CHAPTER

5

Analog Signals

In this chapter, we will cover basic signal processing techniques by analyzing neural data in the form of analog signals. Analog signals are commonly used in neuroscience—the electroencephalogram (EEG) records large electric potentials from the scalp, the magnetoencephalogram (MEG) measures magnetic fields from sources within the brain, the local field potential (LFP) is obtained through recording the activity of many neurons; all are analog neural signals.

Looking at spike trains (which we have done in the last two chapters) amounts to looking at a signal that is fundamentally binary and discrete—a neuron either fires, or it does not fire. In other words, we have been looking at inherently *digital* signals. But the underlying voltage trace from which we extracted the spikes was an analog signal, it was a time course of voltage values. Extracting spikes from this analog signal corresponds to feature extraction—spikes are such salient features (high frequency and high amplitude)—that one would be hard pressed not to note the presence of spikes in a voltage trace. In contrast, when working with analog signals, one is typically working with the entire signal, not with detected features. When working with the entire signal, it is helpful to have some basic knowledge about signal processing, which we hope to develop in this chapter.

A starting point in signal processing is the notion that some signals are best represented in the time domain but aspects of others are better represented in the *frequency domain*. The frequency domain (or *frequency space*) is often rather counterintuitive to the beginner, as we all live our lives in the time domain and our data are usually returned by our data recording apparatus in the form of a time series (in the time domain) as well. So what is frequency space and why is a frequency space representation of data useful?

Simply put, a frequency space representation emphasizes periodic or repeating components of a signal. This might sound somewhat abstract, so it is perhaps best to jump right in, with a coded example. We start by plotting *sine waves*.

A sine wave is a cyclically repeating signal that can be completely described by just three parameters: (1) its frequency (how many times the cycle repeats per second), (2) its amplitude—we didn't specify this here, by default the sine wave

Neural Data Science.
DOI: http://dx.doi.org/10.1016/B978-0-12-804043-0.00005-2

goes from –1 to 1 if we multiplied the signal time course by a scaling factor, we could make it go from –0.1 to 0.1 or from –100 to 100 or any other range we would like, (3) its phase, which is the point in the cycle at which we start drawing the sine wave. We didn't specify that here either and by default, the sine wave starts at *y*-position 0 at time 0. The cosine function is simply a sine wave that is phase-shifted by 90 degrees—it starts to draw at *y*-position 1 at time 0 (see Fig. 5.1).

```
Pseudocode
1. Open figure
2. Define the sampling frequency, here 1000 Hz
3. Duration of our time series, in seconds. Here: 1 second
4. Define a time base t, in steps of sampling frequency
5. Define signal frequency, here 2 Hz
6. Represent a sine wave
7. Represent a cosine wave
8. Open a subplot with two horizontal panels
9. Plot the sine wave over the time base as a black, thick line
10. Open the a subplot in the lower panel
11. Plot the cosine wave over the time base as a black, thick line
```

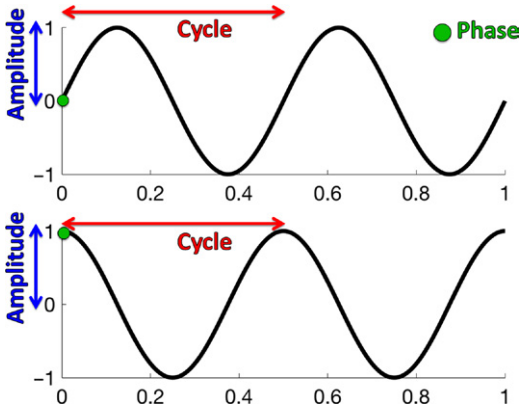

FIGURE 5.1 Sine (top panel) and cosine (bottom panel) function. We added arrows to point out concepts we referenced in the text above, such as amplitude, cycle, frequency, and phase (these are not created in code). The *x*-axis represents time, so both functions have a frequency of 2 cycles per second or 2 Hz.

```python
import matplotlib.pyplot as plt
import numpy as np
fig=plt.figure() #1
fs = 1000; #2
dur = 1; #3
t = np.linspace(0,dur,fs*dur); #4
freq = 2; #5
sinW = np.sin(2*np.pi*freq*t); #6
cosW = np.cos(2*np.pi*freq*t); #7
ax=plt.subplot(2,1,1) #8
ax.plot(t,sinW,c='k',lw=4) #9
ax=plt.subplot(2,1,2) #10
ax.plot(t,cosW,c='k',lw=4) #11
```

```matlab
figure %1
fs = 1000; %2
dur = 1; %3
t = 0:1/fs:dur; %4
freq = 2; %5
sinW = sin(2*pi*freq*t); %6
cosW = cos(2*pi*freq*t); %7
subplot(2,1,1) %8
plot(t,sinW,'color','k','linewidth',4) %9
subplot(2,1,2) %10
plot(t,cosW,'color','k','linewidth',4) %11
```

Now that we know what sine waves are, we can build up a signal that is a bit more complex.

```
Pseudocode
1. Open a new figure
2. Duration of our time series, in seconds. Here: 4 seconds
3. Define a time base t, in steps of sampling frequency
4. Define signal frequency, here 10 Hz
5. Create the time course of the signal, here a sine wave
6. Open a subplot with three horizontal panels
7. Plot the signal over the time base as a black, thick(ish) line
```

```python
fig = plt.figure() #1
dur = 4 #2
t = np.linspace(0,dur,fs*dur) #3
freq = 10 #4
signal = np.sin(2*np.pi*freq*t) #5
ax = plt.subplot(3,1,1) #6
ax.plot(t,signal,c='k',lw=2) #7
```

```matlab
figure %1
dur = 4; %2
t = 0:1/fs:dur; %3
freq = 10; %4
signal = sin(2*pi*freq*t); %5
subplot(3,1,1) %6
plot(t,signal,'color','k','linewidth',2) %7
```

If you execute this code, you can count the number of cycles—either by counting the number of peaks or valleys, see that the number is 40, divide it by the duration of the signal (4 seconds) and conclude that the signal was a sine wave with a frequency of 10 Hz. Which is true. So far, so good, see top panel of Fig. 5.2.

But signals with a single cyclical component are rare, particularly in neuroscience. Let's see what happens if we add a second sine wave to the first one. We haven't done anything fancy, we didn't change phases or amplitudes (which will also most definitely be variable in real neuroscience applications), but can you still discern the two component signals, even in this simple case?

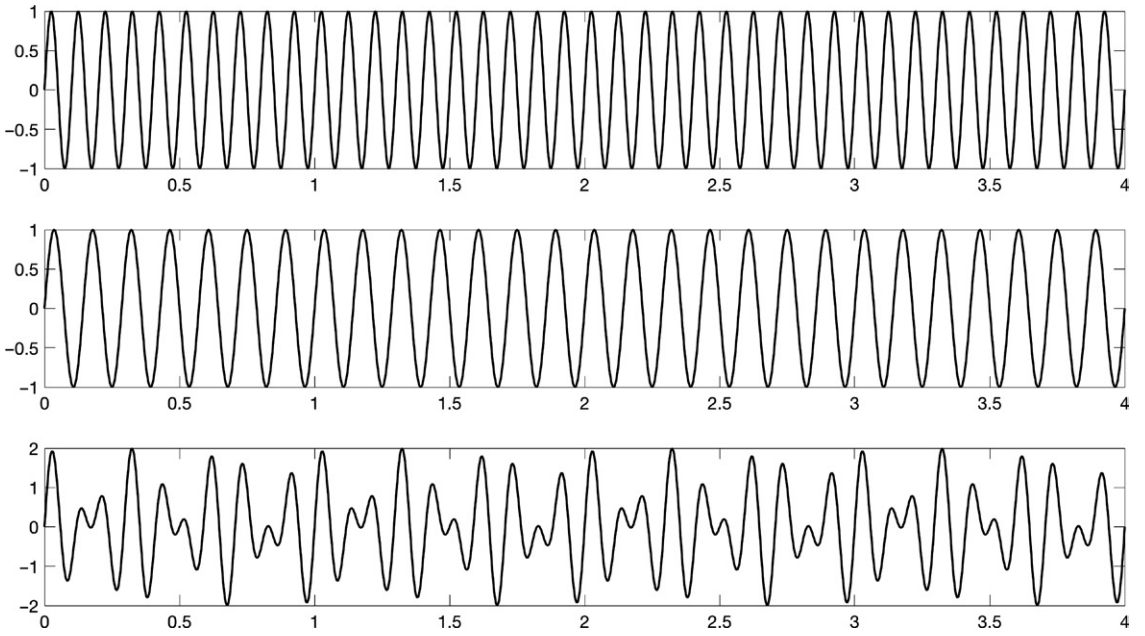

FIGURE 5.2 Adding two sine waves. Top panel: Sine wave with frequency 10 Hz. Middle panel: Sine wave with frequency 7 Hz. Bottom panel: The two sine waves, added; *x*-axes: time in seconds; *y*-axes: amplitude.

Pseudocode

1. Define signal frequency of second sine wave, here 7 Hz
2. Create the time course of the 2nd signal, again a sine wave
3. Open the 2nd horizontal panel in the subplot
4. Plot the 2nd signal over the time base as a black, thick(ish) line
5. Add the two signals
6. Open the bottom horizontal panel in the subplot
7. Plot the joint signal over the time base as a black, thick(ish) line

Python

```
freq2 = 7; #1
signal2 = np.sin(2*np.pi*freq2*t); #2
ax=plt.subplot(3,1,2) #3
ax.plot(t,signal2,c='k',lw=2) #4
jointSignal = signal+signal2; #5
ax=plt.subplot(3,1,3) #6
ax.plot(t,jointSignal,c='k',lw=2) #7
```

MATLAB

```
freq2 = 7; %1
signal2 = sin(2*pi*freq2*t); %2
subplot(3,1,2) %3
plot(t,signal2,'color','k','linewidth',2) %4
jointSignal = signal+signal2; %5
subplot(3,1,3) %6
plot(t,jointSignal,'color','k','linewidth',2) %7
```

Unless you are an expert in signal processing, we wager that you may not be able to. Call it Pascal's wager (reloaded). And this is a signal with just two pure sine waves, with the same amplitude and no relative phase shifts.

In addition, pure sine waves are a platonic concept: they only live in the realm of ideas. The real world, in which we all dwell, is messy and dirty. The real world is corrupted by *noise*. Always. Whereas a black hole might or might not have hairs (Gürlebeck, 2015), anything in neuroscience most definitely has hairs and lots of it, so we better get used to handling this abundant hairiness. This starts now, right away. Let's add just a sprinkle of noise to the signal:

Pseudocode

1. Determine the number of points in the time base
2. Create a vector of random numbers drawn from a normal distribution, with the same length as the signal (and time base)
3. Add the noise to the signal
4. Open a new figure
5. Open a new subplot
6. Plot the joint signal from 5.2 again
7. Switch to middle panel of subplot
8. Plot the signal with the noise over the entire time base

(*Continued*)

Python	MATLAB
`n = len(t) #1`	`n = length(t); %1`
`noise = np.random.randn(n) #2`	`noise = randn(1,n); %2`
`signalAndNoise = jointSignal + noise #3`	`signalAndNoise = jointSignal + noise; %3`
`fig = plt.figure() #4`	`figure %4`
`ax = plt.subplot(3,1,1) #4`	`subplot(3,1,1) %5`
`ax.plot(t,jointSignal,c='k',lw=2) #6`	`plot(t,jointSignal,'color','k','linewidth',2) %6`
`ax = plt.subplot(3,1,2) #7`	`subplot(3,1,2) %7`
`ax.plot(t,signalAndNoise,c='k',lw=2) #8`	`plot(t,signalAndNoise,'color','k','linewidth',2) %8`

This is starting to look a lot like a sleep EEG. At this point, even seasoned veterans of signal processing will be hard pressed to resolve the component signals in this time course. That's ok—it is extremely hard to recover the cyclical components of even simple signals in a moderate amount of noise by looking in the time domain. In perception, perspective matters. That's where a frequency domain representation comes in. It highlights cyclical components in a signal, making it much easier to ascertain which of them are present—or absent.

