stationarity in a time series. It can be used as a diagnostic tool for determining time windows for time-frequency analyses, because stationarity should be maintained for the length of time used for the analysis (this would correspond to the time window for the short-time Fourier transform or the temporal width of the Gaussian taper in Morlet wavelet convolution). The effects of stationarity on the results of a Fourier transform were explored in Chapter 4.

Stationarity can also be an informative indicator of the dynamics and complexity of a system. A rigid, predictable, or simple system is likely to produce a stationary time series, whereas an adaptive, unpredictable, and complex system will likely produce a non-stationary time series.

9.1 Mean stationarity

The mean is the first moment, or statistical description, of a time series. To compute mean stationarity, the time series can be divided into non-overlapping time windows, and then the mean of each window can be computed. In Matlab, this can be achieved by reshaping the time series vector into a matrix with the number of rows corresponding to the number of time windows, or bins.

```
n=10000; nbins=200; % bins = windows
x=linspace(0,5,n) + randn(1,n);
y=randn(1,n);
timeMeanX = mean(reshape(x,n/nbins,nbins));
timeMeanY = mean(reshape(y,n/nbins,nbins));

subplot(221), plot(x)
subplot(222), plot(y)
subplot(223), plot(1:n/nbins:n,timeMeanX)
subplot(224), plot(1:n/nbins:n,timeMeanY)
```

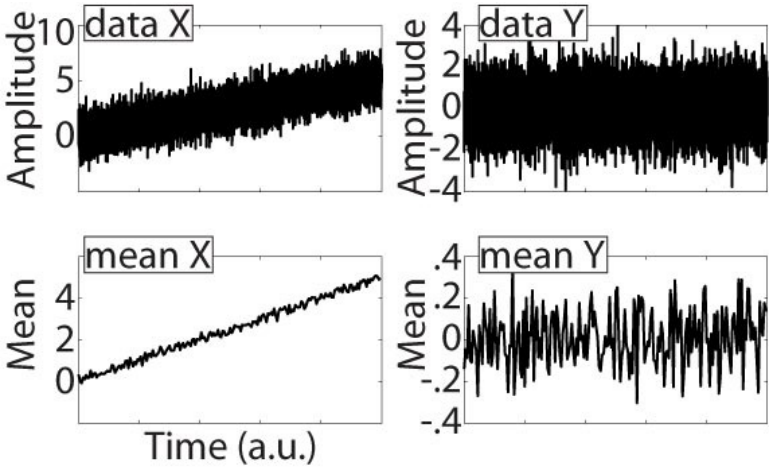

Figure 9.1 | Examples of mean-stationary and mean-non-stationary time series.

Visually, it is clear that time series **`x`** is mean-non-stationary and time series **`y`** is mean-stationary.

If a time series violates mean stationarity, there are several methods that are likely to help make the time series achieve mean-stationarity. The two easiest to implement are detrending (Matlab function **`detrend`**) and computing the derivative (this is shown in Figure 9.2 below). Another method to make a mean-non-stationary time series become mean-stationary is to compute a time series of the time-varying mean and subtract that mean-time-series from the original time series.

```
x=diff(x); x(n)=x(end);
timeMeanX = mean(reshape(x,n/nbins,nbins));
subplot(221), plot(x)
subplot(223), plot(1:n/nbins:n,timeMeanX)
```

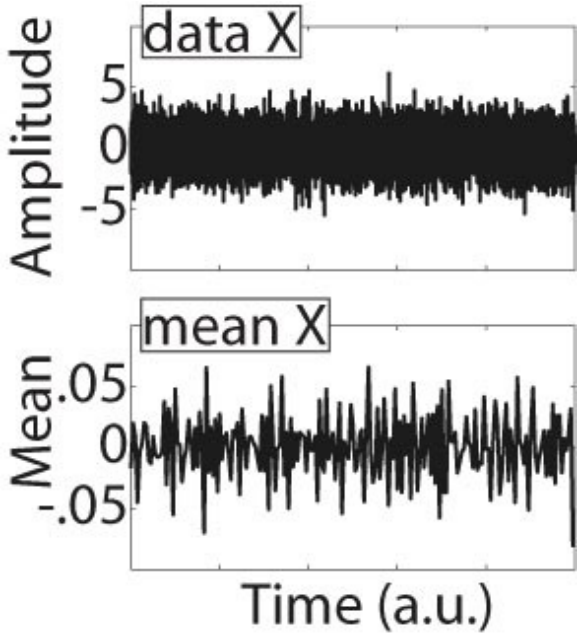

Figure 9.2 | Derivating away the non-stationarity.

9.2 Variance stationarity

The variance of a time series is its second moment, and can also vary over time. Similar to the mean, variance stationarity can be measured by binning the time series and computing the variance within each bin.

```
x=linspace(1,5,n) .* randn(1,n);
y=exp(-linspace(-1,1,n).^2) .* randn(1,n);

timevarX = var(reshape(x,n/nbins,nbins),[],1);
timevarY = var(reshape(y,n/nbins,nbins),[],1);

subplot(221), plot(x)
subplot(222), plot(y)
subplot(223), plot(1:n/nbins:n,timevarX)
subplot(224), plot(1:n/nbins:n,timevarY)
```

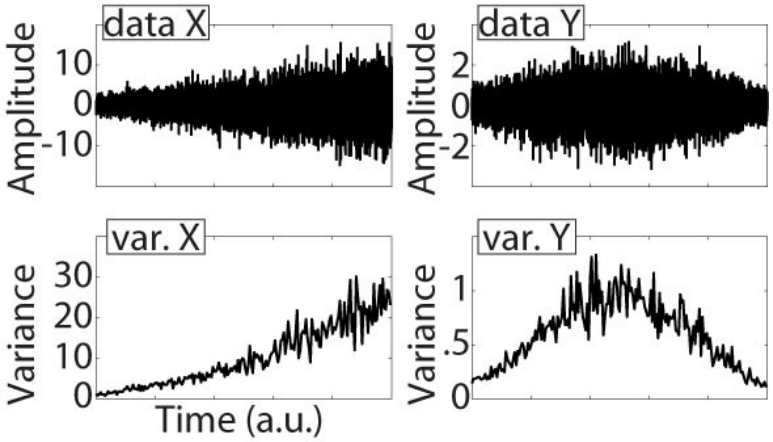

Figure 9.3 | Examples of variance-non-stationary time series.

Both time series appear to exhibit non-stationarities. However, **`x`** exhibits a linear non-stationarity over time, and **`y`** exhibits a nonlinear non-stationarity over time. Time series **`y`** also illustrates the distinction between local and global and local stationarity, because time series **`y`** is likely to be variance-stationary within a time window around the center of the time series.

Removing variance non-stationarity is more difficult compared to removing mean-non-stationarity. A simple approach is to normalize (z-score; subtract the mean and divide by the standard deviation) the time series within each bin separately. However, this will produce non-continuous edges between bins, and thus may be detrimental to subsequent analyses unless those analyses are performed separately on each bin.

Another method to help achieve variance stationarity is to scale the data by a weighting vector that varies from 0 to 1 according to the size of the variance. If the weighting vector is carefully constructed, the time series can be made variance-stationary. However, this will change the amplitude values, which may negatively bias subsequent analyses, such as time-frequency power.

This and the previous sections dealt with the first (mean) and second (variance) moments of a time series. Stationarity can also be computed for higher-order moments such as skewness (asymmetries in variances) or kurtosis (the shape of the distribution ranging from flat to pointy), or any other statistical property of a time series that can be computed

in windows of time, as shown above for mean and variance.

9.3 Phase and frequency stationarity

To achieve phase stationarity, the unwrapped phase angle time series should increase monotonically and at a similar rate over time. Phase stationarity could also be interpreted such that the first temporal derivative of the unwrapped phase angle time series should be positive and with similar values over time.

Because frequency can be defined as the first temporal derivative of phase (instantaneous angular velocity, as discussed in Chapter 8), phase stationarity and frequency stationarity are overlapping concepts.

```
srate=1000; t=0:1/srate:5; n=length(t);

freqTS = interp1(10*randn(5,1), ...
             linspace(1,5,n),'spline');
centfreq = mean(freqTS);
k = (centfreq/srate)*2*pi/centfreq;
y = sin(2*pi.*centfreq.*t + ...
        k*cumsum(freqTS-centfreq));
phases1 = diff(unwrap(angle(hilbert(y))));

y = sin(2*pi.*10.*t);
phases2 = diff(unwrap(angle(hilbert(y))));

subplot(211), plot(t(1:end-1),phases1)
subplot(212), plot(t(1:end-1),phases2)
```

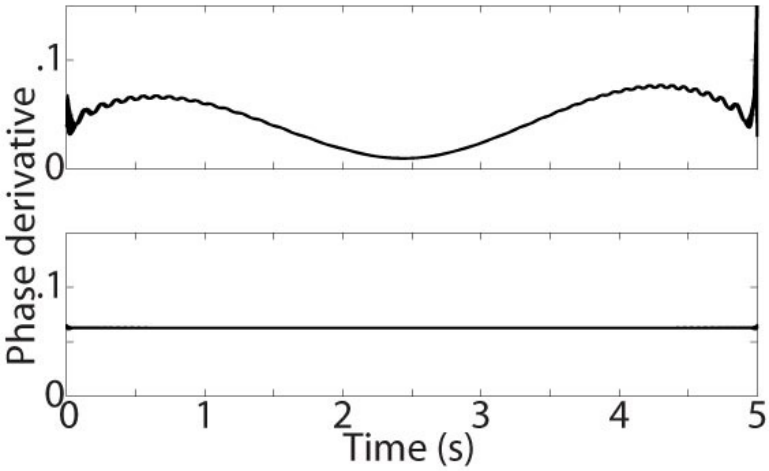

Figure 9.4 | Two time series that exhibit phase non-stationarity (top) and phase stationarity (bottom).

In the example above, the first time series appears to have phase/frequency non-stationarities, while the second time series is phase/frequency-stationary (the large brief fluctuations at the start and end of the time series reflect edge artifacts and can be ignored).

When considering the derivative, a few brief large-amplitude spikes (positive or negative) may occur even with small amounts of noise, as shown in Chapter 8.3. In these cases, the

signal should be filtered before interpreting phase stationarity. In addition to filtering before computing the phase angle derivative as discussed in Chapter 8, the median filter, which will be discussed in Chapter 10, is a useful filter for cleaning the phase angle derivative time series.

