**Chapter 8. Spectral analysis.**

The material in this chapter provides an introduction to the analysis of speech data that has been transformed into a frequency representation using some form of a Fourier transformation. In the first section, some fundamental concepts of spectra including the relationship between time and frequency resolution are reviewed. In section 8.2 some basic techniques are discussed for reducing the quantity of data in a spectrum. Section 8.3 is concerned with what are called spectral moments that encode properties of the shape of the spectrum. The final section provides an introduction to the discrete cosine transformation (DCT) that is often applied to spectra and auditorily-scaled spectra. As well as encoding properties of the shape of the spectrum, the DCT can also be used to remove much of the contribution of the source (vocal fold vibration for voiced sounds, a turbulent airstream for voiceless sounds) from the filter (the shape of the vocal tract) from which a smooth spectrum can be derived.

## 8. 1 Background to spectral analysis

### 8.1. 1 The sinusoid.

The digital sinusoid, which is the building block for all forms of computationally based spectral analysis can be derived from the height of a point above a line as a function of time as it moves in discrete jumps, or steps, at a constant speed around a circle. In deriving a digital sinusoid, there are four variables to consider: the **amplitude**, **frequency**, **phase**, and the **number of points per repetition or cycle**. Examples of digital sinusoids are shown in Fig. 8.1 and were produced with the Emu-R crplot() function as follows:

```
par(mfrow=c(3,2)); par(mar=c(1, 2, 1, 1))
oldpar = par(no.readonly=TRUE)

# Plot a circle and its sinusoid with defaults: A = 1, k = 1, p = 0, N = 16
crplot()
# Set the amplitude to 0.75
crplot(A=.75)
# Set the frequency to 3 cycles
crplot(k=3)
# Set the phase to –π/2 radians.
crplot(p=-pi/2)
# Set the frequency to 3 cycles and with N = 24 points
crplot(k=3, N=24)
# Set the frequency to 13 cycles (and the default of N = 16 points)
crplot(k=13)
par(oldpar)
```

In the default case in Fig. 8.1(a), the point hops around the circle at a constant speed at intervals that are equally spaced on the circumference starting at the top of the circle. A plot of the height of these points about the horizontal line results in the digital sinusoid known as a **cosine wave**. The cosine wave has an amplitude that is equal to the circle's radius and it has a frequency of $k = 1$ cycle, because the point jumps around the circle once for the 16 points that are available. In Fig. 1(b), the same cosine wave is derived but with a reduced amplitude, corresponding to a decrease in the circle's radius – but notice that all the other variables (frequency, phase, number of points) are the same.

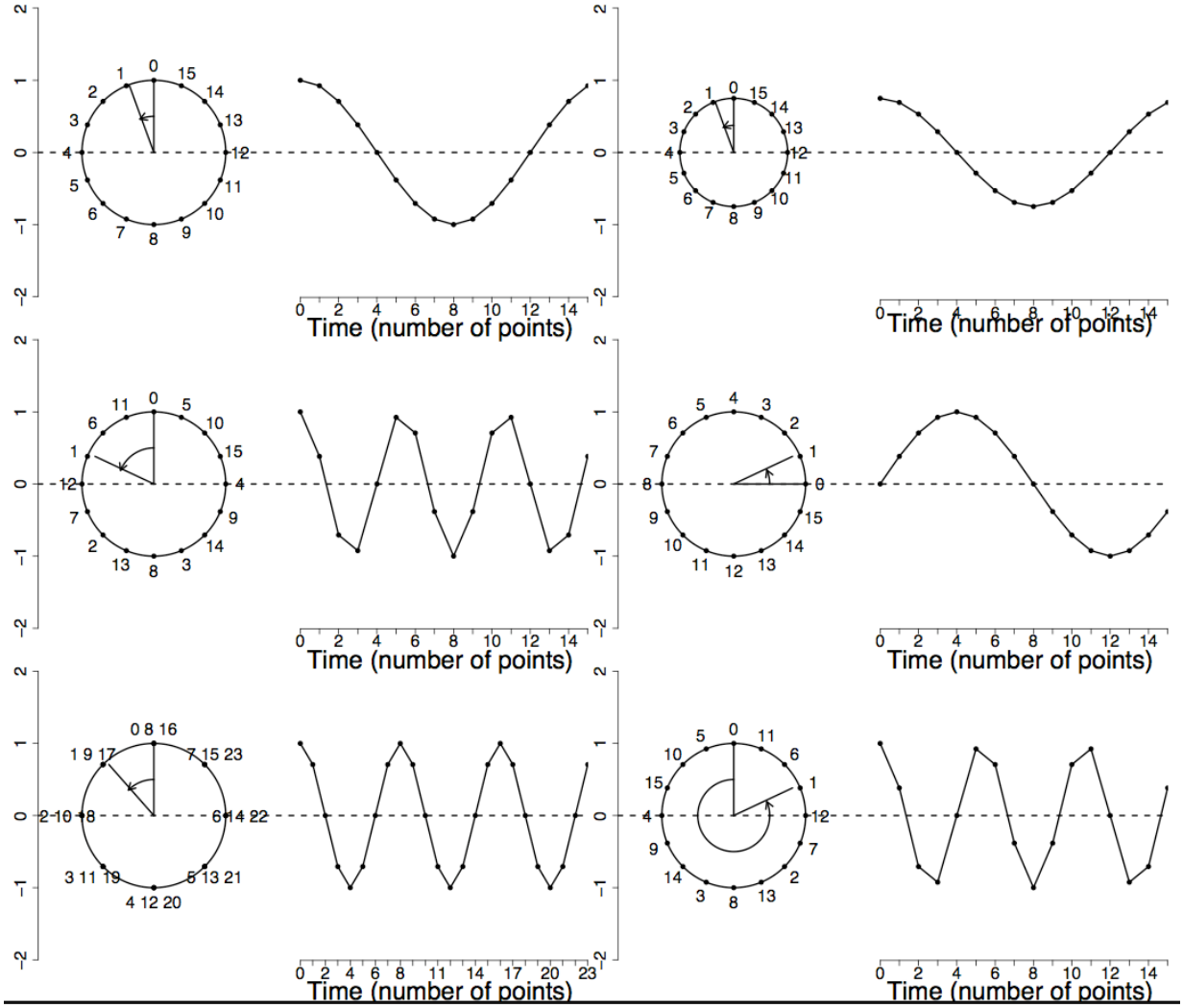



Fig. 8.1: Digital sinusoids and the corresponding circles from which they were derived. The numbers correspond to the position of the point either on the circle or along the corresponding sinusoid at time point *n*. Top left: a 16-point digital cosine wave. Top right: as top left, but in which the amplitude is reduced. Middle row, left: a three-cycle 16-point cosine wave. Middle row, right: a 16-point digital sine wave. Bottom left: the same as middle row left except with 24 digital points. Bottom right: A 13-cycle, 16-point cosine wave that necessarily aliases onto a 3-cycle cosine wave.