To see the power of looking at the frequency domain (literally), we'll look at this signal in frequency space right now. You may not fully understand the code that we are writing at this time, but that is ok—we will use this gap to motivate how to close it in the remaining text of this chapter. Perhaps this code will elicit a burning desire to understand it in you, the reader. If so, we have succeeded. It should be clear what exactly is going on by the end of this chapter.

Pseudocode

1. Determine the Nyquist frequency
2. Do the fast Fourier transform of the combined signal, normalized by time
3. Create a frequency base for plotting, in analogy to the time base that goes from zero to the Nyquist frequency and is a long as there are valid points in the fft
4. Switch subplot to the last panel, call it powerPlot
5. Determine what half the signal in frequency space is
6. Take the complex conjugate of that
7. Calculate power
8. Plot power over the frequency base
9. Pick appropriate x and y limits of the plot—we could in principle look at everything from 0 to 500, but the two values we care about would be bunched too close together
10. Add tickmarks with a higher resolution
11. Add axes labels

(*Continued*)

```
Python          nyquist = fs/2 #1
                fSpaceSignal = np.fft.fft(signalAndNoise)/len(t) #2
                fBase = np.linspace(0,nyquist,np.floor(len(signalAndNoise)/2)+1) #3
                powerPlot = plt.subplot(3,1,3) #4
                halfTheSignal = fSpaceSignal[:len(fBase)] #5
                complexConjugate = np.conj(halfTheSignal)#6
                powe = halfTheSignal*complexConjugate#7
                powerPlot.plot(fBase,powe, c='k',lw=2) #8
                powerPlot.set_xlim([0, 20]); #9
                powerPlot.set_xticks(range(20));#10
                powerPlot.set_xlabel('Frequency (in Hz)') #11
                powerPlot.set_ylabel('Power')#

MATLAB          nyquist = fs/2; %1
                fSpaceSignal = fft(signalAndNoise)/(length(t)/2); %2
                fBase = linspace(0,nyquist,floor(length(signalAndNoise)/2)+1); %3
                powerPlot = subplot(3,1,3) %4
                halfTheSignal = fSpaceSignal(1:length(fBase)); %5
                complexConjugate = conj(halfTheSignal); %6
                pow = halfTheSignal.*complexConjugate; %7
                h = plot(fBase,pow, 'color','k','linewidth',2) %8
                xlim([0 20]); ylim([0 1]); %9
                powerPlot.XTick = 0:20; %10
                xlabel('Frequency in Hz'); ylabel('Power'); %11
```

The figure produced by all this code should look something like Fig. 5.3—although not precisely because we add some random noise that impacts how the last two panels look.

Before we move on, we want to note several things about the code and the plot.

The heavy lifting in this function is done by the Fourier transform, specifically a fast version of it, developed by Tukey (Cooley & Tukey, 1965). It is the Fourier transform that transports us (or rather, the signal) into frequency space. Frequency space is a place of magic and wonder—the Fourier transform of a time series usually involves complex numbers with imaginary parts. That is why we plot the Fourier transform itself by typing `plot(fSpaceSignal)`, it will yield a funky plot (try it). That's because the complex numbers represent both *magnitude* and *phase* at once, but we are only interested in amplitude or power at this point, which is why we plotted only the absolute values above. Also, if you type `plot(abs(fSpaceSignal))`, you will get a plot of all the magnitudes, but note that it is mirror-symmetric in the middle—it repeats after half the sampling rate, which is why we plotted things only up to half the sampling rate above (Fig. 5.4).

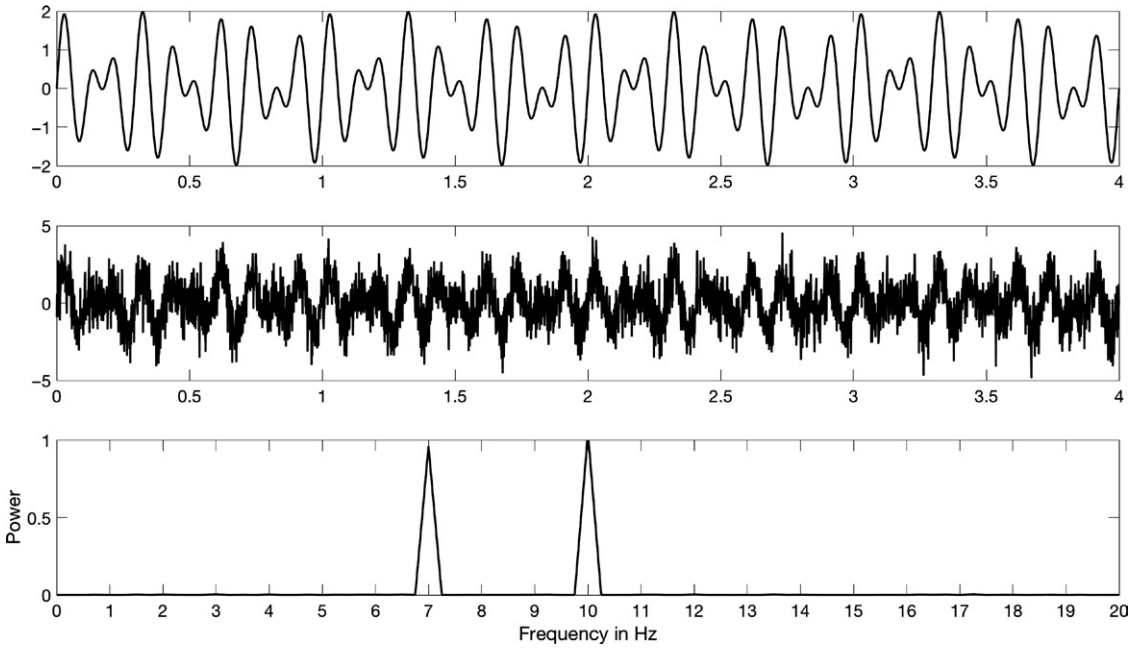

FIGURE 5.3 Adding noise to the sine waves makes it hard to discern what is going on with the signal in the time domain but not in the frequency domain. Axis labels are time vs. amplitude in the first two plots and frequency vs. power on the last one. Note that the power is not exactly 1 for each component, as noise doesn't just add power to where there is none (frequencies other than 7 and 10), it also removes power from where there is signal (7 and 10).

As you can see, power matters. For cyclical phenomena, always have the will to look at power.

If you go beyond half the sampling rate (also called the *Nyquist frequency* in honor of Harry Nyquist), you will encounter a phenomenon called *aliasing*. Put simply, the shape of signals over time can only be represented properly if one is sampling the signal more than twice as fast as the fastest frequency component in the signal itself. Otherwise, the shape of the sampled signal will be distorted. This might appear somewhat abstract, but because it is such a fundamental principle of signal processing, it is worth trying to really understand it—much of what follows will be dependent on it, and we'll try to do so with an example. Again, let's first write the code, then look at the figure, then try to understand what is going on.

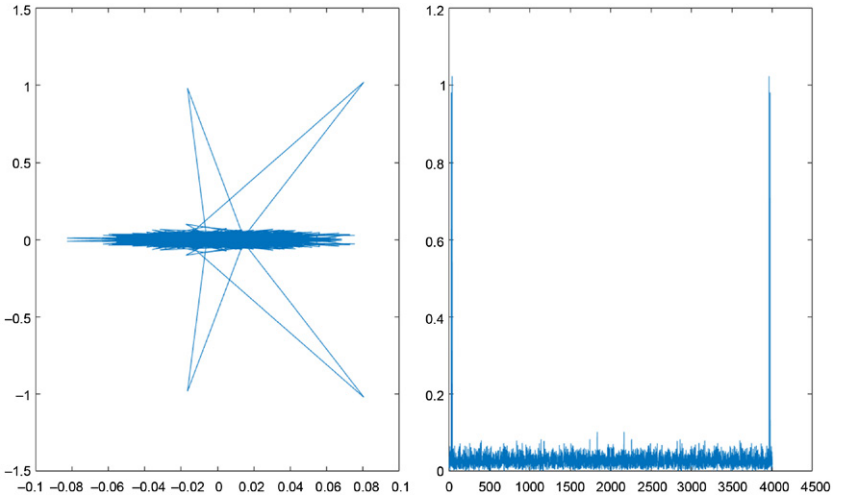

FIGURE 5.4 If you see this, you are doing it wrong. Left panel: `plot(fSpaceSignal)`. Right panel: `plot(abs(fSpaceSignal))`.

`Pseudocode`

1. Define the frequency of reality, here 1000 Hz
2. Define a time base with the same resolution
3. Define the signal frequency, here 10 Hz
4. Pick 9 strategically chosen sampling frequencies to show something interesting
5. Define the signal as a sine wave at the signal frequency over time
6. Open a figure (for Fig. 5.5)
7. Open a subplot
8. Plot the signal over time
9. Turn the hold on so that we can plot in the same subplot again without overwriting the old one
10. Calculate the sampled signal for a given sampling rate
11. Plot the sampled signal over the same time base in red
12. Add black circles as markers, with yellow edges
13. Make them a bit bigger so one can see them
14. Add a title, namely the corresponding sampling frequency, assign to title handle hT
15. Make title font larger, set it to 26
16. Loop over all sampling frequencies

(*Continued*)

```
Python          realSampling = 1000 #1
                t = np.linspace(0,1,realSampling) #2
                signalFrequency = 10 #3
                samplingRate = [3, 4, 8, 10, 11, 12, 20, 41, 200] #4
                signal = np.sin(2*np.pi*signalFrequency*t)#5
                fig=plt.figure() #6
                for ii,sampleRate in enumerate(samplingRate): #15
                    ax = plt.subplot(3,3,ii+1)#7
                    ax.plot(t,signal)#8
                    sampledSignal = np.rint(np.linspace(0, len(t)-1, sampleRate)).astype(int)#10
                    q = ax.plot(t[sampledSignal],signal[sampledSignal],c='r',marker='o',mfc='k',mec='y',
                markersize=6); #11
                    plt.title('fs = '+str(sampleRate)) #14
MATLAB          realSampling = 1000; %1
                t = 0:1/realSampling:1; %2
                signalFrequency = 10; %3
                samplingRate = [3 4 8 10 11 12 20 41 200]; %4
                signal = sin(2*pi*signalFrequency*t); %5
                figure %6
                for ii = 1:length(samplingRate) %16
                subplot(3,3,ii)%7
                plot(t,signal)%8
                hold on %9
                sampledSignal = round(linspace(1, length(t), samplingRate(ii))); %10
                q = plot(t(sampledSignal),signal(sampledSignal),'color',' r'); %11
                q.Marker = 'o'; q.MarkerFaceColor = 'k', q.MarkerEdgeColor = 'y'; %12 q.MarkerSize = 6; %13
                hT = title(['fs = ',num2str(samplingRate(ii))]); %14
                hT.FontSize = 26; %15
                end %16
```

This code yields Fig. 5.5.