9.4 Stationarity in multivariate data

One of the primary measures of a multivariate system is its covariance matrix, an *N*x*N* (where N is the number of channels) matrix indicating the relationships amongst all pairs of channels. Thus, one indicator of multivariate stationarity is covariance stationarity.

In the code below, a multivariate time series will be created based on known covariance matrices. Brief non-stationarities will be created by concatenating three multivariate times series generated by three different covariance matrices.

```
n=100000; % N per section

% covariance matrices
v{1} = [1 .5 0;.5 1 0; 0 0 1.1];
v{2} = [1.1 .1  0; .1  1.1 0; 0 1.1 .1];
v{3} = [.8 0 .9; 0 1 0; 0 0 1.1];

% create multivariate signal (mvsig)
mvsig = zeros(0,length(v));
for i=1:length(v)
  c     = chol(v{i}*v{i}');
  tempd = randn(n,size(v{i},1))*c;
  mvsig = cat(1,mvsig,tempd);
end
```

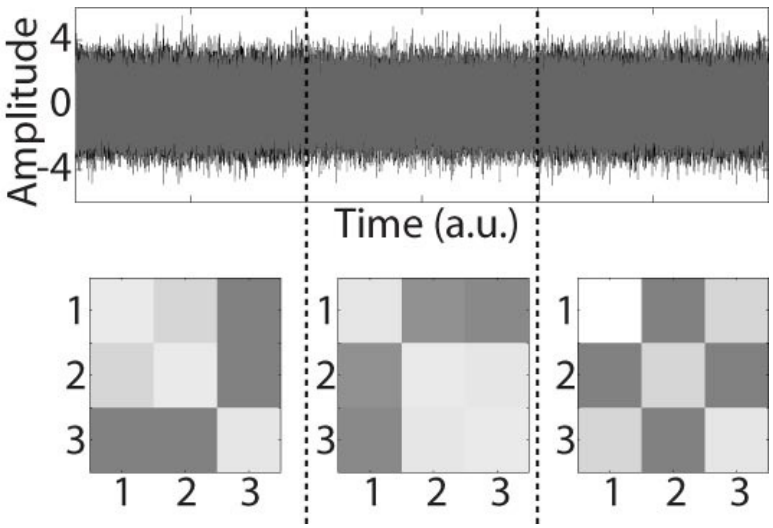

Figure 9.5 | Simulated multivariate time series, comprising three channels of data and defined by three covariance matrices. In the covariance matrices, the color of each box corresponds to the strength of the covariance between each pair of channels. The diagonals indicate the "auto-covariance," i.e., each channel's variance. Vertical dotted lines indicate the boundaries between covariances.

To measure the stationarity of the multivariate signal, the covariance matrix is measured in

non-overlapping time windows, and each covariance matrix is compared to the covariance matrix from the previous time window. The idea is that covariance stationarity will result in temporally successive covariances that are similar to each other; in contrast, covariance non-stationarity will result in successive covariances that are dissimilar.

This comparison can take the form of an Euclidean distance metric, in which the corresponding elements of the two covariance matrices are subtracted and squared, and all squared differences are summed (this can also be conceptualized as a sum of squared errors). This can be implemented in Matlab for two variables as **`sum( (a1-a2)^2 + (b1-b2)^2 );`**.

This produces a time series of "distances" between successive covariance matrices. In Figure 9.6, this metric is compared against the time-varying means and variances of each individual time series, using the same procedures as used earlier in this chapter.

```
nbins = 200;
nn = length(mvsig);
d3d = reshape(mvsig,nn/nbins,nbins,length(v));

for bini=1:nbins
  % 'tm' means temporary matrix
  tm = squeeze(d3d(:,bini,:));
  tCovar(bini,:,:) = (tm'*tm)/size(d3d,1);

  if bini>1
    y(bini)=sum( (reshape(tCovar(bini,:,:)-...
           tCovar(bini-1,:,:),1,[]).^2 ));
  end
end
plot(y)
```

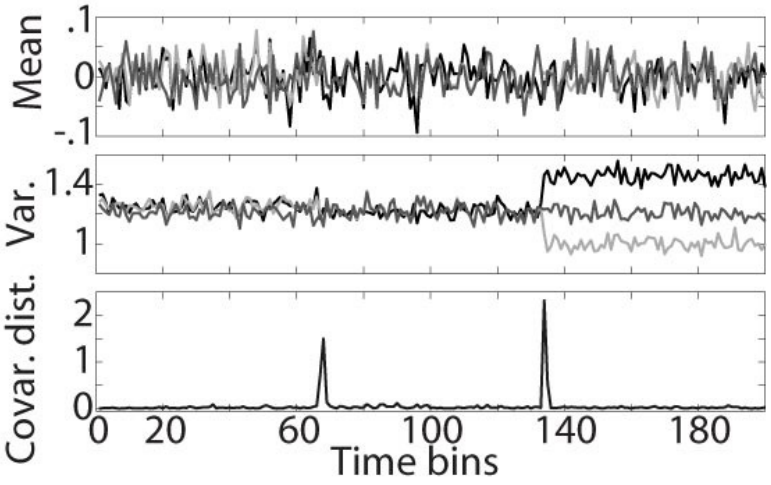

Figure 9.6 | Comparison of changes in stationarity of mean (top plot), within-channel variance (middle plot), and across-channel covariance (bottom plot). Only the Euclidean distance accurately identified both covariance non-stationarities.

An alternative metric to Euclidean distance is to compute the generalized eigenvalues between each two successive covariance matrices, and then take the "distance" as the ratio of the largest to the smallest eigenvalues. This generally produces the same results as the Euclidean distance, and is illustrated in the online Matlab code.

9.5 Statistical significance of stationarity

Sometimes it is visually obvious that a time series is non-stationary. Other times, however, it is difficult to know with certainty—simply by looking at a time series—whether it is stationary or non-stationary. In these cases, statistical evaluation is useful to help determine whether a potential non-stationarity should be deemed significant. This is particularly useful in the case of noisy data, in which some amount of non-stationarity may occur simply by chance.

There are several approaches to determining whether the observed non-stationarity may be significantly non-stationary. One is to test for a linear effect of the variable (e.g., mean or phase derivative) over time. If there is a statistically significant linear effect, one can say that there is significant non-stationarity. Such an approach, however, would fail to identify nonlinear patterns of stationarity, such as that shown in Figure 9.3. Another possibility is to fit polynomials to the stationarity time series and test whether the polynomial coefficients are statistically significant. Again, this procedure requires some knowledge or assumptions regarding the shape of the possible non-stationarity over time.

A better method to identify statistically significant non-stationarities should be sensitive both to linear and to non-linear effects without having to specify *a priori* the nature or shape of its relationship with time. Mutual information provides a useful general method in this case.

Mutual information quantifies the amount of information (entropy) that is shared between two time series. The two time series in this case are the time series of the stationarity variable (mean, variance, etc.) and time. An advantage of mutual information is that it can be used to determine a relationship between two variables regardless of whether the relationship is linear or non-linear, positive or negative.

Mutual information can be computed as the entropy of variables X and Y minus their joint entropy (here, X and Y refer, respectively, to the non-stationarity variable and time). Entropy is the amount of information contained in a time series, and is computed, in Matlab, as **`sum(p*log(p))`**, where **`p`** is the probability of observing a particular value in a particular bin. Binning can be done based on a number of statistical guidelines; the Freedman-Diaconis approach is used in the online Matlab code.

The mutual information result cannot be interpreted for statistical significance on its own. Instead, that result can be compared against a distribution of mutual information values that would be expected by chance, if there were no relationship between the stationarity variable and time. This distribution can be generated by randomly shuffling the time series and re-computing mutual information. The online Matlab code includes a function **`mutualinformationx`** that can be used to test the significance of a non-stationarity. Additional details of the procedure and how to interpret the result can be found by typing **`help mutualinformationx`** or by inspecting the code.

There are other methods to assess the statistical significance of non-stationarity over time that have been developed in the field of economics, most involving tests for a “unit root,” which is related to the presence of significant autoregressive coefficients (that is, current values of the time series can be predicted from previous values). Several of these tests are

included in the Matlab Econometrics toolbox, or the Octave Econometrics package.

9.6 Exercises

1) Generate a time series comprising a frequency-stationary sine wave at 10 Hz and a chirp from 12 to 30 Hz, over nine seconds. In four separate simulations, add noise that is variance-stationary and mean-non-stationary, variance-non-stationary and mean-stationary, and so on. Perform a time-frequency analysis via complex Morlet wavelet convolution, and plot the results. Do the different characteristics of noise differentially affect the results, and why is this the case?

```
% create noiseless signal
srate=1000; t=0:1/srate:9; n=length(t);
freqTS = linspace(12,30,n);
centfreq = mean(freqTS);
k = (centfreq/srate)*2*pi/centfreq;
basesig = sin(2*pi.*centfreq.*t + ...
  k*cumsum(freqTS-centfreq))+sin(2*pi*10*t);

% wavelet parameters
wtime = -2:1/srate:2;
Lconv = n+length(wtime)-1;
halfwavL=floor(length(wtime)/2);
nfrex = 50;
frex  = linspace(5,40,nfrex);
ncyc  = linspace(8,30,nfrex);

% add stationary noise
% (see online code for other simulations)
signal = basesig + randn(1,n)*2;

% FFT of signal
sigX = fft(signal,Lconv);

% perform wavelet convolution
tf = zeros(length(frex),length(t));
for fi=1:nfrex
  w = 2*( ncyc(fi)/(2*pi*frex(fi)) )^2;
  cmw = exp(1i*2*pi*frex(fi).*wtime) .* ...
        exp( (-wtime.^2)/w );
  cmwX = fft(cmw,Lconv);
  cmwX  = cmwX./max(cmwX);
  convres  = ifft( sigX.*cmwX );
  convres = convres(halfwavL:end-halfwavL-1);
  tf(fi,:) = abs(convres).^2;
end
contourf(t,frex,tf,40,'linecolor','none')
```

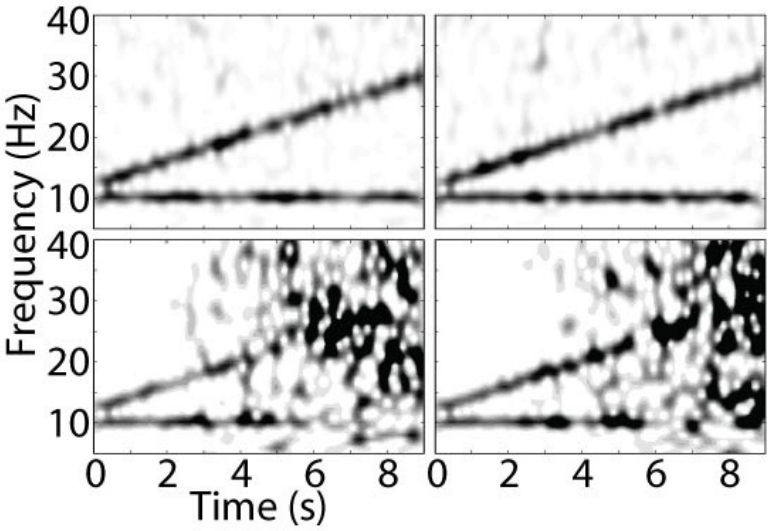

Figure 9.7 | Time-frequency representation of a multi-frequency signal, and the influence of stationary and non-stationary noise (noise amplitude of 2). The characteristics of the added noise are: mean-stationary, variance-stationary (top left); mean-non-stationary, variance-stationary (top right); mean-stationary, variance-non-stationary (lower left); mean-non-stationary, variance-non-stationary (lower right).