In Fig. 8.1(c) the variables are the same as in Fig. 8.1(a) except that the frequency has been changed to $k = 3$ cycles. Here, then, the point hops around the circle three times **given the same number of digital points** – and so necessarily the angle at the centre of the circle that is made between successive hops is three times as large compared with those of 8.1a and 8.1b for which $k = 1$. In Fig. 8.1(d), the variable that is changed relative to Fig. 8.1(a) is the **phase**, which is measured in radians. If the point begins at the top of the circle as in Figs. 8.1a-c, then the phase is defined to be zero radians. The phase can vary between ± π radians: at -π/2 radians, as in Fig. 8.1(d), the point starts a quarter of a cycle earlier to produce what is called a **sine wave**; if the phase is π/4 radians (argument p = pi/4), then the point starts at 1/8 cycle later, and so on.

The number of digital points that are available per cycle can be varied independently of the other parameters. Fig. 8.1(e) is the same as the one in Fig. 8.1(c)

except that the point hops around the circle three times with 24 points (8 more than in 8.1(c). The resulting sinusoid that is produced – a three cycle cosine wave – is the same as the one in Fig. 8.1(c) except that it is supported by a greater number of digital points (and for this reason it is smoother and looks a bit more like an analog cosine wave). Since there are more points available, then the angle made at the centre of circle is necessarily smaller (compare the angles between points in Fig. 8.1(c) and Fig. 8.1(e)). Finally, Fig. 8.1(f) illustrates the property of **aliasing** that comes about when *k*, the number of cycles, exceeds half the number of available data points. This is so in Fig. 8.1(f) for which k = 13 and N = 16. In this case, the point has to hop 13 times around the circle **but in 16 discrete jumps**. Since the number of cycles is so large relative to the number of available data points, the angle between the hops at the centre of the circle is very large. Thus in making the first hop between time points 0 and 1, the point has to move almost all the way round the circle in order to achieve the required 13 revolutions with only 16 points. Indeed, the jump is so big that the point always ends up on the opposite side of the circle compared with the sinusoid at $k = 3$ cycles in Fig. 8.1(c). For this reason, when k = 13 and N = 16, the result is not a 13-cycle sinusoid at all, but the same three-cycle sinusoid shown in Fig. 8.1(c).

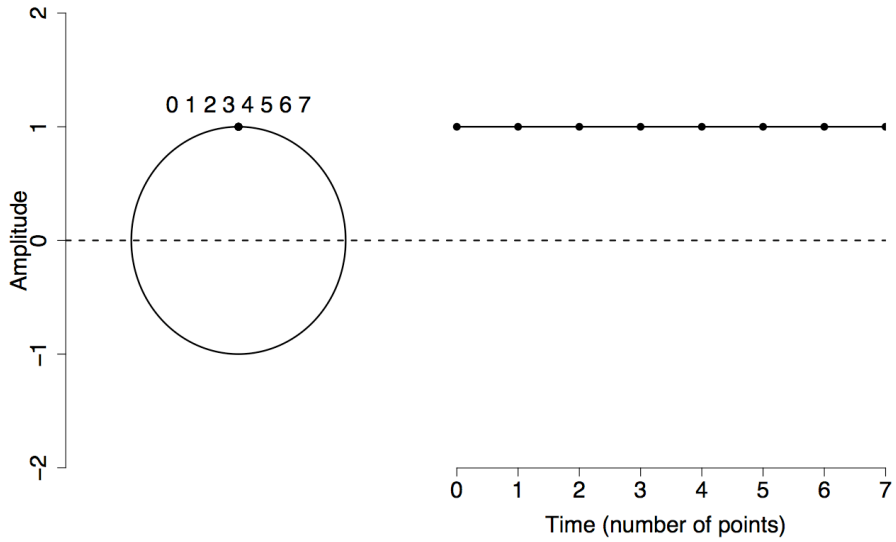



Fig. 8.2: An 8-point sinusoid with frequency $k = 0$ cycles.

It is an axiom of Nyquist's theorem (1822) that it is never possible to get frequencies of more than $N/2$ cycles from $N$ points. So for the present example with $N = 16$, the sinusoid with the highest frequency has 8 cycles (try crplot(k=8) ), but for higher $k$, all the others resulting sinusoids are aliases of those at lower frequencies (specifically the frequencies $k$ and $N-k$ cycles produce exactly the same sinusoids if the phase is the same: thus crplot(k=15) and crplot(k=1) both result in the same 1-cycle sinusoid). The highest frequency up to which the sinusoids are unique is $k = N/2$ and this is known as the **critical Nyquist** or folding frequency. Beyond $k = N/2$, the sinusoids are aliases of those at lower frequencies.

Finally, we will sometimes come across a sinusoid with a frequency of $k = 0$. What could this look like? Since the frequency is zero, the point does not complete any cycles and so has to stay where it is: that is, for $N = 8$ points, it hops up and down 8

times on the same spot. The result must therefore be a straight line, as Fig. 8.2 (given by crplot(k=0, N=8) ) shows.

### 8.1.2 Fourier analysis and Fourier synthesis

**Fourier analysis** is a technique that decomposes any signal into a set of sinusoids which, when summed reconstruct exactly the original signal. This summation or reconstruction of the signal from the sinusoids into which it was decomposed is called **Fourier synthesis**.