The top row in Fig. 5.5. represents cases of serious undersampling. If we sample the signal at 3 Hz, there seems to be no signal, whereas sampling at 4 and 8 Hz seems to yield one and two cycles, respectively. When sampling around the signal frequency, interesting things happen, as seen in the middle row, either missing the signal completely or yielding a single sinusoidal cycle of differing frequencies. The bottom row shows sampling at or beyond the Nyquist frequency. Once one is sampling at the Nyquist frequency (here, 20 Hz), the number of cycles in the original signal will be

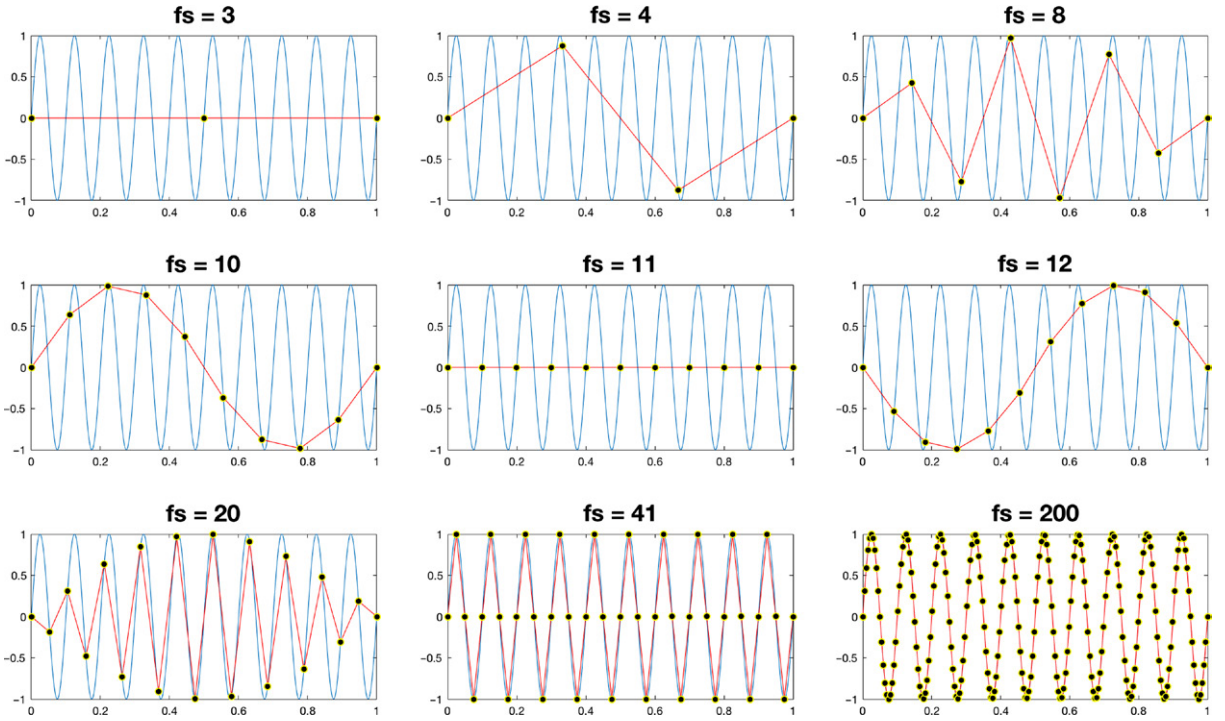

FIGURE 5.5 Aliasing. The blue trace in each subplot is the same and represents the real signal. The red traces correspond to the sampled signal, sampled at the frequency indicated in the title of the book. Black dots represent the positions at which the real signal was sampled.

recovered, but not their amplitude. To get a good estimate of the amplitude as well (try a sampling rate of 23 Hz to see some interesting examples of amplitude modulation), one has to sample at a minimum of twice the Nyquist frequency, or four times the signal frequency. This makes sense because at that frequency, we now have four points (or measurements) per cycle in the original signal—if properly placed, this will allow us to represent everything that matters about the periodic signal. Going much beyond that yields diminishing returns—sampling the signal at 20 times the signal frequency yields only incremental returns over doing it at four times the signal frequency, as seen in the lower right, but it looks much smoother.

What causes a lot of the confusion about sampling is that we have to distinguish three frequencies here. The first frequency we need to consider is that of reality itself. In our toy example, reality is updated a thousand times a second, which is sufficient for our purposes. The reality we live in is (as far as we can tell) updates 1.85×10^{43} times per second (Planck, 1900), or 1 over Planck time (t_p, the time it takes a photon to travel one Planck length, the smallest meaningful unit of time we [science] have been able to discern so far). The second frequency of interest is the periodic frequency of the signal. Here, our signal repeats 10 times per second. You can convince yourself that there are 10 cycles in the 1 second of time we plot. The last question is how often we take a measurement of our signal. If we take two measurements per second, our sampling frequency is 2 Hz. If we take 300 measurements in the same time period, our sampling frequency is 300 Hz. In physiology, it is not unusual for rigs to sample the signal tens of thousands of times per second (about 40 kHz) taking a measurement (in electrophysiology usually of the voltage at the electrode tip) every 25 μs, which will allow for a high fidelity representation of neural signals - as we already discussed in this book, spikes occur on the timescale of about 1 ms, so if you want to resolve the shape of the waveform, e.g. for purposes of spike sorting, you will want to sample the signal several times per ms.

It is this sampled signal that comes out of the *rig* as data and forms the basis of all further analysis, so it is critical to sample the signal at high enough of a rate to fully represent all aspects (frequency and amplitude) of the physiological signals of interest. For reasons outlined above, four times the highest signal frequency or above will do nicely.

Remember this the next time you get into a political argument with someone—you are both commenting on the same reality, but are probably sampling it differently and undersampling at that. Perhaps try to sample at a higher frequency?

So much for aliasing—back to the Fourier transform, the algorithm that does all the work.

The Fourier transform was developed by Joseph Fourier. His use case was to analyze a pressing scientific problem of his time, the heat distribution on a cannon muzzle (Fourier, 1878).

The point of the Fourier transform is that any signal in the time domain can be decomposed into a set of repeating signals with suitable frequencies, amplitudes, and phases (and vice versa—the inverse transform can transport any signal in the frequency domain back into the time domain).

Here is the equation:

$$\Im\{x(t)\} = X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft} dt \tag{5.1}$$

EQUATION 5.1 The Fourier Transform.

At first glance, this equation can look a bit scary and intimidating to the untrained eye. John von Neumann observed that "in mathematics, you don't understand things—you just get used to them" (Zukav, 2012). This seems like a copout and we could not disagree more strongly. Of course you could simply trust the mathematicians that it works and keep using it

until you get used to it, but we think it is much better to understand it, which is actually possible. To attempt this, let's first decompose it, explain the meaning of each component, then visualize what is going on. On the far left is a fraktur "F," which represents the Fourier transform itself. What is the Fourier transform transforming? A function x. Function x is a function of t (time). This function could represent a signal, e.g., a number of measurements, sampled at the sampling rate, over some amount of time. This will be a series or vector of numbers corresponding to values of some signal, measured over time. What does the Fourier transform yield? That's the term to the right of the first equal sign—it yields another function X, but this time, this is a function of frequencies f. In other words, this will be a series of power values (we already plotted some in the bottom panel of Fig. 5.2), one per frequency (not time). That is what we want—we want to transport the signal from the time domain into the frequency domain. So far, so good. How do we get here? The term to the right of the next equal sign shows how. This term represents the (input) signal—represented by the function $x(t)$ again, in the time domain. The Greek capital letter sigma stands for summation or, in this case, integration, integrating the signal that is a function of time over all time, from minus infinity to plus infinity (i.e., a long time indeed). The dt indicates that we integrate little pieces of time (infinitesimally small increments, actually). What is left to explain is the role of the exponential term, with which the signal $x(t)$ is multiplied.

First, let us elaborate on the fact that the signal *is* being multiplied with something. The transform takes the inner-product (also known as the dot-product) between the signal in the time domain and the exponential term, much akin to when the signal in the time domain was multiplied element-wise with the kernel, when we introduced convolution at the end of the last chapter. This kind of arrangement can be used to filter the signal for certain properties and to detect their presence. The color-detecting cones in the retina can be conceptualized in this way—the incoming visual stream is passed through separate color filters that measure the degree to which a certain wavelength (color) is present in the input signal (Gegenfurtner & Kiper, 2003).

But what is this filter? What is the Fourier transform looking for, when it analyzes and filters the signal? *Sine-waves*. The Fourier transform is looking for the presence of sine waves with certain frequencies and outputs the degree (this is conceptualized as "power") to which these sine waves are present in the input signal.

This is far from obvious. For starters, there is no "sine" term visible anywhere in the exponential term, and it is not clear why anyone would want to do that—filtering the signal for the presence of sine waves—in the first place.

Rest assured that the sine wave is right there, in the exponential, but is in deep disguise. There are hints though: upon closer inspection, the exponent contains the terms 2, $\boldsymbol{\pi}$, as well as f (frequency) and t (time). If this sounds familiar, it should. We recognize this as the representation of a sine wave—we took the sine of *2*pi*frequency*time* above, when plotting sine waves in Figs. 5.1 and 5.2. But where is the *sine* (that we used above) that takes these arguments? This mystery is resolved by clarifying another one (one of the rare cases where this actually works), namely why e and i are involved and what i is. Perhaps we should start with the i, to finish defining our terms. i stands for the *imaginary* part of a complex number and is defined as the square root of –1. The square root of –1 is not defined in the realm of real numbers, which is why the solution lies beyond the real numbers, in the *complex* plane. This is a good point to introduce the concept of

complex numbers, we'll need it later. In signal processing, a signal is often represented as complex numbers, where the imaginary part can be used to represent phase. See Fig. 5.6 for an illustration of complex numbers.

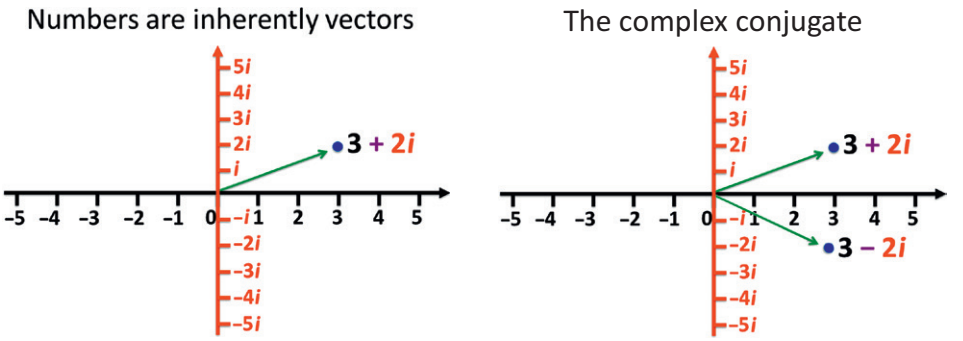

FIGURE 5.6 Complex numbers. Left panel: The complex plane. Real numbers are represented by the classical number line (here the x-axis, in black). All real numbers have no imaginary part. There is an additional axis (here, the y-axis, in red) that represents the imaginary part of complex numbers. Together, these axes span a complex plane. Whereas real numbers are scalars, complex numbers are inherently vectors with both real and imaginary parts. Right panel: The *complex conjugate*. It will be important later. It is simply the mirror image of the complex number across the number line (the real axis): **3 - 2*i*** is the "complex conjugate" of **3 + 2*i***. A fancy term for a geometrically simple idea.

Taking a closer look, we recognize that all terms involved are also present in *Euler's formula*:

$$e^{ix} = \cos x + i \sin x \tag{5.2}$$

EQUATION 5.2 Euler's formula.

This is the key—Euler's formula established the relationship between *exponential* and *trigonometric* functions (such as sine and cosine), which allows sine waves to be represented in disguise here (Euler, 1748). Don't get discouraged if you don't get this the first time around. These things are complex, literally. We suggest sleep and rereading.

But say this is true—so what? Why filter the signal for the presence of sine waves?

Unless you are a mathematician, you are probably wondering what the big deal is about sine waves (likely since around middle school). Well, this isn't about sine waves per se. It is about describing the motion of going around a circle mathematically. There are several equivalent ways to describe circular motion, one that involves decomposing the

Cartesian coordinate on the circle into an x- (cosine) and a y-coordinate (sine) and one that involves moving on the radius itself, by the phase angle (angular distance), see Fig. 5.7.

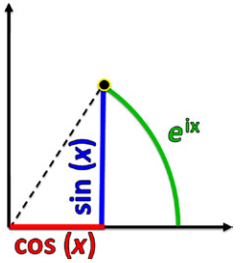

FIGURE 5.7 Moving around on a circle segment: The position on a circle described via Cartesian coordinates (red and blue) and via angular distance green end up at the same point, as we would expect from Euler's identity.