---

2) From the code in Chapter 9.4 for simulating multivariate time series, add a triangle-shaped frequency non-stationarity to one channel such that there is one triangle cycle for each of the three time windows with different covariances. Create a time-frequency power plot using any analysis method from this book (e.g., short-time Fourier transform, complex Morlet wavelet convolution, filter-Hilbert). Does the change in multivariate covariance affect the time-frequency results from this one channel? If so, how, and if not, why not?

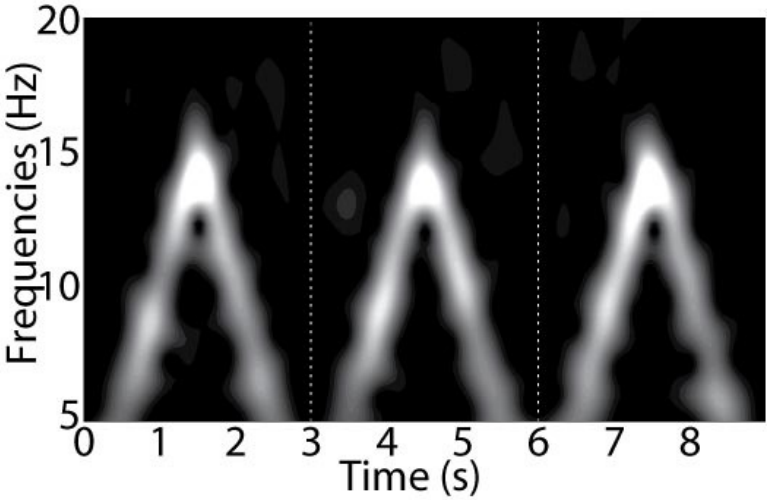

Figure 9.8 | Changes in channel covariance do not have a visually obvious effect on the time-frequency dynamics from one channel.

Chapter 10: Denoising time-domain data

There are many sources of noise that can contaminate signals, and there are many methods for attenuating or removing noise from time series data. The optimal method to attenuate noise depends on the characteristics of the noise and the signal. There is no one-size-fits-all method of reducing noise; it is necessary to inspect the data and to understand the sources and characteristics of the noise. This chapter provides an introduction to some of the common de-noising strategies. More advanced de-noising strategies, including template matching and blind-source separation techniques, are not discussed.

10.1 Filtering to attenuate noise

In some cases, noise has well-defined frequency characteristics and can be removed simply by filtering. This is the case, for example, with power line noise, which can have large amplitude but a frequency at exactly 50 or 60 Hz.

```
srate=1000; t=0:1/srate:3; n=length(t);
signal = .2*sin(2*pi*.8*t) + sin(2*pi*6*t);
noise = 100*sin(2*pi*50*t);
data = signal + noise;

dataX=fft(data);
hz=linspace(0,srate,length(t));
filterkernel = (1-1./(1+exp(-hz+40)));
dataX = dataX.*filterkernel;
data2 = real(2*ifft(dataX));
subplot(211), plot(t,data)
subplot(212), plot(t,data2), hold on
plot(t,signal,'r')
```

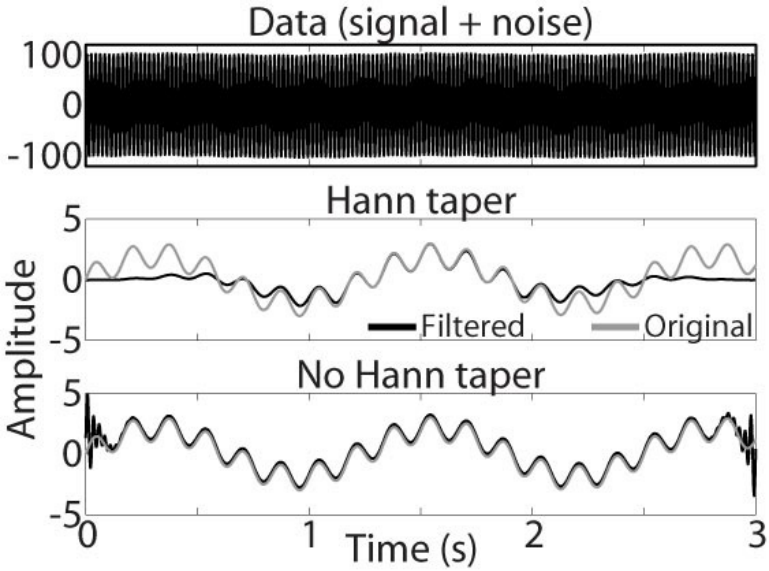

Figure 10.1 | High-amplitude, high-frequency noise (top panel: signal and noise) can be successfully removed with a low-pass filter (bottom two panels: filtered time series and noiseless original signal). Note the difference in y-axis scaling between the top and bottom two plots.

The middle plot of Figure 10.1 shows the results when applying a Hann taper before computing the Fourier transform to eliminate edge artifacts (this is not done in the code above, but can be inspected in the online Matlab code), and the bottom panel shows the results without applying any taper. This illustrates that the benefits of tapering before computing the Fourier transform usually but not always outweigh the costs in terms of loss of signal.

Noise may not always have such a narrow peak. But if the noise has frequency characteristics that are concentrated in a range (such as >100 Hz or <4.5 Mhz), and if there is little or no signal of interest in these frequency ranges, a low-pass or high-pass filter can effectively eliminate noise.

10.2 Moving-average filter

The moving-average filter (also sometimes called a running-mean filter) is intuitive, easy to implement, and works well when the noise is random and with small amplitude relative to the signal. A moving-average filter involves reassigning each data point to be the average (arithmetic mean) of surrounding data points. There is one parameter for this filter, *d*, which is the number of surrounding data points used to compute the mean (*d* is for distribution size).

```
noise = randn(size(signal));
data = signal + noise;
d=9; % 19-point mean filter
dataMean=zeros(size(data));
for i=d+1:length(t)-d-1
  dataMean(i) = mean(data(i-d:i+d));
end

subplot(211), plot(t,data)
subplot(212), hold on
plot(t,dataMean,'b')
plot(t,signal,'r')
```

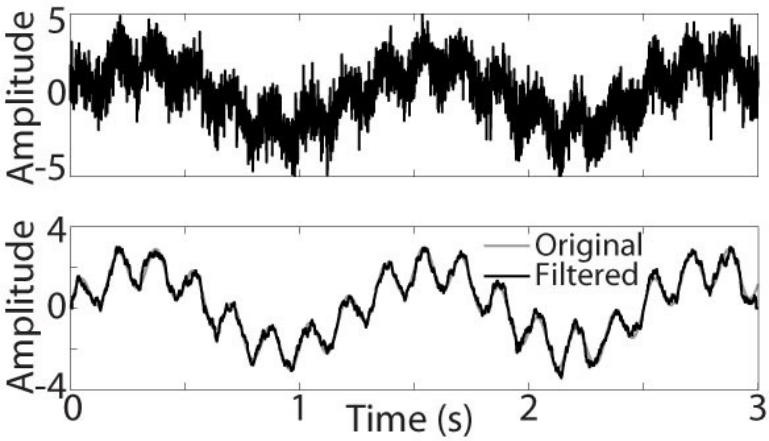

Figure 10.2 | A moving-average filter is effective for attenuating random noise if the noise values are positive and negative relative to the signal and with no outliers.

In the code above, the **`d`** parameter is set to “9” but this actually produces a **`d`** of 19. The

reason is that this implementation computes the mean from each time point minus nine points to plus nine points. Thus, the total number of points used to compute the mean is 9+9+1 (the +1 is for the center time point). There is no rule for setting the **d** parameter. It should be large enough to average over noise without being so large as to average over signal. In practice, it is best to try using several different **d** parameters on sample data to determine empirically the optimal parameter for each dataset.

Recall the discussion in Chapter 7.4 regarding causal vs. non-causal filters. The moving-average filter implemented above is a non-causal filter because the filter applied to each time point incorporates data both from previous and from future time points. The moving-average filter could be made causal if only previous time points were considered when computing the mean. However, this will produce a phase shift (time-lagged response) and is unnecessary in off-line analyses.

10.3 Weighted moving-average filter

In the moving-average filter, all surrounding data points are weighted equally. An adaptation of this method is to compute a weighted mean, such that each data point is reassigned to be a weighted average of surrounding data points.

A frequently used weighted moving-average filter is the Gaussian filter, in which the surrounding data points are weighted according to their distance away from each center data point. In practice, a Gaussian filter can be implemented using convolution via frequency domain multiplication. Because a Gaussian has a negative exponential shape in the frequency domain (that is, it starts high and falls rapidly), convolving the data with a Gaussian is effectively a weighted moving-average low-pass filter.

```
noise = randn(size(signal));
data = signal + noise;
d=9; % 19-point mean filter
% Gaussian, width=2
gausfilt = exp(-(-d:d).^2/4);
halfGausL = floor(length(gausfilt)/2);

% convolution
Lconv = length(gausfilt)+length(t)-1;
convres = ifft( fft(data,Lconv).* ...
                fft(gausfilt,Lconv) );
dataGaus = convres(halfGausL:end-halfGausL-1);

subplot(211), plot(t,data)
subplot(212), hold on
plot(t,dataGaus/sum(gausfilt),'b')
plot(t,signal,'r')
```

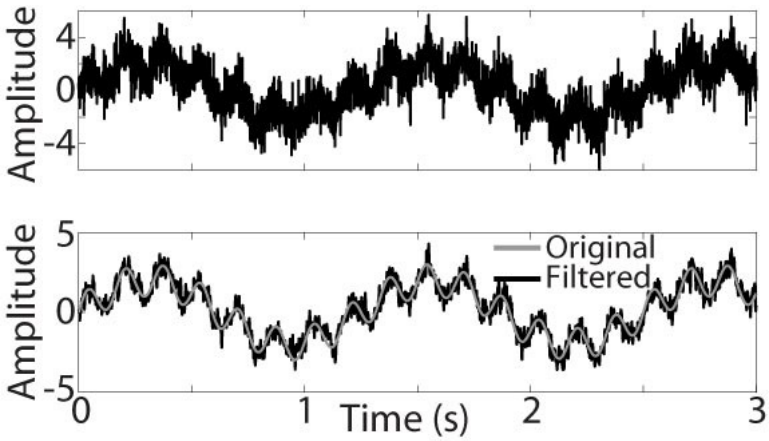

Figure 10.3 | A weighted moving-average filter. Here, the weighting is set as a Gaussian distance away from each center time point.