When Fourier analysis is applied to a digital signal of length $N$ data points, then a calculation is made of the amplitudes and phases of the sinusoids at integer frequency intervals from 0 to $N$-1 cycles. So for a signal of length 8 data points, the Fourier analysis calculates the amplitude and phase of the $k = 0$ sinusoid, the amplitude and phase of the $k = 1$ sinusoid, the amplitude and phase of the $k = 2$ sinusoid… the amplitude and phase of the $k = N\text{-}1 = 7$ sinusoid. Moreover it does so in such a way that if all these sinusoids are added up, then the original signal is reconstructed.

In order to get some understanding of this operation (and without going into mathematical details which would lead too far into discussions of exponentials and complex numbers), a signal consisting of 8 random numbers will be decomposed into a set of sinusoids; the original signal will then be reconstructed by summing them. Here are 8 random numbers between -10 and 10:

```
r = runif(8, -10, 10)
```

The principal operation for carrying out the Fourier analysis on a computer is the **discrete Fourier transform** (DFT) of which there is a faster version known as the **Fast Fourier Transform** (the FFT). The FFT can only be used if the window length – that is the number of data points that are subjected to Fourier analysis – is exactly a power of 2. In all other cases, the DFT has to be used. Since in the special case when the FFT can be used it gives exactly equivalent results to a DFT, the mathematical operation for carrying out Fourier analysis on a time signal will be referred to as the DFT, while recognising in practice that, for the sake of speed, a window length will usually be chosen that is a power of 2, thereby allowing the faster version of the DFT, the FFT, to be implemented. In R, the function for carrying out a DFT (FFT) is the function fft(). Thus the DFT of the above signal is:

```
r.f = fft(r)
```

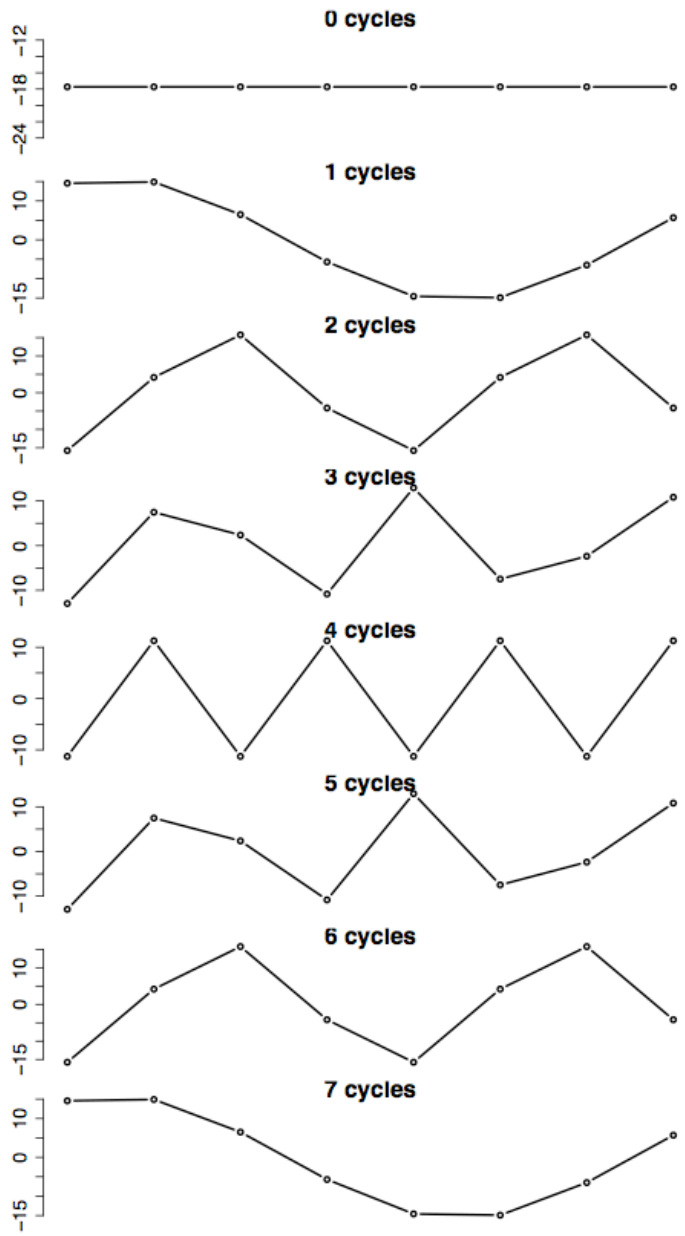



Fig. 8.3: The digital sinusoids into which a sequence of 8 random numbers was decomposed with a DFT.

`r.f` is a vector of 8 complex numbers involving the square root of minus 1. Each such complex number can be thought of as a convenient code that embodies the amplitude and phase values of the sinusoids from $k = 0$ to $k = 7$ cycles. To look at the sinusoids that were derived by Fourier analysis, their amplitude and phase values must be derived (the frequencies are already known: these are $k = 0, 1, ..7$). The amplitude is obtained by **taking the modulus**, and the phase by **taking the argument** of the complex number representations. In R, these two operations are carried out with the functions Mod() and Arg() respectively:

```
r.a = Mod(r.f)  # Amplitudes
```

```
r.p = Arg(r.f)   # Phases
```

So a plot can now be produced of the sinusoids into which the 8 random numbers were decomposed using the cr() function for plotting sinusoids: (Fig. 8.3):

```
par(mfrow=c(8,1)); par(mar=c(1,2,1,1))
for(j in 1:8){
cr(r.a[j], j-1, r.p[j], 8, main=paste(j-1, "cycles"), axes=F)
axis(side=2)
}
```

Following the earlier discussion, the amplitude of the sinusoid with frequency $k = 0$ cycles is a straight line, and the sinusoids at frequencies $N$-$k$ are aliases of those at frequencies of $k$ (so the sinusoids for the pairs $k = 1$ and $k = 7$; $k = 2$ and $k = 6$; $k = 3$ and $k = 5$ are the same). Also, the amplitude of the $k = 0$ sinusoid[1] is equal to the sum of the values of the waveform (compare sum(r) and r.a[1]).