This also explains why sine waves can be described by three parameters—the amplitude (corresponding to the radius of the circle), the frequency (corresponding to how many times per second we go through the entire circle), and the phase (where on the circle we start to move). Again, sine waves simply describe circular motion. This will become more clear in Fig. 5.8.

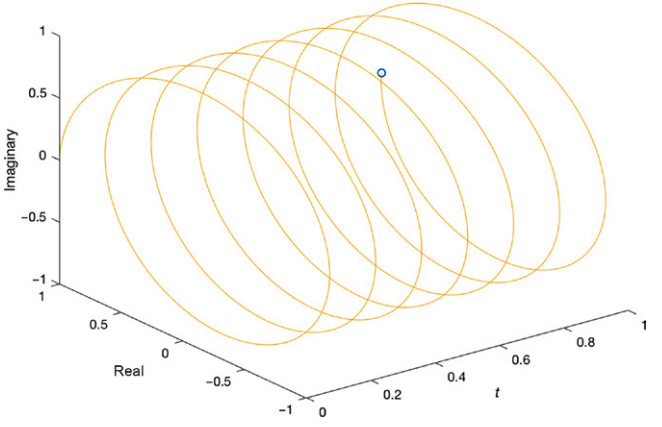

FIGURE 5.8 Tracing a complex sine wave.

By now, you have had to take a lot on faith, so this is perhaps a good point for demonstrations—and some coding. Importantly, let us convince ourselves that the exponential term in the Fourier transform really does represent complex sine waves, by plotting some.

If you are working through this chapter in one sitting and haven't cleared your workspace, this should work—we reuse some of the variables we defined above.

Pseudocode

```
1. Open a new figure
2. Making a trace, where exp stands for e
3. Show an animation of the trace, as it is being traced out
4. Add axis labels
5. Import new package
6. Specify 3d in Python
```

Python

```
from mpl_toolkits.mplot3d import Axes3D #5
fig = plt.figure() #1
ax = fig.gca(projection='3d') #6
trace = np.exp(1j*2*np.pi*freq2*t); #2
plt.xlabel('t'); plt.ylabel('real'); #4
ax.plot(t,np.real(trace),np.imag(trace))#3
```

MATLAB

```
figure %1
trace = exp(i*2*pi*freq2*t); %2
comet3(t, real(trace), imag(trace)); %3
xlabel('t'); ylabel('real'); zlabel('imaginary'); %4
```

This works out of the box. Good thing we didn't use "*i*" as an index or counter before. "*i*" and "*j*" are predefined in MATLAB. The plot of the trace should look something like Fig. 5.8. In Python, we express *i* by adding *j* to the value, as in: `np.exp(1j*2*np.pi*freq2*t)`.

Looks a lot like a spiral or slinky. If we put the plot on its ear by changing the perspective, we can get a deeper appreciation for sinusoidal as well as circular components of the trace: (Figs. 5.9 and 5.10)

Pseudocode

```
1. Find the handle of the trace in the plot
2. Make the trace black and thicker
3. Set azimuth and elevation to zero
```

Python

```
ax.plot(t,np.real(trace),np.imag(trace),c='k',lw=2) #2
ax.view_init(0, 90) #3, axes flipped in py
```

MATLAB

```
h = findobj(gcf,'type','animatedLine') %1
h(1).Color = 'k'; h(1).LineWidth = 2; %2
set(gca,'view',[0 0]) %3
```

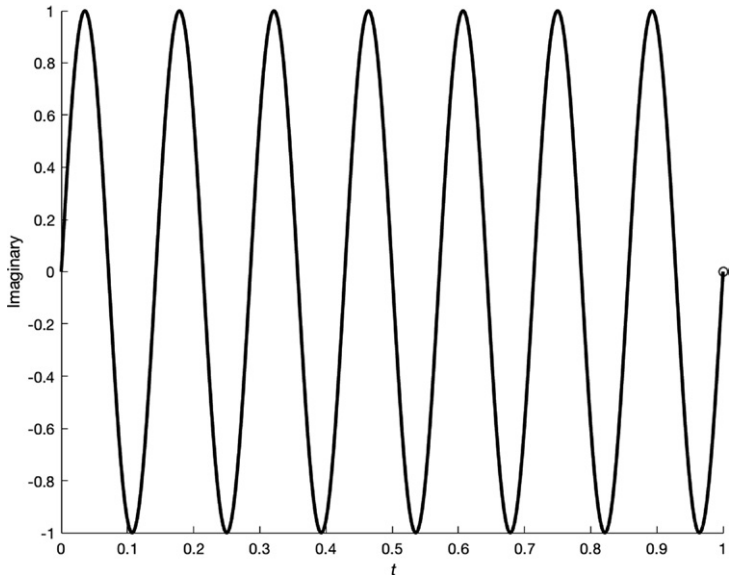

FIGURE 5.9 Looking at the trace in two dimensions—from the right vantage point—clearly reveals the sine wave. It was represented by the imaginary part of the exponential. We count seven cycles, which is what we put in. And all without ever calling the "sine" function.

```
Pseudocode
1. Change perspective to azimuth 90 and elevation zero
2. Make the axis square to avoid distortions from unequal aspect ratios
```

```
Python                                  MATLAB
ax.view_init(0, 0) #1                   set(gca,'view',[90 0]) %1
plt.axis('square') #2                   axis square %2
```

You can visualize this trace from any perspective—at will, by dragging and dropping—if you type `rotate3d` in MATLAB.

At this point, it would be hard to deny that this exponential term allows us to move on a circle or (if we include time as an axis) on a spiral.

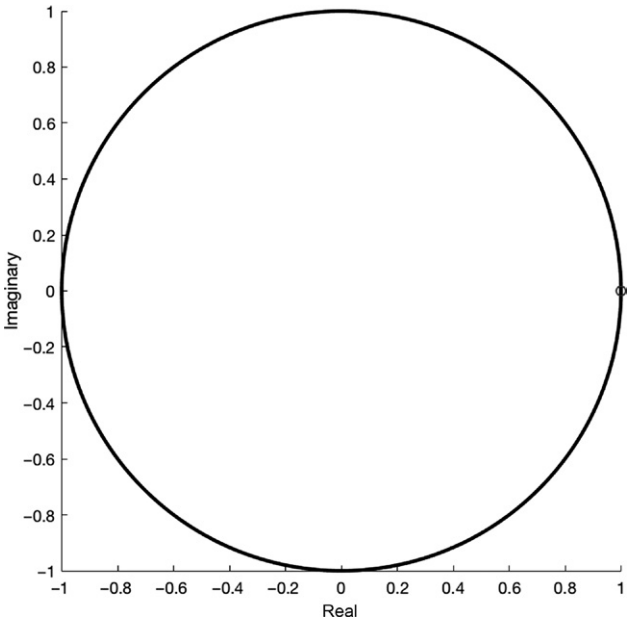

FIGURE 5.10 Collapsing over time by changing the view shows the overall circular structure of the trace, real vs. imaginary parts.

Why do circles matter? Because we can reconstruct any path or trajectory by adding enough circles—circles on top of circles or epicycles (Toomer, 1984).

This is usually more easily shown by using the *inverse Fourier transform*. We could, for instance, try to synthesize a square wave from adding adequately picked sine waves.

Properly aligned sine waves can, via *constructive* and *destructive* interference (alignment of peaks and troughs, see Glossary), recreate any signal, although to recreate a sharp edge without any ripples, we'll need to add an infinite number of them. That's impractical in practice, so there will be distortions or ripples, as you can see in Fig. 5.11—we just add three sine waves (in blue) and the resulting square wave (in black) is far from flat. More on this later.

This seems like a lot of work to devote to understand a single concept. We maintain that this is time well spent and predict that if you are seeing this for the first time and you continue in this field, you'll use Fourier transforms implicitly or explicitly for the rest of your career. It is a concept worth understanding thoroughly.

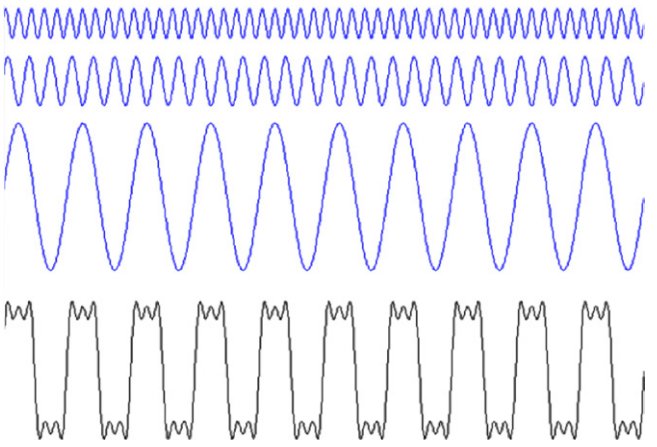

FIGURE 5.11 Synthesizing a square wave by adding suitable sine waves. Top blue wave: a. Middle blue wave: b. Bottom blue wave: c. Black trace: a+ b+ c, approximating a square wave.

The power of the Fourier transform lies in its generality. It can be (and is) applied to the analysis of a wide variety of signals, in neuroscience these include visual signals, sounds, EEG, LFP, and fMRI, to name just a few. However, it took well into the second half of the 20th century for the Fourier transform to be used so ubiquitously. The original Fourier transform is defined for analog signals, over all time. The signals we will want to analyze have been digitized and are typically varying over time—how the frequency content of a signal might change over time, e.g., in response to a stimulus, is exactly what will be of interest. The fast Fourier transform (FFT) algorithm introduced by Tukey allows us to do a *discrete Fourier transform* (i.e., on a digital, or nonanalog signal) on a short time window in which the frequency content is presumably stable and (as the name suggests) allows us to do this quickly and without needing an infinite number of sine waves.

Which brings us to the *spectrogram*, which is a visual representation of power at different frequencies over time. Conceptually, it corresponds to a stack of FFT outputs (like the ones we did to produce Fig. 5.3), stacked in time. But it isn't quite the output of the FFT itself that we are plotting. We are plotting power instead. Power is defined as the output of the FFT multiplied by its complex conjugate. An equivalent version of this is taking the absolute value of the squared FFT output, which also corresponds to power. Both operations get rid of phase information. This means that the spectrogram only uses the information that is contained in the amplitudes. A complementary plot would be a phase coherence plot (that uses only phase information), but that is "beyond 1" material—we have some references on that at the end of this chapter.

The spectrogram results from doing the FFT on the snippet of the signal that falls into a "window," plotting the frequency content in the window, then moving the window in time and plotting the frequency content again (and again) until the window has moved across the entire signal.

"The spectrogram" of a signal is somewhat of a misnomer. "A spectrogram" of a signal would be more apt. As you will see shortly, two spectrograms of the same signal can look dramatically different, depending on our choices of window and how far to move the window before doing the FFT again. Important window choices are *width* and *shape*.

But we are getting ahead of ourselves. Before we get lost in abstract discussions of signal processing concepts that mean nothing to you unless you have encountered them before, let us create a spectrogram of a signal we already have, then discuss these matters with specific code and figures (Fig. 5.12). The following code presumes that you have the signal processing toolbox installed in MATLAB. In Python, we will import the `scipy.signal` library.