In this example, the running-average filter out-performs the Gaussian-weighted filter. The performance of the Gaussian-weighted moving-average filter can be improved by modifying its parameters. This will be explored further in exercise 3 at the end of this chapter.

10.4 Moving-median filter

The moving-median filter is similar to the moving-average filter, except that the median is used instead of the mean. The median, like the mean, is a measure of the central tendency of a distribution of numbers; the median, however, is taken as the middle number, that is, the number that divides the distribution into two equal-sized halves.

The moving-median filter is particularly useful if there are a few noise spikes of very large amplitude. In these cases, the mean will be driven by the noise spike whereas the median will be insensitive to the spike. Because the median is a non-linear estimation of the central tendency of a distribution, the median filter is a non-linear filter.

Similar to the mean filter, the main parameter of the median filter is *d*, the distribution size, or the number of successive points used to compute the median.

```
noise = zeros(size(t));
% necessity of 'find' is version-dependent
noise(find(isprime(1:length(t)))) = 100;
data = signal + noise;

d=9; % 19-point median filter
[dataMed,dataMean]=deal(zeros(size(data)));
for i=d+1:length(t)-d-1
  dataMed(i)  = median(data(i-d:i+d));
  dataMean(i) = mean(data(i-d:i+d));
end

subplot(211), plot(t,data)
subplot(212), plot(t,dataMed), hold on
plot(t,dataMean,'r'), plot(t,signal,'k')
```

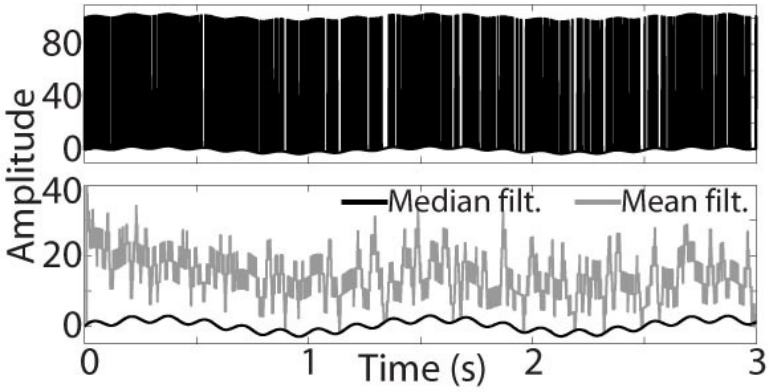

Figure 10.4 | When the noise contains unusually large values (outliers), the moving-median filter will outperform the moving-average filter.

It is clear that the moving-average filter performs poorly in this case. The moving-median filter performs well, which can be better seen by setting the y-axis limits in the lower subplot to [-1.5 1.5]. The moving-median filter and the original signal (before adding noise) are nearly perfectly overlapping.

In the code above, the median was computed once, using one value for the **`d`** parameter. It is also possible to apply the median filter several times, each time using a different **`d`** parameter. This is useful for time series with considerable noise. After the median is computed several times, the resulting median-filtered time series can be averaged together. This procedure was not necessary in the above example, because one **`d`** parameter was sufficient to eliminate the noise.

```
n_order=6; d=round(linspace(5,19,n_order));
dataMedFull = zeros(n_order,length(t));
for oi=1:n_order
  for i=d(oi)+1:length(t)-d(oi)-1
    temp = sort(data(i-d(oi):i+d(oi)));
    dataMedFull(oi,i) = ...
            temp(floor(length(temp)/2)+1);
  end
end
dataMed = mean(dataMedFull);
```

Instead of using the function **`median`**, the code above computes the median directly by sorting the values and then selecting the middle value. This implementation is faster than using the function **`median`**.

For very long time series, the median filter can be slow because it involves many floating point operations. If you are using Matlab and have the Matlab compiler installed, you can use an implementation of a fast median included in the “nth_element” toolbox, which is available for free download from the Matlab file exchange.

Finally, note that in the implementation above, the variable **`dataMedFull`** contains zeros towards the beginning and the end of the time series. This is due to the lack of data before the start of the median computation, as defined by the parameter **`d`**. The mean over median filter orders (the final line of code) will thus be weighted by zeros. If the time series contains sufficient buffer time (see Chapter 3.8), the zeros will not adversely affect

the time ranges of interest. Alternatively, the variable **`dataMedFull`** can be initialized as **`nan`**'s (not-a-number), and averaging can be done with the Matlab function **`nanmean`** (included in the Signal Processing toolbox in Matlab, or the Signal package in Octave), which computes the mean ignoring nan's.

10.5 Threshold-based filters

Depending on the characteristics of the signal and of the noise, it may be unwise to compute the mean or median at each time point. Instead, a threshold can be selected, and the filter will apply only to points exceeding the specified threshold. In the example below, inspection of the data suggests that a threshold of two standard deviations above the median provides a useful distinction between signal and noise. The mean and median filters will be computed only on data points exceeding that threshold; all other data points will be untouched.

```
signal = .5*sin(2*pi*60*t) + sin(2*pi*6*t);
noise = zeros(size(t));
noise(find(isprime(1:length(t)))) = 100;
data = signal + noise;
d=9; % 19-point median filter
dataMed=data;
points2filter = ...
     find(data>2*std(data)+median(data));

for i=1:length(points2filter)
  centpoint = points2filter(i);
  dataMed(centpoint) = median( ...
     data(max(1,centpoint-d): ...
          min(length(data),centpoint+d)));
end
subplot(211), plot(t,data)
subplot(212), plot(t,dataMed), hold on
plot(t,signal,'k')
set(gca,'xlim',[0 .5])
```

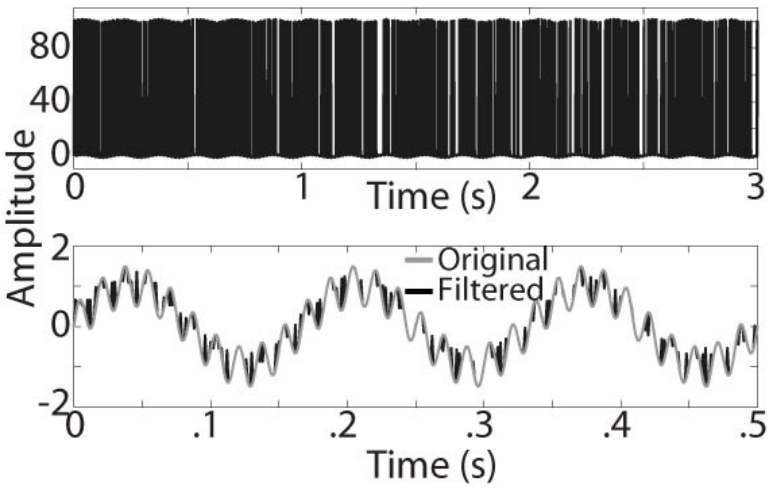

Figure 10.5 | Threshold-based filters are exclusive.

Some residual noise remains, but the signal is fairly clean with minimal disruption of the

high-frequency dynamics. This can be compared with the performance of the threshold-less moving-median filter introduced in the previous section, and implemented in the code below.

```
dataMed2 = zeros(size(data));
for i=d+1:length(t)-d-1
    dataMed2(i)  = median(data(i-d:i+d));
end
plot(t,dataMed2,'r')
```

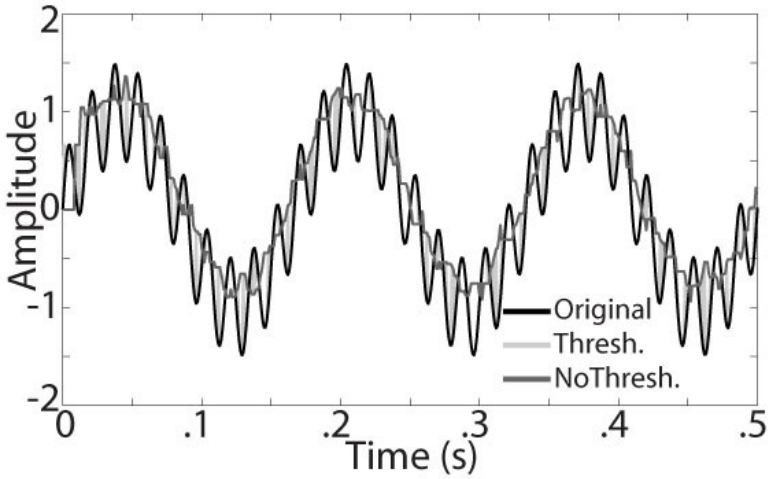

Figure 10.6 | Comparison of original signal (black line), threshold-based median filter (light gray), and non-thresholded median filter (thin dark gray line).

In this example, the standard median filter nearly entirely removed the high-frequency component of the signal.

If the **`d`** parameter is set to 3 instead of 9, both filters perform fairly well.

The important lesson here is that the optimal de-noising strategy, as well as optimal parameters for each algorithm, depend strongly on the characteristics of both the signal and the noise.

Because the filter is applied only to some time points, threshold-based filters can introduce non-linearities and can reduce the smoothness of the time series. This may have a negative bias for subsequent analyses. As with all de-noising strategies, the results should be closely inspected to make sure that too much signal has not been removed along with the noise.

10.6 Polynomial fitting

In Chapter 10.1, it was discussed that band-pass filtering can be a useful de-noising strategy. However, this relies on the components of the time series being fairly well localized in the frequency domain.