Carrying out Fourier synthesis involves summing the sinusoids in Fig. 8.3, an operation which should exactly reconstruct the original random number signal. However, since the results of a DFT are inflated by the length of the window – that is by the length of the signal to which Fourier analysis was applied - what we get back is $N$ times the values of the original signal. So to reconstruct the original signal, the length of window has to be normalised by dividing by $N$ (by 8 in this case). The summation of the sinusoids can also be done with the same cr() function:

```
# Sum the sinuoids and divide by N
summed = cr(r.a, 0:7, r.p, 8, values=T)/8
```

Rounding errors apart, summed and the original signal r are the same, as can be verified by subtracting one from the other. However, the usual way of doing Fourier synthesis in digital signal processing is to carry out an **inverse Fourier transform** on the complex numbers and in R this is also done with the fft() function by including the argument inverse=T. The result is something that looks like a vector of complex numbers, but in fact the so-called imaginary part, involving the square root of minus one, is in all cases zero. To get the vector in a form without the +0i, take the real part using the Re() function:

```
# Inverse Fourier transform and normalise for the length of the signal
summed2 = Re(fft(r.f, inverse=T))/8
```

There is thus equivalence between the original waveform of 8 random numbers, r, summed2 obtained with an inverse Fourier transform, and summed obtained by summing the sinusoids.

### 8.1.3 Amplitude spectrum

An amplitude spectrum, or just spectrum, is a plot of the sinusoids' amplitudes values as a function of frequency. A phase spectrum is a plot of their phase values as a function of frequency. (This Chapter will only be concerned with amplitude spectra, since phase spectra do not contribute a great deal, if anything, to the distinction

---

[1] This sinusoid at 0 Hz is equal to what is sometimes called the DC offset divided in this case by N, the length of the signal (i.e., the DC offset is the mean of the signal's amplitude).

between most phonetic categories). Since the information above $k = N/2$ is, as discussed in 8.1.1, replicated in the bottom part of the spectrum, it is usually discarded. Here then is the (amplitude) spectrum for these 8 random numbers up to and including $k = N/2$ cycles (normalised for the length of the window). In the commands below, the Emu-R function as.spectral() does nothing more than to make the objects of class 'spectral' so that various functions like plot() can handle them appropriately[2].

```
# Discard amplitudes above k = N/2
r.a = r.a[1:5]
# Normalise for the length of the window and make r.a into a spectral object:
r.a = as.spectral(r.a/8)
# Amplitude spectrum
plot(r.a, xlab="Frequency (number of cycles)", ylab="Amplitude", type="b")
```

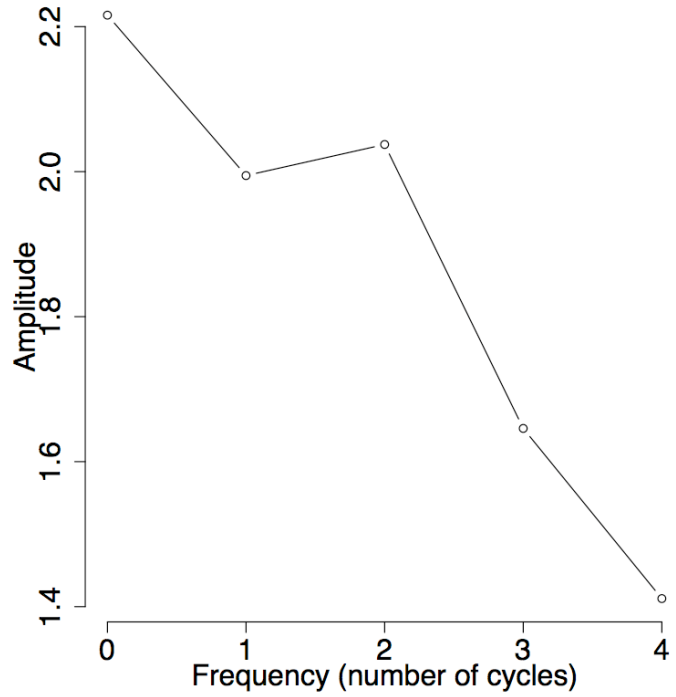



Fig. 8.4: An amplitude spectrum of an 8-point signal up to the critical Nyquist frequency. These spectral values are sometimes referred to as the unreflected part of the spectrum.

### 8.1.4 Sampling frequency

So far, frequency has been represented in integer numbers of cycles which, as discussed in 8.1.1 refers to the number of revolutions made around the circle per $N$ points. But usually the frequency axis of the spectrum is in cycles per second or Hertz and the Hertz values depend on the sampling frequency. If the digital signal to which the Fourier transform was applied has a sampling frequency of $f_s$ Hz and is of length $N$ data points, then the frequency in Hz corresponding to $k$ cycles is $f_s\, k/N$. So for a

[2] Specifically, when an object of class 'spectral' is called with `plot()`, then, as a result of an object oriented programming implementation of spectral data in Emu-R, it is actually called with `plot.spectral()`).

sampling frequency of 10000 Hz and $N = 8$ points, the spectrum up to the critical Nyquist has amplitudes at frequency components in Hz of:

```
10000 * 0:5/8
0   1250   2500   3750   5000
```

From another point of view, the spectrum of an $N$-point digital signal sampled at $f_s$ Hz consists of $N/2 + 1$ spectral components (has $N/2 + 1$ amplitude values) extending between 0 Hz and $f_s/2$ Hz with a frequency interval between them of $f_s / N$ Hz.

**8.1.5 dB-Spectrum**

It is usual in obtaining a spectrum of a speech signal to use the decibel scale for representing the amplitude values. A decibel is 10 times one Bel and a Bel is defined as the logarithm of the ratio of the acoustic intensity of a sound ($I_S$) to the acoustic intensity of a reference sound ($I_R$). Leaving aside the meaning of 'reference sound' for the moment, the intensity of a sound is given by:

(1) $I_S = 10 \log_{10}(I_S/I_R)$ dB

The acoustic or sound intensity $I_S$ is a physical measurable quantity with units Watts per square metre. Thus to take an example, if $I_S = 10^{-8}$ watts/m$^2$ and $I_R = 10^{-4}$ watts/m$^2$, then $I_S$ is 40 dB relative to $I_R$ (40 decibels more intense than $I_R$).