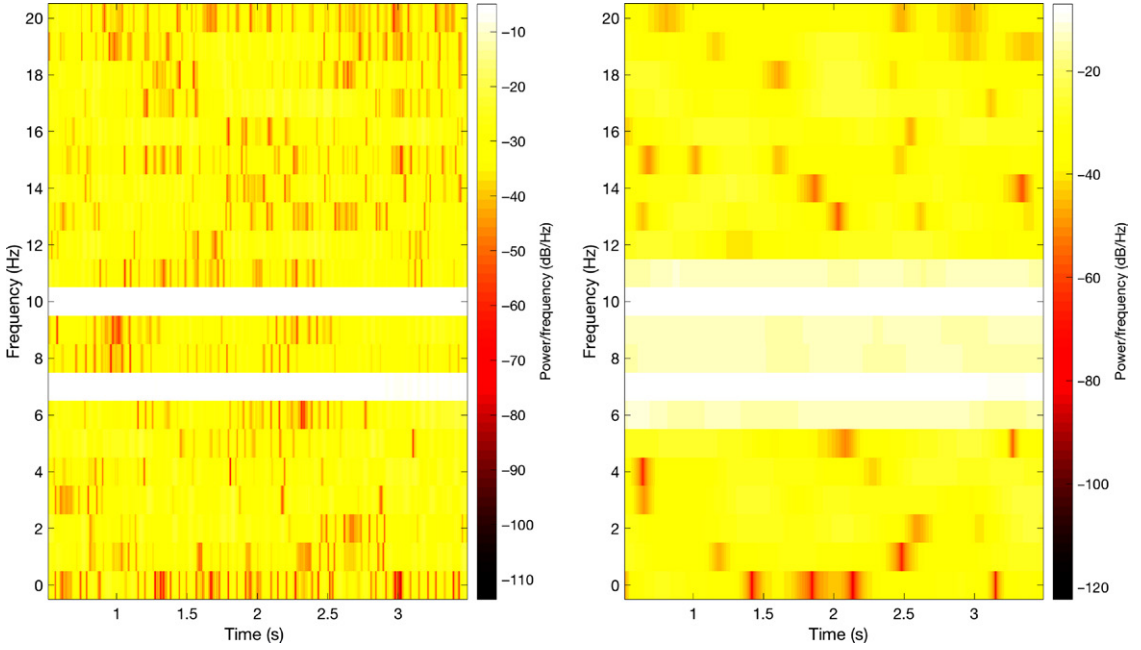

FIGURE 5.12 Spectrogram of stationary signals with a window of adequate width. Left: Using a Kaiser window. Right: Looking at the same data through a Hanning window.

Pseudocode

1. Define the number of elements in the window, here 1024
2. Create a Kaiser window of that length
3. Define that the overlap is maximal (1023 elements)
4. Define the frequencies we want to assess and plot on the *y*-axis
5. Open a new figure
6. Open a new subplot
7. Do the spectrogram with the right parameters. In addition to the ones we defined, we also supply the sampling rate and the parameter "yaxis" to put it into the format we want (frequencies on the *y*-axis, time on the a-axis) (default to –1 axis in Python)
8. Define a new window of the same length, a "Hanning" window
9. Open a new subplot
10. Do the spectrogram on the same signal, with the same parameters, but a different window
11. Change colormap to hot
12. Show graph
13. Import Python Package
14. Plot colormesh
15. Label axis
16. Set y-limits

Python

```
import scipy.signal as sg #13
windLength = 1024; #1
wind = np.kaiser(windLength,0); #2
overl = len(wind)-1; #3
yFreqs = range(21); #4
fig = plt.figure() #5
plt.subplot(1,2,1) #6
f, tt, Sxx =sg.spectrogram(signalAndNoise,fs,wind,len(wind),overl) #7
plt.pcolormesh(tt,f,Sxx,cmap='hot') #!4
plt.ylabel('Frequency (Hz)');plt.xlabel('Time (sec)') #15 label axes
plt.ylim([0,20]) #16
wind = np.hanning(windLength);#8
plt.subplot(1,2,2) #9
f, tt, Sxx =sg.spectrogram(signalAndNoise,fs,wind,len(wind),overl) #7
plt.pcolormesh(tt,f,Sxx,cmap='hot') #14
plt.ylabel('Frequency (Hz)');plt.xlabel('Time (sec)')#15 label axes
plt.ylim([0,20]) #16
```

MATLAB

```
windLength = 1024; %1
wind = kaiser(windLength); %2
overl = length(wind)-1; %3
yFreqs = 0:20; %4
figure %5
subplot(1,2,1) %6
spectrogram(signalAndNoise,wind,overl,yFreqs,fs,'yaxis'); %7
wind = hanning(windLength Switch; and)%8
subplot(1,2,2)%9
spectrogram(signalAndNoise,wind,overl,yFreqs,fs,'yaxis'); %10
colormap(hot) %11
shg %12
```

This might be one of the few times where using a *Kaiser* window is actually superior. It allows us to clearly resolve the two frequencies we know to be present (we made the signal when creating Fig. 5.3), whereas the two frequencies are kind of blending together when looking at the same signal through the *Hanning* window.

Different windowing functions emphasize different things and differ in how much and how fast they attenuate *side-lobes* of power and how much they distort the signal. The need for using a proper window arises from the fact that the Fourier transform itself is defined for infinite time. If we split the signal into pieces and look at each piece separately, this will introduce *edge artifacts*, similar to the edge effects seen with convolutions in Chapter 4, Correlating Spike Trains. The name of the game in windowing is to pick the right window to avoid these edge artifacts as much as possible. Recall that it is hard to create sharp edges with a finite number of sine waves. In other words, a sharp window will cause rippling in frequency space (the Fourier transform of a square wave [or boxcar] function is a *sinc* function). So there are various windows that try to amplify the center of the window, but arrive at the edge in a smooth fashion. Again, we recognize that this is rather abstract. In MATLAB, you can simply look at the windows and their characteristics by typing `wvtool(wind)` (see Fig. 5.13).

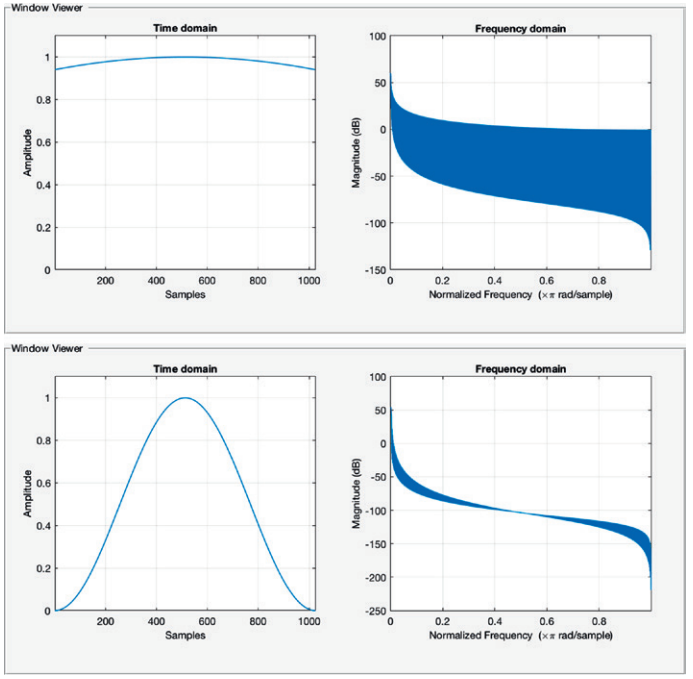

FIGURE 5.13 Top: The Kaiser window we used. Bottom: The Hanning window we used. Left panels: The window characteristics in the time domain. Note that the Hanning window goes to zero, but the Kaiser window does not. Right panels: The window characteristics in the frequency domain. The Kaiser window produces a lot of rippling in the side-lobes, which is why we don't usually want to use it. You can see this rippling in the noise of Fig. 5.12. The noise in the random part of the signal (outside of frequencies 7 and 10) appears much more "jumpy" when looking through a Kaiser window, whereas the Hanning window seems to smooth and calm things down quite a bit outside of the signal. The price to pay for this is that it blends the signal a bit more—7 and 10 Hz are harder to resolve.

Generally speaking, you'll want to use window lengths that are powers of 2 - (2, 4, 8, 16, 32, 64, etc.) because they are faster to compute, for algorithmic reasons in the FFT. Here, we use a rather wide window with 1024 elements. But we are looking at a signal that doesn't change over time. We know this because we created the signal ourselves to make Fig. 5.3. We also know from Fig. 5.3 (bottom panel) that if we use the entire signal of 4001 elements, we can resolve that there is a narrow signal at 7 Hz and a signal at 10 Hz without a problem, even in the presence of considerable noise. But note that this is asking a lot—the Nyquist frequency is 500 Hz, so we are interested in minute differences in frequency. If we use a window that is too narrow, the noise will prevent us from being able to resolve this difference. Even halving the window length will be too drastic to resolve these frequencies in the Hanning window, and even the Kaiser window is starting to have trouble. Run the code above again, but set `windLength` to 512, to yield Fig. 5.14.

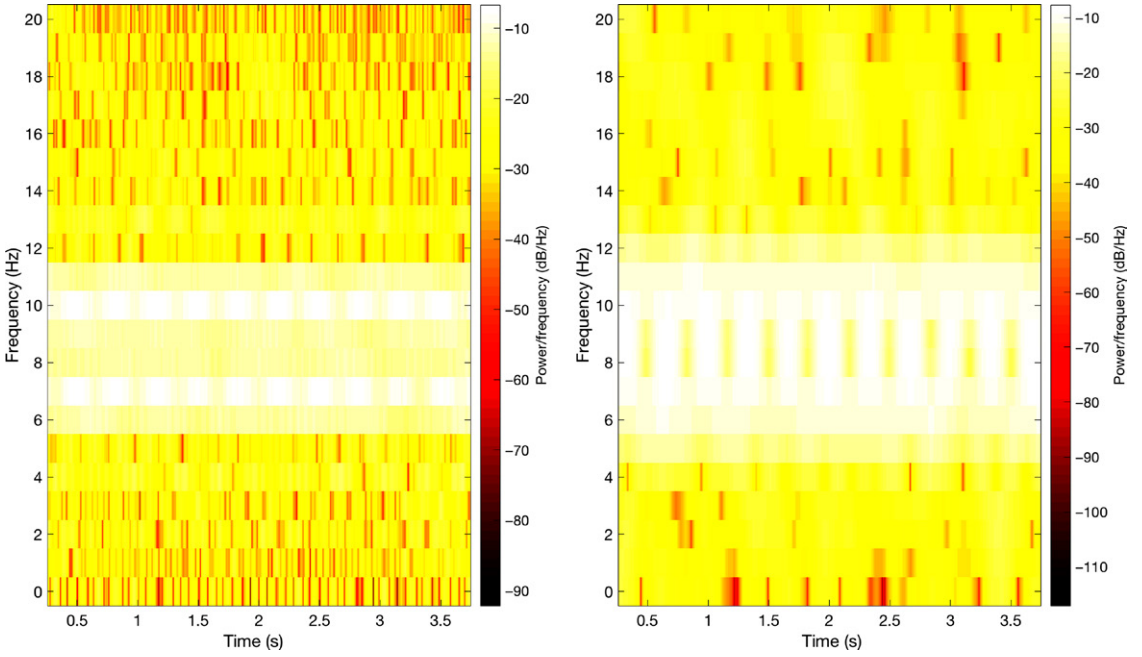

FIGURE 5.14 Spectrogram of the same signals as in Fig. 5.9, using the same parameters, but with half the window length. The Hanning window is starting to fail to resolve the difference between the two signal frequencies (7 and 10 Hz).

Generally speaking, there is a fundamental tradeoff (a kind of uncertainty principle) between determining the frequency content of a signal and determining when this frequency content occurred. The longer a window is, the better the estimate of the frequencies it contains. There are more data to go around, more noise to cancel out and so on. The price for this is a long window, so it is unclear when any of this happened (in the limit, we use a single window for the entire signal, as in the simple FFT, where we have no idea when any of this happened). In contrast, a very narrow window will be quite precise about when the frequencies happened, but not which ones. Any spectrogram will have to make a reasonable choice between these two extremes. Doing this in a principled fashion is a challenge and an active area of ongoing research (Casey, 2012). An alternative approach is not to commit to any particular window, but simply do multiple ones at the same time, then average them, usually using Slepian functions (Mitra, 2007). This "multitaper" approach is reasonable, but in practice may yield spectrograms that appear a bit "blobby."

So far, we have only considered spectrograms of *stationary* signals, in order to understand what spectrograms do, but the point of doing spectrograms is to look at signals that vary in frequency over time—if the signal doesn't change, you can just do a single Fourier transform over the entire signal. So let us move on to spectrograms of signals where the frequency content changes over time—if the signal doesn't change, you can just do a single Fourier transform over the entire signal. One of the simplest such signals is a "chirp," which has recently gained tremendous scientific publicity, as it seems to be the signal emitted from colliding black holes, relevant for the detection of gravity waves (Abbott et al., 2016).