If the signal does not have a “clean” frequency representation but still fluctuates more slowly than the noise, a polynomial function may be a useful de-noising strategy. Fitting polynomials is a regression-like procedure in which the data are predicted from coefficients that scale a predictor variable. Below is a simple example to illustrate how

polynomial fitting works.

```
x=1:20; y=2*x+randn(size(x));
p=polyfit(x,y,1);
plot(x,y,'o-'), hold on
plot(x,p(2)+p(1)*x,'r*-')
```

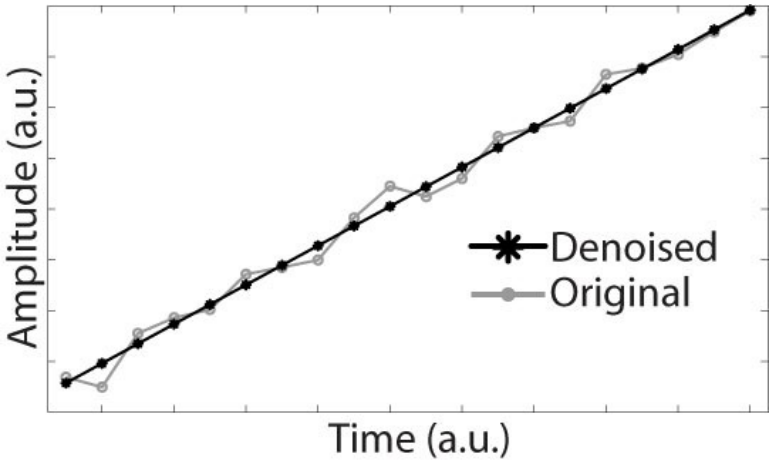

Figure 10.7 | Denoising with polynomials, part I.

The variable **p** contains n+1 numbers, where n is the polynomial order (in the above example, set to "1"). For a polynomial of order one, the expansion reduces to the perhaps more familiar linear expression "y=mx+b" (in Matlab: **y=p(1)*x+p(2)**). For higher order polynomials, the coefficients are expanded as in the following example, using a polynomial order of three.

```
y = p(4) + p(3)*x + p(2)*x.^2 + p(1)*x.^3;
```

In practice, for a larger polynomial order, or when the order can change, it is useful to use the function **polyval** to evaluate the polynomial expansion.

```
srate=1000; t=0:1/srate:5;
signal = interp1(0:5,randn(6,1),t,'spline');
noise = 3*randn(size(t));
data = signal+noise;

polyorder = 6;
p = polyfit(t,data,polyorder);
dataPolyFit = polyval(p,t);

subplot(211), plot(t,data)
subplot(212), plot(t,dataPolyFit), hold on
plot(t,signal,'r')
```

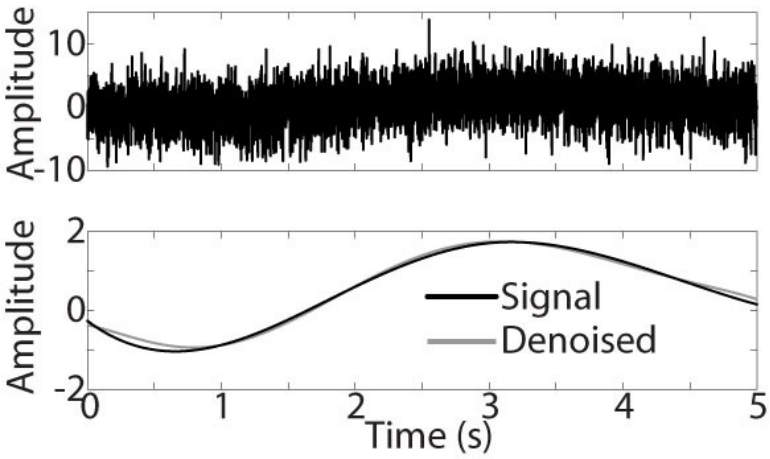

Figure 10.8 | Denoising with polynomials, part II.

The polynomial order should not be set higher than necessary. Models with a large number of parameters require a lot of data for reliable parameter estimation. The polynomial order must roughly match the changes in the signal; under-fitting and over-fitting will produce suboptimal results by being insensitive to the signal changes or by being too sensitive to noise. This can be seen by changing the **`polyorder`** parameter above, for example to "2" or to "40". Sometimes the fit to the data is so poor that Matlab will issue a warning.

In the above example, the signal contained slowly changing features while the noise contained rapidly changing features. Polynomial fitting can also be used in the opposite case: When the signal contains rapidly changing features and the noise contains slowly changing features such as drifts. In this case, de-noising involves subtracting the fitted time series to obtain the signal.

```
plot(t,data-dataPolyFit,'k')
```

Thus, polynomial fitting can be a useful de-noising strategy when the signal and noise can be distinguished by rapidly vs. slowly changing time series features, even if they do not have well-defined frequency characteristics.

10.7 From 1 to n-dimensions

This chapter deals only with 1-dimensional time-domain data. However, the de-noising strategies shown here can also be applied to higher-dimensional data, such as 2-dimensional images, or higher-dimensional matrices. Below is one example of using the median filter on an image.

```
image
signal = get(findobj(gcf,'type','image') ...
                     ,'CData');
data = signal + 1000*reshape( ...
        isprime(1:numel(signal)),size(signal));
d=1; thresh=2*std(data(:))+median(data(:));
dataMed = data;
for i=d+1:size(data,1)-d-1
  for j=d+1:size(data,2)-d-1
```

```
    if data(i,j)>thresh
      temp = data(i-d:i+d,j-d:j+d);
      dataMed(i,j) = median(temp(:));
end, end, end

subplot(131), imagesc(signal)
  set(gca,'clim',[3 30]), axis image
subplot(132), imagesc(data)
  set(gca,'clim',[3 30]), axis image
subplot(133), imagesc(dataMed)
  set(gca,'clim',[3 30]), axis image
```

Figure 10.9 | Threshold-based median filter, in 2D.

In this example, the median filter successfully removed the “salt” noise from the image without a significant loss of detail.

10.8 Exercises

1) Inspection of the right-most image in Figure 10.9 reveals that the threshold-based median filter failed to clean the noise around the edges. Without changing any filter parameters, modify the code in Chapter 10.7 to remove noise around the edges.

```
% This solutions requires only minor
% modifications of the code presented above.
% The first modification is to get the size
% of matrix 'data'
dims = size(data);

% Next, the line 'temp=data…' will be
% replaced (note the addition of
% 'max' and 'min'):
temp = data( max(1,i-d):min(dims(1),i+d) , ...
             max(1,j-d):min(dims(2),j+d));

% Additionally, the 'for' loops are modified:
for i=1:1:size(data,1)
    for j=1:size(data,2)

% Consider what happens when i=1: i-d is
% negative, which will produce an error for
% matrix indexing. Hence the index is
% whatever is larger: 1 or i-d. Same idea
% for 'min' at the end of the image boundary.
```

Figure 10.10 | An improved threshold-based median filter in 2D.

---

2) From the two simulations in exercise 1 of Chapter 9 that included variance non-stationarities, apply a de-noising strategy to minimize the non-stationarities (after the noise has already been added), and then re-compute the time-frequency power plots. Did the de-noising successfully attenuate the noise, and did this come at the expense of reduced signal?

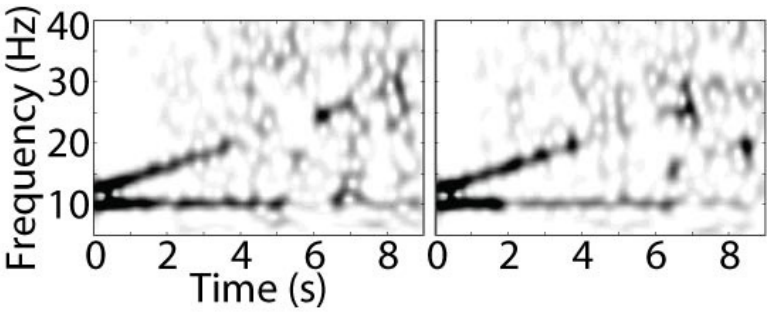

Figure 10.11 | Because the increased variance is broadband, it is difficult to selectively attenuate the noise while preserving the signal. Thus, although noise can be diminished, information from the signal is also lost to some extent.

3) A Gaussian has a negative exponential shape in the frequency domain. What is the frequency shape of the moving-average filter? In the same plot, show the frequency response of the moving-average and Gaussian-weighted-moving average filters that were used in Figures 10.2 and 10.3. Based on this plot and on the knowledge learned from Figures 10.2 and 10.3, modify the parameters of the Gaussian-weighted moving-average filter so that it performs comparably (or better) compared to the moving-average filter on the double-sine wave in Figure 10.3.

Chapter 11: Introduction to multivariate time series analysis

Many systems contain distinct but interacting components; it may not be possible to understand the system dynamics when analyzing data from only one channel. This is the case, for example, with EEG (brain electrical activity): Dozens or hundreds of electrodes are placed around the head, and the brain's electrical activity is measured simultaneously from all electrodes. Each electrode provides both unique and overlapping information compared to other electrodes.

Generally speaking, there are two approaches to analyzing multivariate time series data. The first approach is to perform univariate analyses, such as those shown in this book, on each channel separately, and then characterize the population activity. This approach would lead to results such as "80% of channels had a spectral peak at 23.4 Hz" or "all channels had linear increasing chirps between 250 Hz and 400 Hz, and noise below 100 Hz."

A second approach to analyzing multivariate time series data is to perform analyses that reflect interactions amongst channels. In contrast to the first approach, multivariate analyses are those that can be performed only on multichannel data, and those for which the results will differ depending on how many channels are included in the analyses.

Multivariate analyses are generally more complex compared to univariate analyses. Two multivariate analyses will be introduced here, a bivariate analysis (for pairs of channels), and a multivariate covariance analysis (for more than two channels).

11.1 Spectral coherence

Spectral coherence is computed between two channels. It is analogous to correlation, but there are two noteworthy distinctions between correlation and spectral coherence: Coherence cannot be negative and it is computed in the frequency domain, thus providing frequency specificity.

Spectral coherence is computed by taking the Fourier transforms of each channel, and computing the conjugate of those transforms. The conjugate of a complex number is the same number but with the sign of the imaginary part flipped. Thus, the conjugate of the complex number $4+i9$ is $4-i9$.

An aside on the use of the complex conjugate in time-frequency analysis: Although power is often computed as amplitude squared (**`abs(fft(x)).^2`**), it can also be implemented by multiplying the Fourier coefficients by their conjugates (**`fft(x).*conj(fft(x))`**).

Spectral coherence involves multiplying the Fourier coefficients of one time series by the conjugate of the Fourier coefficients of a second time series, and their joint magnitude is taken as the strength of the coherence between them.