More importantly for the present discussion, there is a relationship between acoustic intensity and the more familiar amplitude of air-pressure which is recorded with a pressure-sensitive microphone. The relationship is $I_S = kA^2$, where $k$ is a constant. Thus (1) can be rewritten as:

(2) $I_S = 10 \log_{10} (kA_S^2 / kA_R^2)$ dB $= 10 \log_{10}(A_S/A_R)^2$ dB

and since $\log a^b = b \log a$ and since $\log(a/b) = \log a - \log b$, (3) is the same as (2):

(3) $I_S = 20 \log_{10}A_S - 20 \log_{10}A_R$ dB

The Emu-tkassp system that is used to compute dB-spectra referred to throughout this book assumes that the amplitude of the reference sound, $A_R$ (and hence $20 \log_{10}A_R$), is zero. So, finally, the decibel values that you will see in the spectra in this Chapter and indeed in this book are $20 \log_{10}A_S$ where $A_S$ refers to the amplitudes of sinusoids produced in Fourier analysis. Now since log(1) is always equal to zero, then this effectively means that the reference sound is a signal with an amplitude of ± 1 unit: a signal with any sequence of +1 or -1 like 1, 1, -1, 1… etc.

There are two main reasons for using decibels. Firstly, the decibel scale is more closely related to perceived loudness than amplitudes of air-pressure variation. Secondly, the range of amplitude of air-pressure variations within the hearing range is considerable and, because of the logarithmic conversion, the decibel scale represents this range in more manageable values (from e.g. 0 to 120 dB).

An important point to remember is that the decibel scale always expresses a **ratio of power**s i.e., in a decibel calculation, the intensity of a signal is always defined relative to a reference sound. If the latter is taken to be the threshold of hearing, then the units are said to be dB SPL where SPL stands for sound pressure level (and 0 dB is

defined as the threshold of hearing). Independently of what we take to be the reference sound, it is useful to remember that 3 dB represents an approximate doubling of power and whenever the power is increased ten fold, then there is an increase of 10 dB (the power is the square of the amplitude of air-pressure variations).

Another helpful way of thinking of decibels is as percentages, because this is an important reminder that a decibel, like a percentage, is not an actual quantity (like a meter or second) but a ratio. For example, one decibel represents a power increase of 27%, 3 dB represents a power increase of 100% (i.e., a doubling of power), and 10 dB represents a power increase of 1000% (a tenfold increase of power).

### 8.1.6 Hamming and Hann(ing) windows

If a DFT is applied to a signal that is made up exactly of an integer number of sinusoidal cycles, then the only frequencies that show up in the spectrum are the sinusoids' frequencies. For example, row 1 column 1 of Fig. 8.5 shows a sinusoid of *exactly* 20 cycles i.e. there is one cycle per 5 points. The result of making a spectrum of this 20-cycle waveform is a single component at the same frequency (Fig. 8.5, row 1, right). Its spectrum was calculated with the spect() function below which simply packages up some of the commands that have been discussed in the previous section.

```
spect <- function(signal, ...)
{
# The number of points in this signal
N <- length(signal)
# Apply FFT and take the modulus
signal.f <- fft(signal);  signal.a <- Mod(signal.f)
# Discard magnitudes above the critical Nyquist and normalise
signal.a <- signal.a[1:(N/2+1)]/N
as.spectral(signal.a, ...)
}
```

The 20 cycle sinusoid (Fig. 8.5, row 1) and its spectrum are then given by:

```
par(mfrow=c(3,2))
a20 = cr(k=20, N=100, values=T, type="l", xlab="")
# Spectrum thereof
plot(spect(a20), type="h", xlab="", ylab="")
```

But the sinusoid's frequency only shows up faithfully in the spectrum in this way if the window to which the DFT is applied is made up of an **exact** number of whole sinusoidal cycles. When this is not the case, then a phenomenon called **spectral leakage** occurs, in which magnitudes show up in the spectrum, other than at the sinusoidal frequency. This is illustrated in the lower panel of the same figure for a sinusoid whose frequency is fractional at 20.5 cycles.

```
# Fig. 8.5 middle panel
a20.5 = cr(k=20.5, N=100, values=T, type="l", xlab="")
plot(spect(a20.5), type="h", xlab="", ylab="")
```