`Pseudocode`

1. Create a new time base, a 2-second signal
2. Create a chirp over that time, one that starts at 100 Hz, crosses 200 Hz at 1 second and is quadratic
3. Listen to it
4. Open a new figure
5. Open a new subplot
6. Plot amplitude of the signal over time
7. Open a new subplot
8. Set window length to 128
9. Set overlap to 127
10. Determine that we want to show 250 frequency bins on the y axis
11. Do a spectrogram of the signal with a 128 element Kaiser window
12. Open a new subplot
13. Do a spectrogram of the signal with a 128 element Hanning window
14. Open a new subplot
15. Do a spectrogram of the signal with a 128 element Chebichev window

(*Continued*)

```
Python          time = np.linspace(0,2,fs*2)#1
                y=sg.chirp(time,100,1,200,'quadratic'); #2
                #3 playing sounds is beyond "1" in Python
                fig=plt.figure(figsize=(10,10))#4
                ax = plt.subplot(4,1,1)#5
                ax.plot(time,y) #6
                ax = plt.subplot(4,1,2) #7
                windLength = 128; #8
                overl = windLength-1; #9
                freqBins = 250; #10
                wind=np.kaiser(windLength,0)
                f, tt, Sxx =sg.spectrogram(y,fs,wind,len(wind),overl); #7
                plt.pcolormesh(tt,f,Sxx);
                ax = plt.subplot(4,1,3) #12
                wind=np.hanning(windLength);
                f, tt, Sxx =sg.spectrogram(y,fs,wind,len(wind),overl); #7
                plt.pcolormesh(tt,f,Sxx)
                ax = plt.subplot(4,1,4); #14
                wind=sg.chebwin(windLength, at=100);
                f, tt, Sxx =sg.spectrogram(y,fs,wind,len(wind)); #7
                plt.pcolormesh(tt,f,Sxx);
MATLAB          time = 0:1/fs:2; %1
                y=chirp(time,100,1,200,'quadratic'); %2
                sound(y,fs)%3
                figure %4
                subplot(4,1,1) %5
                plot(time,y) %6
                subplot(4,1,2) %7
                windLength = 128; %8
                overl = windLength-1; %9
                freqBins = 250; %10
                spectrogram(y,kaiser(windLength),overl,freqBins,fs,'yaxis'); %11
                subplot(4,1,3) %12
                spectrogram(y,hanning(windLength),overl,freqBins,fs,'yaxis'); %13
                subplot(4,1,4) %14
                spectrogram(y,chebwin(windLength),overl,freqBins,fs,'yaxis'); %15
```

This code will yield Fig. 5.15.

Some notes: Fig. 5.15 nicely illustrates the benefits of looking at a spectrogram. It is obvious that the frequency starts at 100, then is rising over time, in a quadratic fashion, which corresponds to what we hear when we listen to the signal.

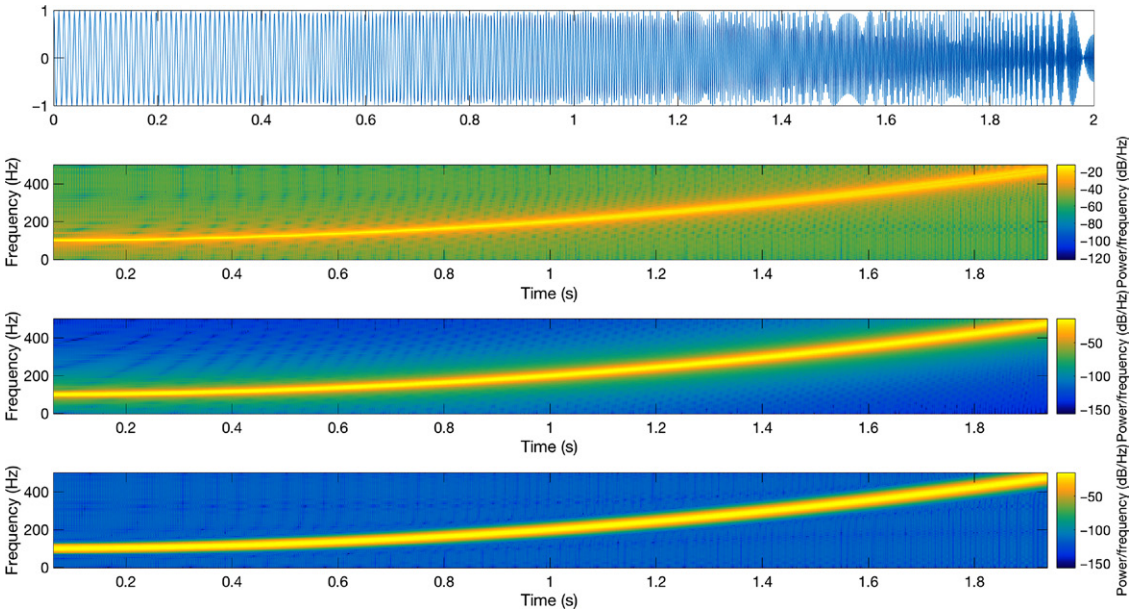

FIGURE 5.15 Looking at the same signal—a time varying quadratic chirp in the time domain (top panel), through a Kaiser window (second panel), a Hanning window (third panel) and a Chebichev window (bottom panel). Again, note the power of looking at power (over time). Powerful. Brave.

Looking at the same signal in the time domain (top panel) is hopeless. The Chebichev window looks a lot like a Gaussian (use the `wvtool` to confirm this) and seems to perform best here—with a strong suppression of the power in the noise and a clearly defined signal. The results of doing the signal processing with a Kaiser window are afflicted by brutal noise, which is characteristic of the Kaiser window—it usually produces severe rippling in the side lobes. The Hanning window performs somewhere in between. Instead of specifying which frequencies we want to plot on the y-axis, we specified that we want 250 divisions (which will correspond to 2 Hz bins, given the Nyquist frequency).

So far, we have always used almost complete window overlap, i.e., an overlap of 511 elements if the window length was 512 or an overlap of 127 if the window length was 128. This leads to plots with the most resolution, but is computationally the most expensive. If you are in a rush, use less overlap. This will lead to blocky plots, but they are done much faster. For an extreme case of this, see Fig. 5.16—we use the same signals and do spectrograms with the same windows, but no overlap at all. This figure is produced by setting `overl` to 0, but otherwise running the same code that produced Fig. 5.15:

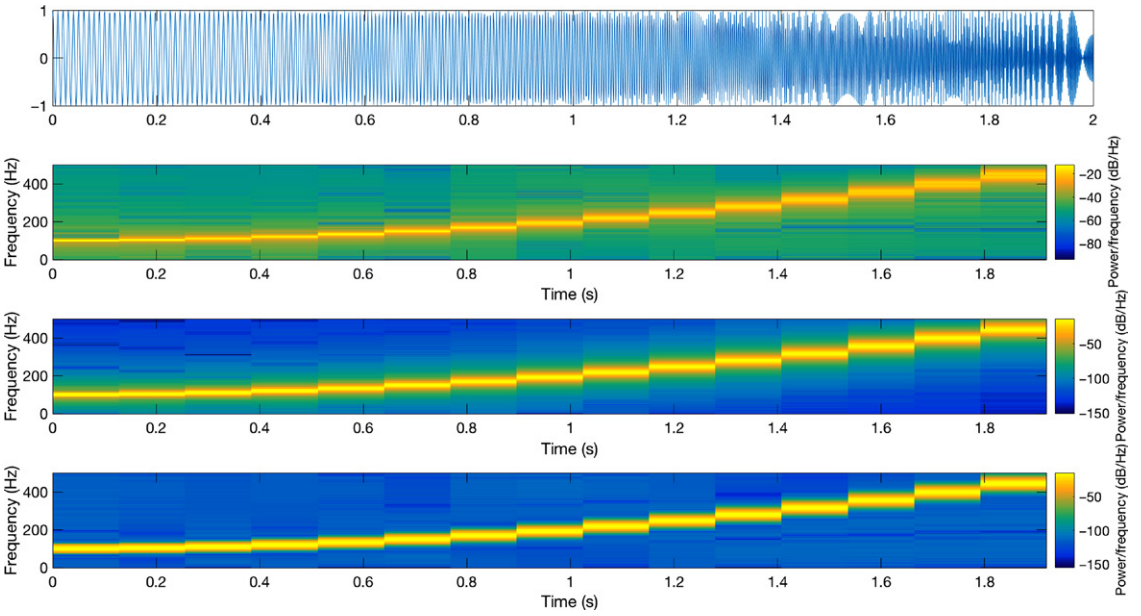

FIGURE 5.16 Same signal and windows, but no overlap. As you can see, the spectrogram is now very "blocky." The signal is simply divided into pieces of 128 elements and the fft is computed on those elements, then the window is advanced by an entire 128 elements and the fft is done again. That's what yields the discontinuities between windows. Generally speaking, spectrograms with more overlap will appear much smoother, but take a lot longer to compute.

This is a good time to introduce the notion of *filtering*. Understand that we can only scratch the very surface of this here—explaining why it is done, what it is, and doing a simple example of it. For a more comprehensive treatment of filtering, see Bibliography. We want to emphasize this because there is no way we can do the topic justice on a few pages (which is all we have here), people get entire degrees in this kind of thing. None of it is obvious or intuitive, all of it is complicated.

Filtering of signals also occurs in nature, more or less explicitly. When lightning strikes far away and you hear the rumbling thunder some time later, the atmosphere has effectively performed a low-pass filtering of the signal, emphasizing lower frequencies as they carry farther in air—higher frequencies lose power much faster. The uterus similarly acts as a low-pass filter—strongly attenuating frequencies above 300 Hz—keep that in mind next time you sing or talk to your fetus (Querleu, Renard, Versyp, Paris-Delrue, & Crèpin, 1988; Abrams et al., 1998; Gómez et al., 2014). When you use an edge detection algorithm in Photoshop, you are running a high-pass filter on the image, under the hood. When

you record data with your electrophysiology rig and don't want to be swamped by line noise, you'll need to use a band-reject filter centered on 60 Hz (in the United States; 50 Hz in Europe).

So it is worth understanding what is going on. We'll re-use the signals we already have, but now create versions that have been filtered in different ways.

There are many filter types in use in electrical engineering and signal processing. The one we will cover here is the one we think you are most likely to use in practice, namely the *Butterworth* filter. It has several nice characteristics, including a flat frequency response in the *passband* (the region of frequencies you wish to not get filtered out), with power gently falling off outside of it—as smooth as butter, without introducing distortions or ripples (Butterworth, 1930). The ideal filter would completely pass everything in the passband (i.e., allow through the parts of the signal we care about) and completely reject everything outside of it, but this cannot be achieved in reality—the Butterworth filter is a close approximation. Other filters, such as the Chebishev filter, reject frequencies outside of the passband much more sharply, but this comes at the price of ripples in the passband itself, which is often unacceptable.

Let's create four filters, two Butterworth and two Chebishev, to illustrate this. Filters are defined by their filter coefficients (usually called "A" and "B"). In the code below, we first make a fifth order, then 10th order filters. In software, the order of a filter pertains to the number of polynomials that are involved, in hardware the number of resistors and capacitors.