However, simply multiplying Fourier coefficients is not an appropriate measure of coherence. This is because it is influenced by the amplitude of each time series, regardless of the actual relationship of the two time series to each other.

An interpretable measure of spectral coherence must be scaled by the amplitudes of the individual spectra. The multiplication of the Fourier coefficients of the two spectra is thus taken as the numerator in a ratio. The denominator is the product of the amplitudes of each spectrum individually. This scaled ratio is the measure of spectral coherence, and it depends only on the relationship between the two time series, not on the amplitude of either time series. Spectral covariance varies between 0 (no covariance at all) and 1 (perfect covariance).

To simulate two coherent signals, one is created by incorporating the signal from the other.

```
srate=1000; t=0:1/srate:9; n=length(t);

% create signals
f = [10 14 8];
k1=(f(1)/srate)*2*pi/f(1);
sigA = sin(2*pi.*f(1).*t + k1* ...
    cumsum(5*randn(1,n))) + randn(size(t));
sigB = sin(2*pi.*f(2).*t + k1* ...
    cumsum(5*randn(1,n))) + sigA;
sigA = sigA + sin(2*pi.*f(3).*t + ...
              k1*cumsum(5*randn(1,n)));

% show power of each channel
hz = linspace(0,srate/2,floor(n/2)+1);
sigAx = fft(sigA)/n;
sigBx = fft(sigB)/n;

subplot(221)
plot(hz,2*abs(sigAx(1:length(hz)))), hold on
plot(hz,2*abs(sigBx(1:length(hz))),'r')

% spectral coherence
specX = abs(sigAx.*conj(sigBx)).^2;
spectcoher = specX./(sigAx.*sigBx);

subplot(222)
plot(hz,abs(spectcoher(1:length(hz))))
```

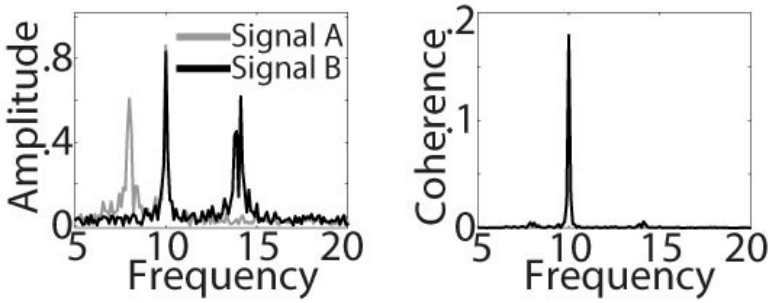

Figure 11.1 | Spectral power and spectral coherence.

Both signals contain power at two frequencies (signal A: 8 Hz and 10 Hz; signal B: 10 Hz

and 14 Hz). However, the coherence plot shows a peak only at 10 Hz. This indicates that the two signals have both unique and shared properties.

11.2 Principal components analysis

Principal components analysis (PCA) is a multivariate analysis in which weights are computed for each channel, and the weighted combination of all channels reflects commonalities across the channels. These weighted combinations are called principal components or "modes." The components explain all of the variance of the data and are constructed such that they are uncorrelated with each other.

PCA is often used as a data-reduction technique for multivariate time series with inter-correlated channels. For example, a group of 10 channels might be accurately characterized by only 3 principal components.

To compute PCA, a covariance matrix is first required. A covariance matrix is computed by multiplying a multivariate time series by its transpose, and then dividing by N-1, where N is the number of time points. The mean should be subtracted from each channel separately before computing the covariance matrix, otherwise the first component of the PCA will reflect the mean time series value.

The code below will generate covarying time series using the method introduced in Chapter 3.9, compute its covariance matrix, and then display the covariance matrix as an image.

```
% covariance of data
v=rand(10); c=chol(v*v');
n = 10000; % n time points
d = randn(n,size(v,1))*c;

% subtract mean and compute covariance
d = bsxfun(@minus,d,mean(d,1));
covar = (d'*d)./(n-1);
imagesc(covar)
```

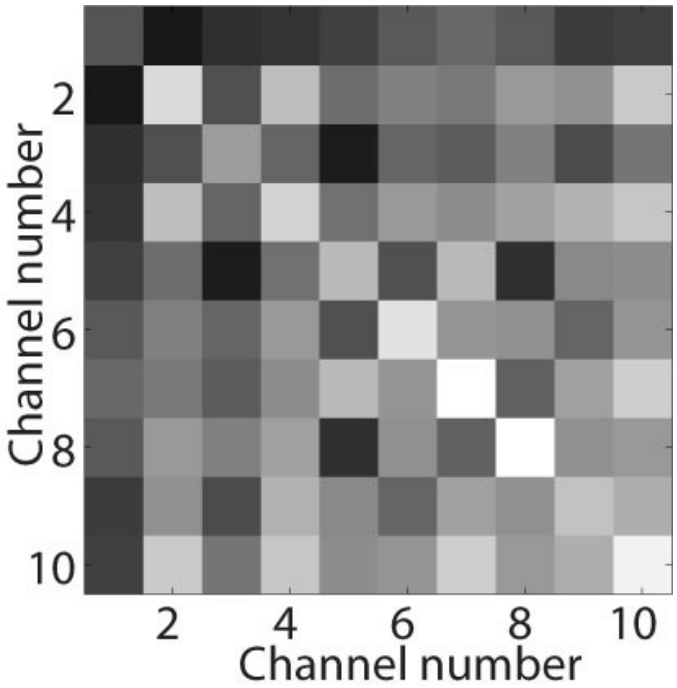

Figure 11.2 | Covariance matrix.

After the covariance matrix has been obtained, the PCA is computed as the eigenvalue decomposition of that matrix (identical or nearly identical results can also be obtained from a singular-value decomposition of the covariance matrix).

The eigenvalue decomposition returns two matrices, the eigenvectors and the eigenvalues. The eigenvectors provide weights per channel that, when multiplied by the channel time series, provide the principal components.

The eigenvalues reflect the “strength” along each PC, and can be extracted from the diagonal of the matrix and then converted to percent variance accounted for. Eigenvalue plots are useful to determine how many large components are present in the data. In the above simulated data, for example, the first component accounts for over 70% of the variance, while the remaining components each account for less than 10%.

In Matlab, the components are sorted according to how much variance of the covariance matrix they account for. The sorting is ascending, meaning that the smallest components are listed first. Some people find this “backwards” and prefer to re-sort the components to be largest-first. This is shown in the code below.

```
% compute PCA and eigenvalues (ev)
[pc,ev] = eig(covar);

% re-sort components
pc = pc(:,end:-1:1);
% extract eigenvalues and covert to %
ev = diag(ev);
ev = 100*ev(end:-1:1)./sum(ev);

plot(ev,'-o')
```

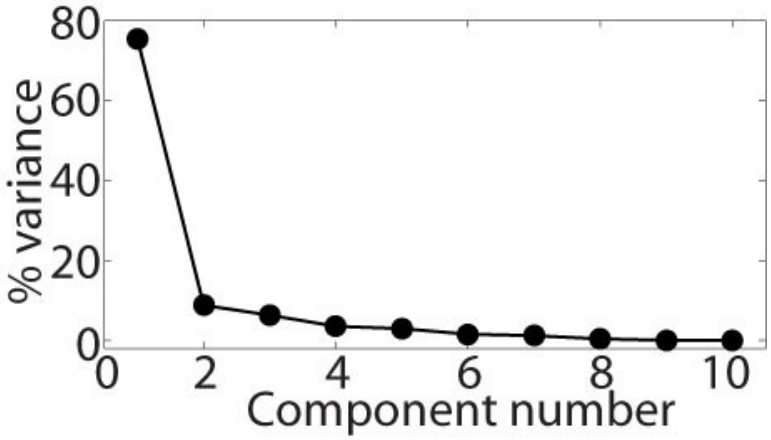

Figure 11.3 | Eigenvalues, converted to percent variance accounted for.

The PCA variable **`pc`** contains the weights that map the channel activity to each component. The time course of each PC can be obtained by multiplying the weights by the channel data. This time course is a weighted combination of all channels, where the weights are defined according to the multivariate covariances.

The PC time course can be treated like a regular time course, to which any analysis described in this book can be applied.

```
subplot(211), plot(d)
subplot(212)
plot(pc(:,1)'*d'), hold on
plot(pc(:,2)'*d','r')
```

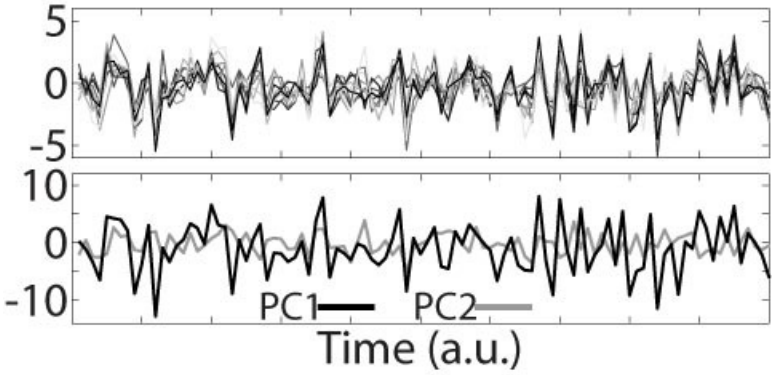

Figure 11.4 | Time course of the first 100 time points of the channel data (top) and the first two PCs (bottom). Note that the first PC captures much of the variance common across channels.

Chapter 12: Concluding remarks

12.1 What is signal and what is noise?

In this book (and in many educational situations), signal and noise are easily dissociable: The signal is created first, and thereafter the noise, often defined by randomly generated numbers, is added. Unfortunately, however, many time series in the real world are more complex, and it is not always so easy to separate the signal from the noise.

What is the difference between signal and noise? It is not a question of what can be interpreted and what cannot be interpreted. In the field of neuroscience, for example, scientists continue to be puzzled over exactly how to interpret brain electrical time series, and yet there is no ambiguity about how to interpret 50/60 Hz electrical line noise that contaminates unshielded recordings. It is also not the case that the signal contains information while the noise contains no information. With radios, for example, frequencies corresponding to unwanted stations can be considered "noise" that needs to be eliminated when tuning into the desired station.

Instead, the critical difference between signal and noise is that the signal is relevant for the current application, while the noise is irrelevant for the current application. What is considered relevant vs. irrelevant, and, thus, what is considered "signal" and what is considered "noise," in turn is constrained by current knowledge, technological limitations, and assumptions about the system under investigation. Perhaps some of what is now considered noise will in the future be considered signal.