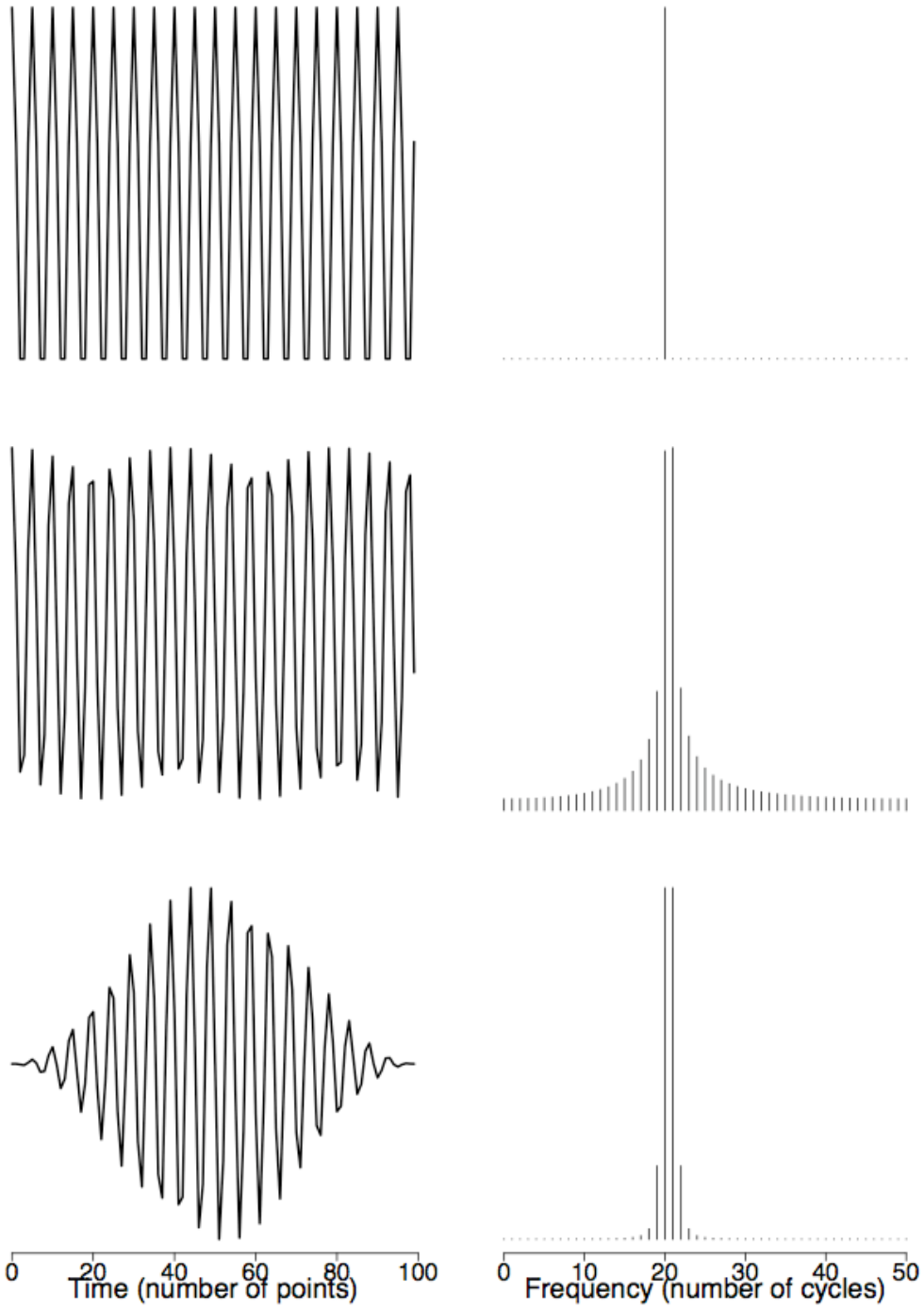



Fig. 8.5: Waveforms of sinusoids (left column) and their corresponding amplitude spectra (right column). Row 1: a 20-cycle sinusoid. Row 2: a 20.5 cycle sinusoid. Row 3: As row 2 but after the application of a Hanning window.

Because a DFT can only compute amplitudes at integer values of $k$, the sinusoid with a frequency of 20.5 cycles falls, as it were, between the cracks: the spectrum does have a peak around $k = 20$ and $k = 21$ cycles, but there are also amplitudes at other frequencies. Obviously something is not right: a frequency of $k = 20.5$ should somehow show up in the spectrum at only that frequency and not as energy that is distributed throughout the spectrum.

What is actually happening for the $k = 20.5$ frequency is that the spectrum is being convolved with the spectrum of what is called a **rectangular window**. Why should this be so? Firstly, we need to be clear about an important characteristic of digital signals: they are finite, i.e. they have an abrupt beginning and an abrupt ending. So although the sinusoids in the first two rows of Fig. 8.5 look as if they might carry on forever, in fact they do not and, most importantly, as far as the DFT is concerned, they are zero-valued outside the windows that are shown: that is, the signals in Fig. 8.5 start abruptly at $n = 0$ and they end abruptly at $n = 99$. In digital signal processing, this is equivalent to saying that a sinusoid whose duration is infinite is multiplied with a rectangular window whose values are 1 over the extent of the signals that can be seen in Fig. 8.5 (thereby leaving the values between $n = 0$ and $n = 99$ untouched, since multiplication with 1 gives back the same result) and multiplied by zero everywhere else. This means that any finite-duration signal that is spectrally analysed is effectively multiplied by just such a rectangular window. But this also means that when a DFT of a finite-duration signal is calculated, a DFT is effectively *also* being calculated of the rectangular window and it is the abrupt switch from 0 to 1 and back to 0 that causes extra magnitudes to appear in the spectrum as in the middle right panel of Fig. 8.8. Now it so happens that when the signal consists of an integer number of sinusoidal cycles so that an exact number of cycles fits into the window as in the top row of Fig. 8.5, then the rectangular window has no effect on the spectrum: therefore, the sinusoidal frequencies show up faithfully in the spectrum (top right panel of Fig. 8.5). But in all other cases – and this will certainly be so for any speech signal – the spectrum is also influenced by the rectangular window.

Although such effects of a rectangular window cannot be *completely* eliminated, there is a way to *reduce* them. One way to do this is to make the transitions at the edges of the signal gradual, rather than abrupt: in this case, there will no longer be abrupt jumps between 0 and 1 and so the spectral influence due to the rectangular window should be diminished. A common way of achieving this is to multiply the signal with some form of cosine function and two such commonly used functions in speech analysis are the Hamming window and the Hanning window (the latter is a misnomer, in fact, since the window is due to von Hann). An $N$-point Hamming or Hanning window can be produced with the following function:

```
w <- function(a=0.5, b=0.5, N=512)
{
n =  0: (N-1)
a - b * cos(2 * pi * n/(N-1) )
}
```

With no arguments, the above w() function defaults to a 512-point Hanning window that can be plotted with plot(w()); for a Hamming window, set a and b to 0.54 and 0.46 respectively.

The bottom left panel of Fig. 8.5 shows the 20.5 cycle sinusoid of the middle left panel multiplied with a Hanning window. The spectrum of the resulting Hanning-

windowed signal is shown in the bottom right of the same figure. The commands to produce these plots are as follows:

```
# Lower panel of Fig. 8.5. Multiply a20.5 with a Hanning window of the same length
N = length(a20.5)
a20.5h = a20.5 * w(N=N)
# The Hanning-windowed 20.5 cycle sinusoid
plot(0:(N-1), a20.5h,  type="l", xlab="Time (number of points)")
# Calculate its spectral values
plot(a20.5h.f, type="h", xlab="Frequency (number of cycles)", ylab="")
```

A comparison of the middle and bottom right panels of Fig. 8.5 shows that, although the spectral leakage has not been eliminated, multiplication with the Hanning window has certainly caused it to be reduced.