`Pseudocode`

1. Create a fifth order Butterworth low-pass filter that passes the lowest ⅕ of the frequency range below Nyquist (1 would be the Nyquist frequency)
2. Create a 10th order Butterworth low-pass filter that passes the lowest ⅕ of the frequency range below Nyquist
3. Create a fifth order (type I) Chebichev low-pass filter that passes the lowest ⅕ of the frequency range below Nyquist and accepts 7 dB of ripple
4. Create a 10th order (type I) Chebichev low-pass filter that passes the lowest ⅕ of the frequency range below Nyquist and accepts 7 dB ripple
5. Python code to approximate MATLAB's `fvtool` function

`Python`

```
b1_low, a1_low = sg.butter(5, .2, 'low', analog=True)#1
b2_low, a2_low = sg.butter(10, .2, 'low', analog=True)#2
b3_low, a3_low = sg.cheby1(5, .2, 100, 'low', analog=True)#3
b4_low, a4_low = sg.cheby1(5, .2, 100, 'low', analog=True)#4
w, k = sg.freqs(b3_low, a3_low) #5
plt.semilogx(w, 20 * np.log(abs(k))) #5
```

`MATLAB`

```
[B1_low,A1_low] = butter(5,0.2,'low'); %1
[B2_low,A2_low] = butter(10,0.2,'low'); %2
[B3_low,A3_low] = cheby1(5,7,0.2,'low'); %3
[B4_low,A4_low] = cheby1(10,7,0.2,'low'); %4
```

Note that the Chebichev filter requires four parameters—that's because we have to specify what degree of ripple (in dB) in the passband is acceptable. The more ripple we deem acceptable, the more sharply the Chebichev filter is able to suppress frequencies outside of the passband. As the Butterworth filter is maximally flat in the passband, we don't need to specify that parameter for Butterworth. Fig. 5.17 was created by looking at the filter characteristics with the filter visualization tool, typing `fvtool(B,A)`, where B and A stand for the suitable coefficient from the 8 above. Note that this tool plots the *y*-axis in log scale, so the falloff of the Chebichev filter is more dramatic than it looks.

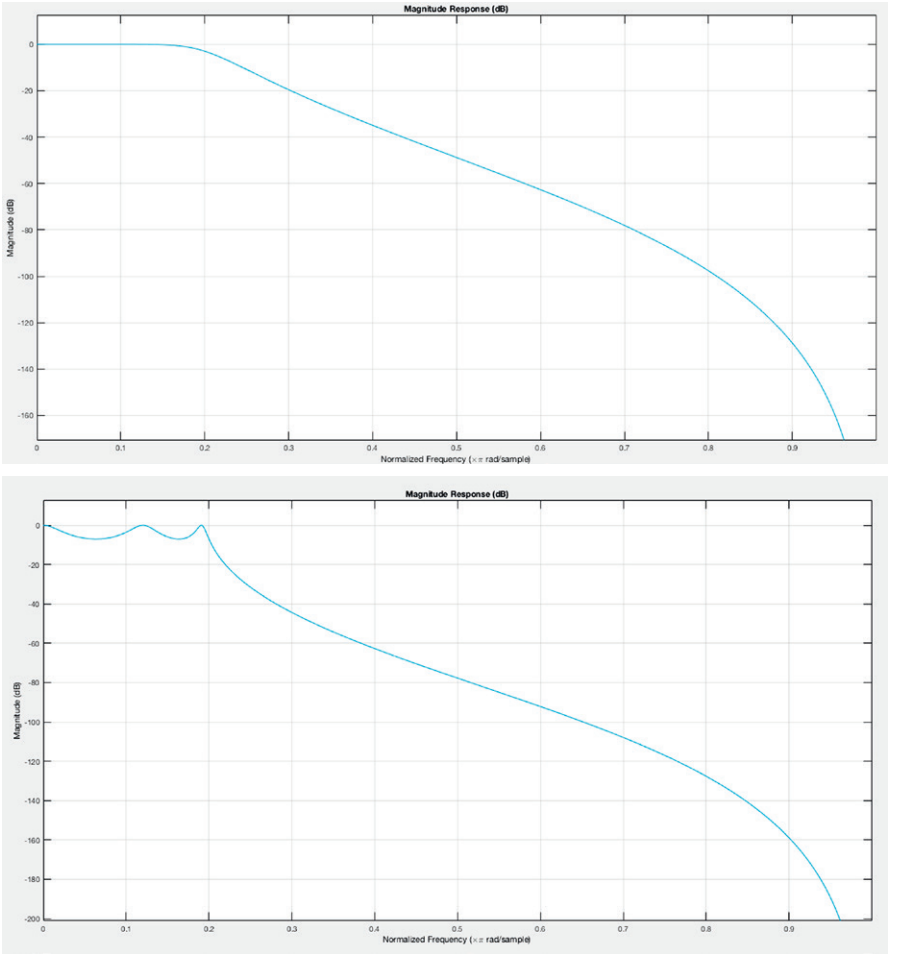

FIGURE 5.17 Filter characteristics of fifth order (top panel) and 10th order (bottom panel). Butterworth (left) and Chebichev I (right) filters.

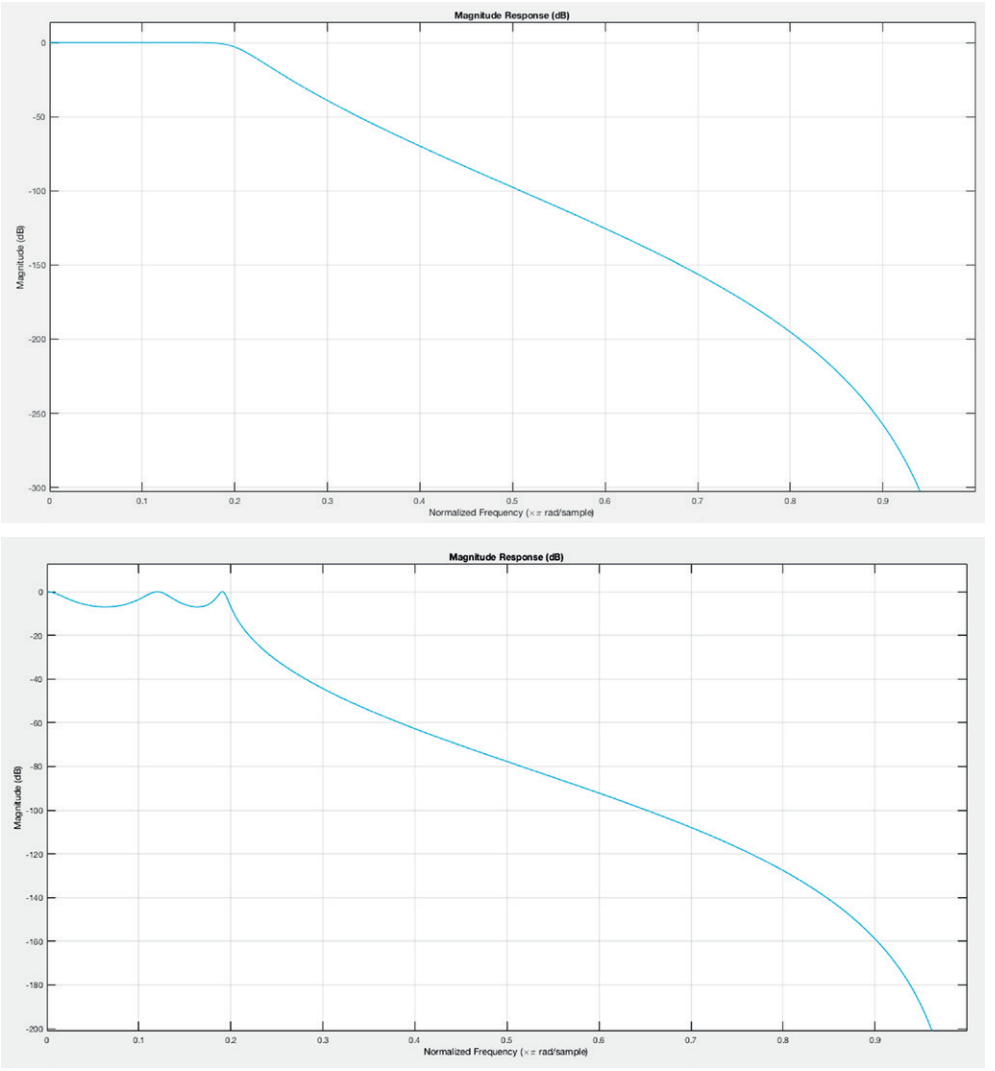

FIGURE 5.17 (*Continued*)

As you can see, the lower-order Butterworth filter starts attenuating even before the end of the passband and falls off rather gradually. The higher-order Butterworth filter starts attenuating much closer to the passband and drops off steeper. The Chebichev filters ripple along but achieve more precipitous rejection outside of the passband than the Butterworths. Which one you want to use in practice depends on your use case.

For now, let's adopt the eighth order Butterworth filter in order to do some filtering of the signal.

If you listen to this and compare it to the original chirp, filtering with Butterworth simply corresponds to an attenuation of amplitude, the frequency content of the sound is not distorted.

Pseudocode

1. Create an eighth order Butterworth low-pass filter with a cutoff at 0.6 Nyquist
2. Create an eighth order Butterworth high-pass filter with a cutoff at 0.4 Nyquist
3. Filter the chirp signal with the low-pass filter
4. Filter the chirp signal with the high-pass filter
5. Filter the filtered signal we created in (4) again with a low-pass filter to create a band-pass filtered version of the signal

Python

```
B_low,A_low = sg.butter(8,0.6,btype='low',analog=True) #1
B_high,A_high = sg.butter(8,0.4,btype='high',analog=True) #2

winds = {}
winds['y'] = y
winds['yLow'] = sg.filtfilt(B_low,A_low,y); #3
winds['yHigh'] = sg.filtfilt(B_high,A_high,y);# %4
winds['yBand'] = sg.filtfilt(B_low,A_low,winds['yHigh']);#5
```

MATLAB

```
[B_low,A_low] = butter(8,0.6,'low') %1
[B_high,A_high] = butter(8,0.4,'high') %2
yLow = filtfilt(B_low,A_low,y); %3
yHigh = filtfilt(B_high,A_high,y); %4
yBand = filtfilt(B_low,A_low,yHigh); %5
```

Note that simple filtering would cause a phase-shift of the filtered signal. We don't want that, so we filter twice, once forward and once backward, in order to yield a filtered, but not phase-shifted version of the signal. In MATLAB and `scipy.signal` this is achieved by the function `filtfilt`.

Now, let's look at and listen to these filtered signals and compare to the original (Fig. 5.18).

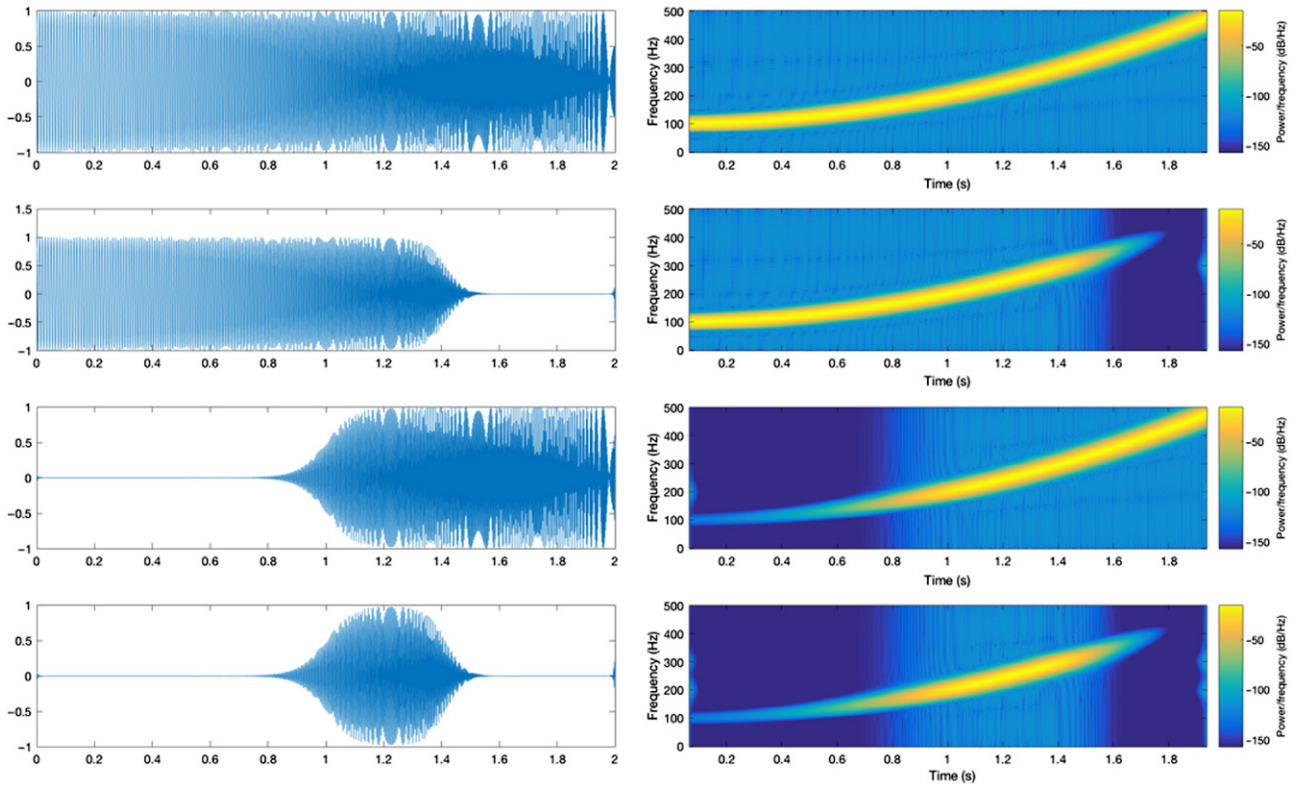

FIGURE 5.18 The effects of filtering. Top panel: original chirp. Second panel: low pass. Third panel: high pass. Bottom panel: band-pass. Left-panels: Signal as amplitude over time. Right panels: Spectrograms.