Consider the famous discovery in the 1960's of residual microwave background radiation in the universe. This "signal" was initially thought to be "noise" by Penzias and Wilson, who were trying to detect weak radio waves from satellites. Realizing that this "noise" was not noise at all, but in fact was a direct measurement of the radiation resulting from the Big Bang, was one of most important findings of 20th century cosmology, and was a crucial piece of evidence that showed that our universe had a beginning and was expanding.

On the other hand, despite the relevance of this finding for theories of the origin and fate of the universe, this background radiation is still considered "noise" that must be filtered out when detecting radio waves from satellites orbiting the Earth. The point here is that the term "noise" is one used for convenience, rather than as an absolute assessment of the usefulness of part of a time series. A more accurate description would be "time-varying fluctuations that are known to be irrelevant for the current application, or about which too little is known to determine whether it contains meaningful information." As long as this is understood, "noise" is a suitable short-hand term.

12.2 Losing signal, but losing more noise

Many time-frequency analyses and other signal processing strategies involve some degradation or attenuation of the signal. That is, in many cases, the original time series cannot be perfectly reconstructed based on the results of the time-frequency analysis.

In part, this is because time-frequency analyses often use only power and ignore the phase, and in part this is because time-frequency analyses often involve subsampling frequencies and time points (that is, the frequency and temporal resolutions of the time-frequency results are often lower than those of the original time series).

On the other hand, time-frequency analyses allow a separation of signal and noise, or a separation of multiple temporally overlapping but spectrally distinct components of a signal. This is a major advantage that allows a significant increase in the amount of information that can be isolated and interpreted from a time-domain signal. This increase in information must be balanced against the potential loss of signal. With appropriate time-frequency analyses, the minor loss of signal is often inconsequential compared to the strong attenuation of noise and other unwanted parts of the time series.

12.3 Analyses are not right or wrong, they are appropriate or inappropriate for a certain context

If you would ask “Is analysis method A or analysis method B better?” the answer would always be “it depends…” Similarly, if you would ask “Is parameter X better than parameter Y for analysis A” the answer would again be “it depends…”

Some analysis methods are better suited for narrow-frequency signals, other methods are better suited for broadband signals. Some parameters are better suited for rapidly changing dynamics whereas other parameters are better suited for slowly changing dynamics. Some analyses are more robust to noise whereas other analyses are more sensitive to noise. Although it is true that for some situations and some types of signals, some types of analyses with some ranges of parameters are better than other analyses and parameters, it is important to realize that there is never a single method or parameter set that “wins” all the time, for all types of time series and signal characteristics.

How do you know which analysis and with which parameters to use? In general, a good strategy is to use analysis methods and parameters that have been used before for the kind of data you are dealing with. This is a good way to start, but keep in mind that just because someone else has used a particular analysis method, does not mean that it is the optimal method for that type of data.

Fortunately, as you build experience with time-frequency analyses and signal processing techniques, you will develop an intuition of which methods are likely to be most useful in which situations.

Most importantly, try different analysis methods and different sets of parameters, and determine for yourself the best approach for your data. You can simulate time series that contain characteristics of your data in order to develop an analysis protocol best suited for your applications.

12.4 Good luck and have fun

Above all else, embrace and enjoy the experience of learning time-frequency analyses. Extracting information from time series data, learning about different analysis methods,

programming, thinking critically, plotting data, and inspecting results will help you develop important critical thinking skills that you can use in work, education, science, and day-to-day decision-making. Good luck and have fun!

Chapter 13: Alphabetical list and explanation of all Matlab functions used in this book

This chapter lists and defines all of the Matlab functions used in this book. Usage hints and common errors are also provided. This chapter does not contain an exhaustive description of all possible inputs and outputs of the listed functions; rather, the functions are explained in the manner necessary to complete this book successfully.

This chapter is the full expansion of the custom dictionary that comes with the book and that can be downloaded from sincxpress.com. When installed on your e-reader device, the dictionary will provide information about each Matlab function as you work through this book.

Additional information on Matlab functions can be obtained by typing **`help <function>`** or in some cases **`doc <function>`**, by searching the internet, or by asking a friend or colleague.

In general, Matlab functions take the form **`output = function(input);`** or **`[output1,output2] = function(input1,input2);`** If only one output is given, it is omitted in the list below.

`abs(data)`

If **`data`** comprises real-valued numbers, **`abs`** returns their absolute values. If **`data`** comprises complex numbers, **`abs`** returns their magnitude (the length of a line in complex space starting from the [0,0] origin and ending at a point defined by the real and imaginary coordinates). In time-frequency analyses, **`abs`** is most often used to obtain amplitude information from Fourier coefficients or the result of complex Morlet wavelet convolution.

`angle(data)`

Returns the angle of a complex number, which is to say, the angle in radians formed between the origin of a complex plane and a vector drawn from the origin to a point described by a complex number, relative to the positive real axis.

`bar(x,y)`

Produces a bar plot of **`y`** using x-axis coordinates in **`x`**. If only **`y`** is inputted, **`x`** is assumed to be integer values from 1 to **`length(y)`**.

`bsxfun(@function,a,b)`

Applies **`@function`** to vectors or matrices **`a`** and **`b`**. **`a`** and **`b`** can be different sizes, as long as the size of one dimension matches. Often, **`bsxfun`** can be used instead of **`repmat`**. **`@function`** can be one of several built-in functions such as **`@times`**, **`@rdivide`**, **`@add`**, or can be a user-defined function.

`cat(dim,a,b)`

Concatenate matrices **`a`** and **`b`** along dimension **`dim`**. In some cases, it may be easier to

concatenate via square brackets, as in **[a b]** or **[a; b]**.

ceil(data)

Round up to the nearest integer. For example, **ceil(1.0001)** is **2**.

colorbar

Adds a colorbar to the current axis. This facilitates understanding the mapping between color in an image, and numerical values in the data. If a colorbar is already present, typing **colorbar** again will remove the colorbar.

contourf(x,y,data,N,'linecolor','none')

Produces a smooth image of 2D matrix **data** with x-axis and y-axis coordinates **x** and **y**. **N** refers to the number of contour lines; more lines produce a smoother-looking plot, but may take longer to render. For many images, between 20 and 50 lines is sufficient. **'linecolor','none'** produces an image without explicit contour lines, and is more easily interpretable for N>10. Without this option, **contourf** will draw black lines for each contour.

cos(data)

Returns the cosine transform of **data**. See **sin**.

conv(data,kernel)

Returns the result of convolution between **data** and **kernel**. The result will be of length D+K-1, where D is the length of **data** and K is the length of **kernel**. In Matlab (not Octave), an optional third input can be provided (**'same'**) that automatically trims the result of convolution such that it is the same length as **data**.

cumsum(data)

Returns the cumulative sum, meaning that at each point in **data**, the return value is the sum from the first point in **data** to the current point. For example, **cumsum([1 3 1 5])** is **[1 4 5 10]**.

detrend(data)

Removes a linear trend from **data**. This is also used to remove DC offsets. This can be useful during processing/cleaning, because some forms of noise produce slow linear drifts. Detrending can also remove mean non-stationarities if those non-stationarities are linear or approximately linear.

diff(data,N,dim)

Computes the **N**-order derivative (difference over successive elements) along dimension **dim**. Note that the result will contain one element fewer compared to **data**. It is thus often useful to add an extra data point to the derivative; this may simplify subsequent programming and analyses. For example: **xd=diff(x); xd(end+1)=xd(end);**

disp(statement)

Prints **statement** to the Matlab command prompt. It is often used to update progress during long analyses. For example, at the end of a loop over channels (looping index **chani**) there could be **disp(['Finished analyzing channel' num2str(chani) '.'])**. This would print in the Matlab command prompt as "Finished analyzing channel 34."

double(data)

Converts the single-precision or Boolean (true/false) **data** to floating-point precision. Storing data in single precision is often useful to reduce variable and file sizes. Converting to floating-point precision is necessary for some plotting and filtering functions. Note that converting numerical data to single precision can result in information loss, depending on the scale of the data.

dpss(pnt,N)

Returns 2**N** **pnt**-length discrete prolate Slepian sequences. These are used as tapers during a multitaper analysis. This function is part of the Signal Processing toolbox of Matlab. As of summer 2014, it is not available in Octave.

dsearchn(data,values)

Finds the indices in **data** closest to **values**. **dsearchn** expects column-format **data**. Thus, if **dsearchn** produces an error, try first to transpose either or both of **data** and **values**.

exp(k)

Natural exponent to the power of **k** (e^k).

fft(data,N,dim)

Returns the fast Fourier transform of **data**, using **N** points. **data** will be zero-padded if **N** is larger than **size(data,dim)**, and **data** will be truncated if shorter. The default **N** is **size(data,dim)**. If **data** is a matrix, the FFT will be computed over dimension **dim**.

filtfilt(kernel,1,data)

Apply a zero-phase-shift time-domain filter specified by **kernel** on **data**. The second input is a weighting vector, and is often set to 1 (thus, no weighting). **kernel** can be created by Matlab functions such as **firls** or **fir1**, which are in the Signal Processing toolbox in Matlab or the Signal package in Octave.

firls(order,frex,shape)

Computes and returns a time-domain filter kernel of length **order** for frequencies specified by **frex** and corresponding to the frequency-domain shape **shape**. The order must be larger than the lowest frequency, and is typically set to between 3 and 10 times as long as the lowest frequency (thus, if the lower frequency filter bound is 10 Hz, **order** should be at least 300 ms). Narrow-frequency-band activity may be better resolved with relatively higher orders. **frex** is a vector of frequencies between 0 (DC) and 1, which correspond to fractions of the Nyquist frequency. For example, if the sampling rate is 500 Hz, 25 Hz corresponds to 0.1. **shape** is the desired ("ideal") frequency response shape. In most cases, this is specified as a plateau with softened edges. The **shape** and **frex** vectors must have the same number of elements. A simple plateau-shaped filter should have six points: 0, start of lower transition zone, lower frequency bound, upper frequency bound, end of upper transition zone, and Nyquist. This function is in the Signal Processing toolbox in Matlab or the Signal package in Octave.

floor(data)

Rounds down **data** to the nearest integer. For example, **floor(1.9999)** is 1, and

floor(-1.9999) is -2. To round to the integer closest to zero (thus, **floor** if positive and **ceil** if negative), use **fix(data)**.

for i=1:5, …, end

Loop (in this example) 5 times and run the commands listed before **end**. The variable **i** is automatically incremented immediately before **end**. Loops can be escaped with **break**, and individual iterations can be skipped with **continue**. For example, to skip iteration 2:

```
for i=1:5
  if i==2, continue; end
  disp(num2str(i));
end
```

get(gca,'property')

Returns the value of the requested property of the current (most recently used, or mouse-clicked) axis. **gca** can be replaced with an axis or figure handle if one is created. Common uses of **get** are to find the x-axis or y-axis limits, for example **get(gca,'xlim')**. For a list of properties, type **get(gca)** or **get(gcf)**.

hilbert(data)

Returns the Hilbert transform of **data**. This function is contained in the Matlab Signal Processing toolbox, or the Octave Signal package. However, it can be created using the explanation in Chapter 7.1. This function accepts matrices, but will always compute the Hilbert transform along columns. If **data** are organized as channels X time, it will be necessary to transpose **data** first (and it might be useful to transpose the results after the Hilbert transform). When in doubt, input only a single vector into the function to match the output against the output of matrix input.