### 8.1.7 Time and frequency resolution

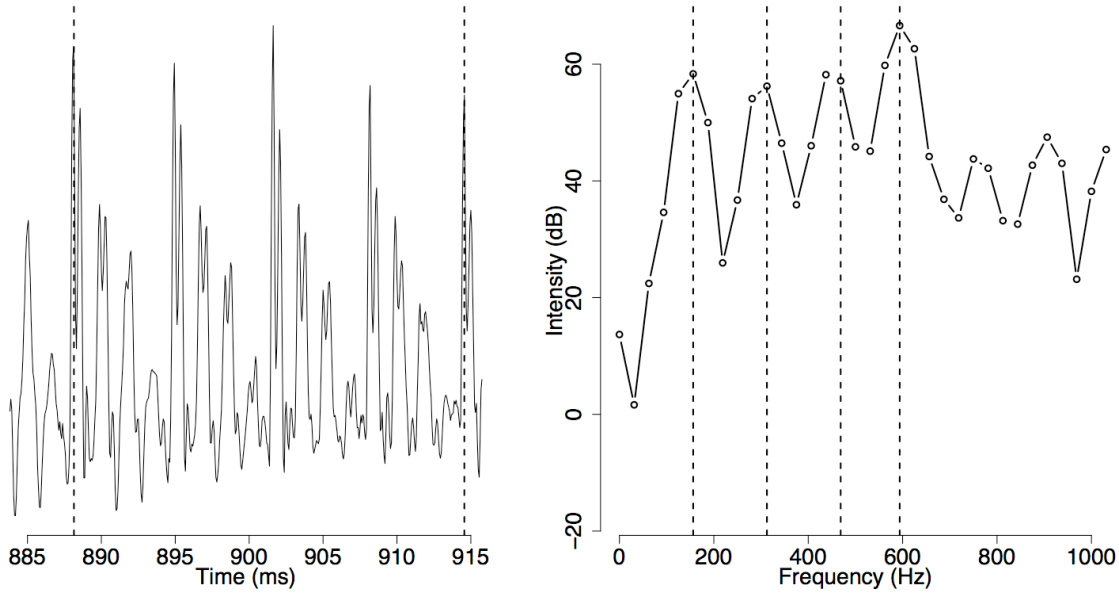



Fig. 8.6: Left: A 512-point waveform of a German [ɛ] produced by a male speaker. The dashed vertical lines mark out 4 pitch periods. Right: A spectrum of this 512-point signal. The vertical dashed lines mark the expected frequency location of f0, the 2nd, 3rd, and 4th harmonics.

One of the fundamental properties of any kind of spectral analysis is that time resolution and frequency resolution are inversely proportional. 'Resolution' in this context defines the extent to which two events in time or two components in frequency are distinguishable. Because time and frequency resolution are inversely proportional, then if the time resolution is high (i.e., two events that occur very close together in time are distinguishable), then the frequency resolution is low (i.e., two components that are close together in frequency are indistinguishable) and *vice-versa*.

In computing a DFT, the frequency resolution is the interval between the spectral components and, as discussed in 8.1.2, this depends on $N$, the length of the signal or window to which the DFT is applied. Thus, when $N$ is large (i.e., the DFT is applied to a signal long in duration), then the frequency resolution is high, and when $N$ is low, the frequency resolution is coarse. More specifically, when a DFT is applied to an $N$-point signal, then the frequency resolution is $f_s/N$, where $f_s$ is the sampling frequency. So for a sampling frequency of $f_s$ = 16000 Hz and a signal of length $N$ = 512 points (= 32 ms at 16000 Hz), the frequency resolution is $f_s/N$ = 16000/512 = 31.25 Hz. Thus magnitudes will show up in the spectrum at 0 Hz, 31.25 Hz, 62.50 Hz, 93.75 Hz… up to 8000 Hz **and at only these frequencies**, i.e. all other frequencies are zero or undefined. If, on the other hand, the DFT is applied to a much shorter signal at the same sampling frequency, e.g., to a signal of $N$ = 64 points (4 ms), then the frequency resolution is much coarser at $f_s/N$ = 250 Hz and this means that there can only be spectral components at intervals of 250 Hz, i.e. at 0 Hz, 250 Hz, 500 Hz… 8000 Hz.

These principles can be further illustrated by computing DFTs for different values of $N$ over the signal shown in Fig. 8.6. This is 512 sampled speech data points extracted from the first segment of segment list vowlax of German lax monophthongs. The sampled speech data is stored in the trackdata object v.sam (if you have downloaded the `kielread` database, then you can (re)create this object from v.sam = emu.track(vowlax[1,], "samples", 0.5, 512) ). The data in Fig. 8.6 was plotted as follows:

```
plot(v.sam[1,], type="l", xlab="Time (ms)", ylab="Amplitude")
# Mark vertical lines spanning 4 pitch periods (click the mouse twice,
# once at each of time point shown in the figure
v = locator(2)$x
abline(v=v, lty=2)
# Interval (ms) between these lines
diff(v)
[1] 26.37400
```

The figure shows that four pitch periods have a duration of 26.42 ms, so the fundamental frequency over this interval is 4000/26.42 ≈ 151 Hz. If a DFT is applied to this signal, then the fundamental frequency and its associated harmonics should show up at multiples of this fundamental frequency in the resulting spectrum. In the following, the spect() function has been updated to include the various commands discussed earlier for calculating a dB-spectrum and for applying a Hanning window:

```
spect <- function(signal,..., hanning=T, dbspec=T)
{
# The Hanning window function
w <- function(a=0.5, b=0.5, N=512)
{
n =  0: (N-1)
a - b * cos(2 * pi * n/(N-1) )
}
# The number of points in this signal
N <- length(signal)
# Apply a Hanning window
if(hanning)
```

```
signal <- signal * w(N=N)
# Apply FFT and take the modulus
signal.f <- fft(signal);  signal.a <- Mod(signal.f)
# Discard magnitudes above the critical Nyquist and normalise for N
signal.a <- signal.a[1:(N/2+1)]/N
# Convert to dB
if(dbspec)
signal.a <- 20 * log(signal.a, base=10)
as.spectral(signal.a, ...)
}
```