Pseudocode

1. Open figure
2. Higher overlap, so that spectrograms are smooth again
3. Define a cell of four signals: The chirp and filtered versions thereof
4. Initialize a counter
5. Open the subplot that corresponds to the counter number
6. Check if the counter is odd or even with the mod function
7. If it is odd, plot the amplitude of the suitable signal over time and increment the counter
8. If it is even, plot the spectrogram of the suitable signal, play the sound that correspond to it and wait for an appropriate length of time, then increment the counter (code not included in Python)
9. Loop over column/types of plot
10. Loop over rows/signals

Python

```
fig=plt.figure(figsize=(10,10))#1
overl = windLength-1; #2
signals = ['y','yLow','yHigh','yBand']; #3
counter = 1; #4
for ii in range(4): # 10
    for jj in range(2):# 9
        ax=plt.subplot(4,2,counter) # 5
        if counter%2 == 1: # 6
            ax.plot(time,winds[signals[ii]]) # 7
            counter = counter + 1; # 7
        else : #6
            f,tt,Sxx=sg.spectrogram(winds[signals[ii]],fs)
            plt.pcolormesh(tt,f,Sxx);
            counter = counter + 1; # 8
```

MATLAB

```
figure %1
overl = windLength-1; %2
signals = {'y','yLow','yHigh','yBand'}; %3
counter = 1; %4
for ii = 1:4 %10
    for jj = 1:2 %9
        subplot(4,2,counter) %5
        if mod(counter,2) == 1 %6
             plot(time,eval(signals{ii})) %7
            counter = counter + 1; %7
        else     %6
spectrogram(eval(signals{ii}),chebwin(windLength),overl,freqBins,fs,'yaxis'); %8
            sound(eval(signals{ii}),fs) %8
            pause(length(eval(signals{ii}))/fs) %8
            counter = counter + 1; %8

        end %6
    end %9
end %10
```

Note that the Butterworth filter is so smooth that we never quite cut the power to zero on the low end. If you feel adventurous, try a higher-order Butterworth filter or a Chebichev filter and see if you can get it to zero. Also note that we implemented the MATLAB code in a nested loop here. Per "eval," we modified what is plotted per iteration, instead of writing things out four times.

That's it for signal processing concepts. Let us apply what we learned to real data. In this case, local field potentials (LFPs) that were recorded in the auditory cortex in response to sound signals. LFPs are the lower-frequency components in the voltage signal. If spikes have most of the power above 1000 Hz, LFPs are defined as the spectral component of the voltage signal below 1000 Hz, usually well below that—with most of the action below 100 Hz (anything above that is considered to be *high gamma*). Whereas voltage spikes correspond to the outputs of single neurons, LFPs are thought to reflect the synchronized inputs to large populations of neurons, as they reflect *EPSPs* and *IPSPs* in their apical dendrites (Logothetis, Pauls, Augath, Trinath, & Oeltermann, 2001). How "local" the local field potential is, is subject to active debate (Kajikawa & Schroeder, 2011).

Thus far, we have built all the programs from the bottom up, step by step. From now on, some of the more advanced programs in each chapter will be too complex to do so effectively, as we are drawing from many concepts and building blocks. Instead, we will provide you with programs we wrote, programs that are heavily commented. It is useful to try to reverse-engineer them.

The program to do the LFP analysis can be downloaded from our companion website in POM. Here is what the program does, in brief.

First it loads the workspace "`mouseLFP.mat`." The data in the file were recorded in four electrophysiological recording sessions from mouse auditory cortex, one session for each row in the DATA cell. The experiment itself consisted of presenting 200 auditory stimuli (each presented for 50 ms, with a 500 ms interstimulus interval, sampled from two tones, in random order). Data were sampled at 10 kHz. The seven columns represent (in order): (1) Voltage snippets per trial (tone-locked). (2) Background noise per trial (outside of stimulation). (3) Raw voltage trace over the entire session (not cut up). (4) Trace of triggers over the entire session (when tone was on, 1 if on, 0 if not). (5) List of tones, in order presented. (6) Tone on- and offset-time over the entire session. (7) Recording site, date, etc.

We then filter the signal with a low-pass Butterworth filter with a cutoff frequency at 1000 Hz. This is necessary because the DATA file contains the all-pass, broadband raw voltages. This includes spike information, but we want to look at the LFP alone, which is classically defined as being below 1000 Hz. We will use this filtered signal for analysis and plotting. This is a classical instance of the "filtering/cleaning" stage of the canonical processing cascade. The program then makes one figure per recording session, for a total of four figures. Each figure contains four subplots: two subplots for the ERP for each tone (including an error band), and two subplots for the corresponding spectrograms (one for each tone). We plot frequencies from 0 to 200 with a 5 Hz bin-width.

See Fig. 5.19 for one of the four figures that this program produces.

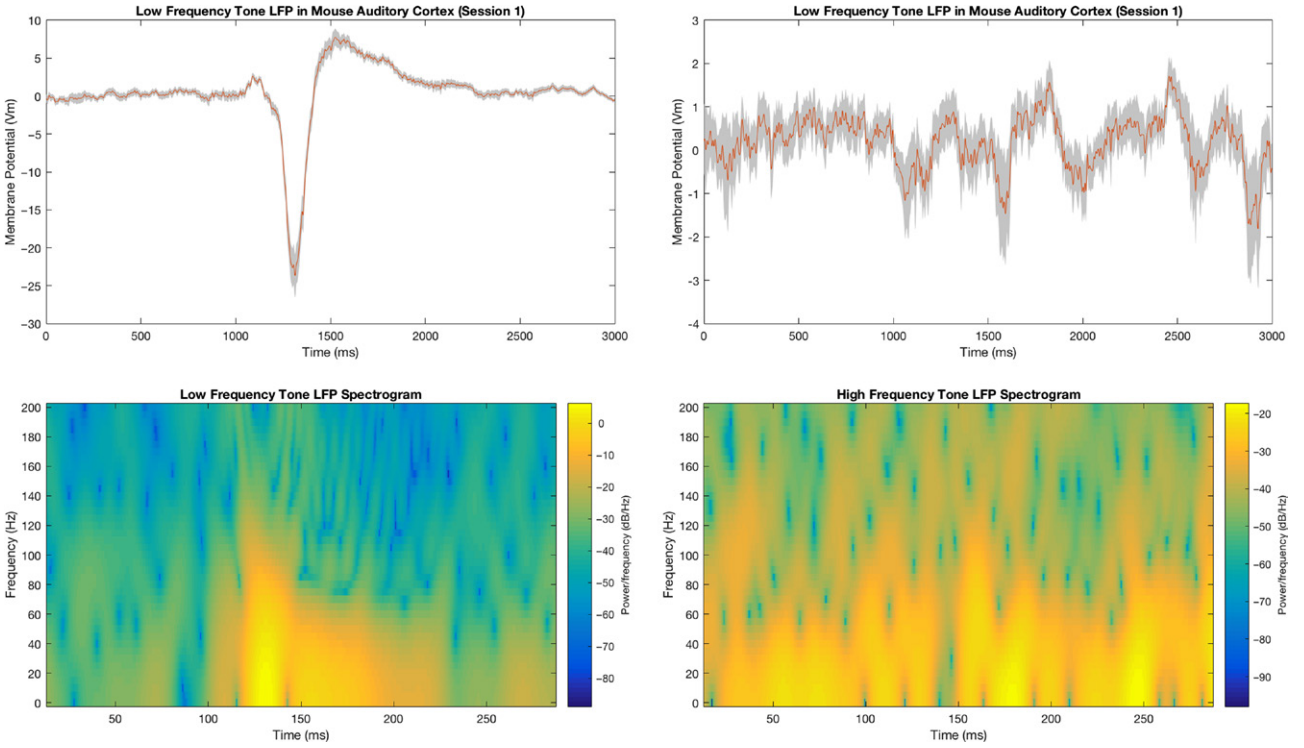

FIGURE 5.19 Plots from recording session 1. Top row: Event-related potentials (ERPs). Gray error bands represent the standard of the mean (SEM). Bottom row: Spectrograms. One way to conceptualize this is to think of them as the "singing of the heavenly choirs." ⌘Each chorister sings at a particular frequency, e.g., 100 Hz. The more yellow the trace is at a given time, the louder that frequency is sung at that time. The linear combination of all these tracks produces the signal that you can see (or hear, if you listen to the data at the sampling rate, which we recommend). Left column: Response to the low tone. Right column: Response to the high tone. As you can see, the neuron is tuned—there is a brisk response to the low, but not the high, tone in the LFP.

That's it. We believe we got you from 0 to 1 in signal processing of analog signals. To go beyond that and learn about such exciting topics as the Morlet wavelet, the filter Hilbert method, phase coherence, and multitaper methods, we warmly recommend a close read of Mike X Cohen's "Analyzing neural time series data" (2014).

⌘. *Pensee on harmonic oscillators in the brain:* We have noted in a previous chapter that it is fundamentally inappropriate to express spike rates in units of Hz. A power analysis of EEG, MEG, and LFP data is also somewhat puzzling. For instance, we determine which harmonic oscillators are most active within an EEG trace, e.g., those oscillating around 10 Hz if most of the power was in the alpha band or around 6 Hz if most of the power was in the theta band, *if there were harmonic oscillators.* To our knowledge, there are no neurons in the cortex that spike in any way that could meaningfully be described as purely sinusoidal. If there are any in the brain, they might be situated in subcortical structures like the suprachiasmatic nucleus—a small population of neurons that control circadian rhythms that discharge at a very low frequency—about 1 cycle a day (Buhr et al., 2010) or the olfactory bulb, a key structure in odor perception (Rojas-Libano & Kay, 2008). There is no question that neurons exchange information in the form of spikes, but the theoretical significance of these oscillations is extremely controversial, with positions ranging from high-powered signals in EEG and LFP dominating what happens in the brain, as they reflect synchronized activity that readily propagates through cortex (Fries, 2005, 2009) to the position that these are simply signals we happen to be able to measure conveniently and noninvasively, but that are fundamentally epiphenomenal in nature and have no role in the way neural populations communicate (Shadlen & Movshon, 1999). It is fair to say that the functional role of these oscillations (and their neural correlate—networks?) is still very much in question and subject to active research debate (Salinas & Sejnowski, 2001; Buszaki and Draguhn, 2004; Uhlhaas et al., 2009), far beyond this brief note and indeed the entire book.