[y,x]=hist(d,n)

Generates a histogram of data **d** using **n** bins. If no outputs are requested, a plot is generated. The histogram can be recreated with the outputs as **plot(x,y)**.

hold on/hold off/hold all

Freezes the active plot so that new **plot** commands draw on top of, rather than replacing, the current graph. The active plot can be called from the command (see **subplot** and **figure**) or clicked on with a mouse. **hold all** additionally preserves the order of setting line colors.

if … elseif … else … end

A conditional control statement that runs a section of code only if what appears after **if** is true. **if** and **end** are required; **elseif** and **else** are optional. Below is an example.

```
i=4;
if i==2
  disp('2')
elseif i==3
```

```
  disp('3')
else
  disp('not 2 or 3')
end
```

ifft(data,N,dim)
Returns the fast inverse Fourier transform. See **fft**.

imag(data)
Returns the imaginary part of a complex number (or vector or matrix). If the input is real-valued only, **imag** returns 0.

interp1(x,y,t,'spline')
Performs linear interpolation of values in **y** at time points in **x**, over the points **t**. **t** must be in the same range as **x**. For example, the following code will perform a spline interpolation of three points over 20 points. **y = interp1(1:3,[5 12 -4],linspace(1,3,20),'spline');**

legend({'a';'b';'c'})
Places a legend in the current plot. The order of texts in the cell array must match the order in which the data were plotted.

length(data)
Returns the number of elements in the longest dimension of matrix **data**. If there is uncertainty as to which dimension **length** is returning, it is better to use the function **size**.

linspace(low,high,N)
Returns **N** linearly spaced numbers between **low** and **high**.

log(data)
Returns the natural logarithm of **data**. Remember that the logarithm of zero is not defined, and the logarithm of a negative number is complex. If you might have zeros in the data, use **log(data+eps)** to add a tiny number to data. If you have negative values, you can use **log(data-min(data)+eps)**.

logspace(low,high,N)
Returns **N** logarithmically spaced numbers between **low** and **high**. Note that **low** and **high** are interpreted as power-of-10. Thus, to obtain six logarithmically spaced numbers between 4 and 20, type **logspace(log10(4),log10(20),6)**.

[val,idx]=max(data)
Returns both the value and the index (location in the matrix) of the largest number in **data**.

mean(data,dim)
Computes the arithmetic mean of matrix **data** along dimension **dim**.

[val,idx]=min(data)
Returns both the value and the index of the smallest number (that is, the left-most on the number line; thus, -10 is smaller than 0) in **data**. **min** can be used in combination with

abs to find the point in a vector that is closest to a desired value. For example, **[val,idx]=min(abs([1 2 5 6]-4.6));** will return **idx** of 3, indicating that the 3rd element in the [] array is closest to 4.6. See also **dsearchn**.

mod(data,n)

Returns the modulus of **data** by **n**, in other words, the integer remainder when dividing **data** by **n**.

nextpow2(N)

Returns the number m such that 2^m is the next power-of-2 greater than **N**. For example, **nextpow2(3)** is 2, meaning that 2^2 is the first power-of-2 number that is greater than 3. It is often used when determining how much to zero-pad a time series to maximize the efficiency of a fast Fourier transform.

num2str(data)

Converts the number in **data** to a text string. This is often used when generating plot labels.

numel(data)

Returns the total number of elements in **data**, regardless of the number of dimensions. For example, if **data** were a 2x5x8x6 matrix, **numel(data)** would be 480.

plot(x,d,'r-','linewidth',3)

Plots data **d** as a function of x-axis values **x**. This function can take one- or two-dimensional data. **'r-'** refers to a red dashed line. **'linewidth',3** refers to the thickness of the lines. Type **help plot** for additional plotting options.

plot3(x,y,z)

Similar to **plot**, except for 3D data. Vectors **x**, **y**, and **z** must have equal length. It is often useful to visualize 3D plots by moving them around in space with the mouse. In Octave, this function is on by default; in Matlab, it can be activated with **rotate3d** (or by clicking on the rotate button on the top of the figure).

p=polyfit(x,y,o)

Computes a polynomial fit of variable **x** to variable **y**, using an order of **o**. Output **p** contains a vector of **o**+1 coefficients, where the final coefficient is the intercept (the value on the y-axis when the predicted function is at 0 on the x-axis). The resulting prediction of **y** based on **x** and **p** can be evaluated using **yy=polyval(p,x);**.

prod(data)

Returns the product (multiplication) of all elements in **data**. For example, if **data** were a matrix **[2 4 1 7]**, the result of **prod** would be 56.

rand(dims)

Returns randomly generated numbers with a uniform distribution between 0 and 1. The inputs to **rand** are the size of the matrix (**dims**). Note that giving one input *N* defaults to an *N*x*N* matrix; a *1xN* vector is created with **rand(1,N)**. To create uniformly distributed random numbers between arbitrary numbers **a** and **b**, type **a+(b-a)*rand(dims)**.

randn(size)

Same as **rand** but produces normally distributed numbers (mean of zero, variance of one). To create random numbers with mean **a** and standard deviation **b**, type **a+b*randn(dims)**.

real(complexnumber)

Extracts the real part of a complex number.

repmat(data,a,b)

Repeats matrix **data a** by **b** times. For example, if **data** were [1 2], **repmat(data,1,2)** would be [1 2 1 2] and **repmat(data,2,1)** would be [1 2; 1 2]. Higher dimensional replications are also possible. This function is most often used when multiplying two matrices of different sizes, for example, when multiplying a 1x4 matrix by each row of a 200x4 matrix. See also bsxfun.

reshape(data,new_dims)

Changes the shape of **data** to be **new_dims**. For example, if **data** is a 1x10 vector and **new_dims** is [2 5], the output of this function will be a 2x5 matrix (be careful here: this is different from reshaping to [5 2]). The number of elements in **data** must be the same as the size of **new_dims**. That is, **numel(data)** must equal **prod(new_dims)**.

rotate3d

Allows click-and-drag rotation of plots or images. This function is necessary only in Matlab, not in Octave.

round(data)

Rounds to the nearest integer. To round to the nearest decimal, use **round(data*N)/N**, where **N** is the n^{th} decimal to round to in power-of-ten (10, 100, 1000, etc.).

set(gca,'property',value)

Changes aspects of a axis. **gca** indicates the current axis (the one most recently used, or most recently mouse-clicked), or can be replaced with a handle (e.g., **h=plot(rand(3)); set(h,…)**). **'property',value** refer to changeable properties and their new values, for example, **set(gca,'xlim',[0 100])** changes the x-axis limit. See also **get()**.

sin(x)

Returns the sine function of **x**. It is typically used to generate a sine wave as **a*sin(2*pi*f*t+p)**, where **a** is the amplitude, **f** is the frequency, **t** is time (in seconds if **f** is specified in Hz), and **p** is the phase.

size(data[,dim])

Returns the sizes of each dimension (that is, the number of elements) of matrix **data**. If you want to know the total number of elements in a matrix, use either **prod(size(data))** or **numel(data)**. If the optional second input **dim** is provided, only the number of elements along dimension **dim** will be returned. For example, if **data** is a 3x4x2x10 matrix, **size(data,3)** will return **2**.

[vals,idx]=sort(data)

Sorts **data** according to ascending values. **idx** contains the new indices. Thus, **data(idx)** is the same thing as **vals**.

squeeze(data)

Removes singeton dimensions of **data**. For example, if **data** were a 4x1x8x1x3 matrix, **squeeze(data)** would be a 4x8x3 matrix. This is often necessary when plotting or imaging parts of high-dimensional matrices.

std(data,[],dim)

Returns the standard deviation of **data** along dimension **dim**. The second input can be set to 1 for normalization by N instead of N-1. In most cases, samples (rather than the entire population of data) are provided, and thus N-1 is the appropriate normalization.

subplot(c,r,i)

Divides the figure into **c** columns and **r** rows, and sets subplot **i** to be the current.

sum(data,dim)

Computes the sum (addition of all elements) over dimension **dim**.

switch/case/otherwise/end

A **switch** statement is a conditional, similar to the if-then statement. The value of the variable after **switch** is compared for each **case**. If none of the cases match, the optional **otherwise** will be evaluated. See below for an example.

```
val = 2;
switch val
  case 1
    disp('val is one');
  case 2
    disp('val is two');
  case 3
    disp('val is three');
  otherwise
    disp('val is out of range.');
end
```

title('text')

Gives a title to the current plot.

unwrap(phases)

"Unwraps" phase angles in **phases** (in radians). This means that 2pi is added to the phase angles whenever they would otherwise return to -2pi. This is used when computing the derivative of phase, which is part of computing instantaneous frequencies.

view([az el])

Sets the orientation of a 3D plot (generated by, e.g., **plot3** or **surf**) according to the azimuth (**az**) and elevation (**el**). Values range from -90 to +90. By clicking-and-dragging the plot around, the current azimuth and elevation are printed in the lower left of the Matlab figure.

`while … end`

Continuously runs a set of code as long as the Boolean expression after **`while`** is true. For example, the code below will continue generating random numbers until a number is bigger than 1.8.

```
aRandomNumber = randn;
while aRandomNumber<1.8
  aRandomNumber = randn
end
```

`xlabel('text')`

Gives a label to the x-axis of the current plot.

`ylabel('text')`

Gives a label to the y-axis of the current plot.

`zeros(dims)`

Create a matrix entirely of zeros, of specified size **`dims`**. This is often used to initialize variables. Note that **`zeros(N)`** produces an *N*x*N* matrix of zeros. To produce a vector, use **`zeros(1,N)`** or **`zeros(N,1)`**.

`zlabel('text')`

Gives a label to the z-axis of the current plot.