Here is the dB-spectrum (right panel, Fig. 8.6):

```
# Store the sampled speech data of the 1st segment in a vector for convenience
sam512 = v.sam[1,]$data
# dB-spectrum. The sampling frequency of the signal is the 2nd argument
sam512.db = spect(sam512, 16000)
# Plot the log-magnitude spectrum up to 1000 Hz
plot(sam512.db, type="b", xlim=c(0, 1000), xlab="Frequency (Hz)", ylab="Intensity
(dB)")
```

Consider how the harmonics show up in the spectrum. Firstly, since f0 was estimated to be 151 Hz from the waveform in Fig.8.6, then f0 and harmonics 2-5 are to be expected at the following frequencies:

```
seq(151, by=151, length=5)
151  302  453  604  755
```

However, for a sampling frequency of 16000 Hz and a window length of *N*=512, the first 20 spectral magnitudes are computed only at *these* frequencies:

```
seq(0, by=16000/512, length=20)
0.00  31.25  62.50  93.75 128.00 156.25 187.50 218.75 250.00 281.25
312.50 343.75 378.00 406.25 437.50 468.75 500.00 531.25 562.50 593.75
```

Thus, there can be no peaks in the spectrum due to f0 and the harmonics at **exactly** their frequencies (because there is no frequency in the digital spectrum to support them), but instead at whichever spectral component they come closest to. These are shown in bold above, and if you count where they are in the vector, you will see that these are the 6th, 11th, 16th, and 20th spectral components. Vertical lines have been superimposed on the spectrum at these frequencies as follows:

```
abline(v=seq(0, by=16000/512, length=20) [c(6, 11, 16, 20)], lty=2)
```

and indeed the spectrum does show peaks at these frequencies. Notice that the relationship between the expected peak location (vertical lines) and actual peaks is least accurate for the 3rd vertical line. This is because there is quite a large deviation between the estimated frequency of the 3rd harmonic (453 Hz) and the (16th) spectral component that is closest to it in frequency (468.75 Hz). (From another point of view: the estimated 3rd harmonic falls almost exactly halfway in frequency between two spectral components).

The harmonics can only make their presence felt in a spectrum of voiced speech as long as the signal over which the DFT is applied includes at least two pitch periods. From a related point of view, harmonics can only be seen in the spectrum if the frequency resolution (interval between frequency components in the spectrum) is quite a lot less than the fundamental frequency. Consider, then, the effect of applying a DFT to just the first 64 points of the same signal. A 64-point DFT at 16000 Hz results in a frequency resolution of 16000/64 = 250 Hz, so there will be 33 spectral components between 0 Hz and 8000 Hz at intervals of 250 Hz, i.e., at a frequency interval that is greater than the fundamental frequency, and therefore too wide for the frequency and separate harmonics to have much of an effect on the spectrum. The corresponding spectrum up to 3500 Hz in Fig. 8.7 was produced as follows:

```
sam64 = sam512[1:64]
ylim = c(10, 70)
plot(spect(sam64, 16000), type = "b", xlim=c(0, 3500), ylim=ylim, xlab="Frequency
(Hz)", ylab="Intensity (dB)")
```

The influence of the fundamental frequency and harmonics is no longer visible, but instead the formant structure emerges more clearly. The first four formants, calculated with the Emu LPC-based formant tracker at roughly the same time point for which the spectrum was calculated, are:

```
dcut(vowlax.fdat[1,], 0.5, prop=T)
T1   T2   T3   T4
562 1768 2379 3399
```

The frequencies of the first  two of these have been superimposed on the spectrum in Fig. 8.7 (left) as vertical lines with `abline(v=c(562, 1768))` .

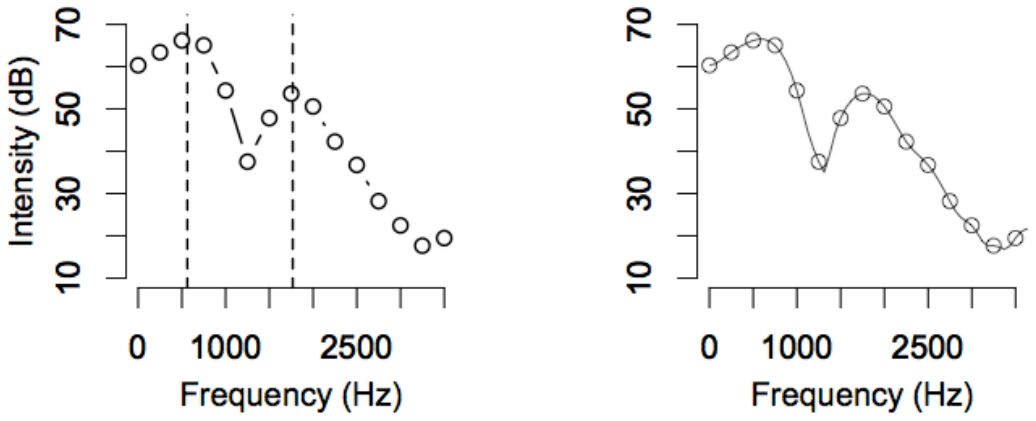



Fig. 8.7: A spectrum of the first 64 points of the waveform in Fig. 8.7 (left) and of the first 64 points and padded out with 192 zeros (right).

As discussed earlier, there are no magnitudes other than those that occur at $f_s/N$ and so the  spectrum in Fig. 8.7 (left)  is zero-valued except at frequencies 0, 250 Hz, 500 Hz…8000 Hz. It is possible however to interpolate between these spectral components are thereby make the spectrum smoother using a technique called **zero**

**padding**. In this technique, a number of zero-valued data samples are appended to the signal to increase its length to, for example, 256 points. (NB: these zero data values should be appended *after* the signal has been Hamming- or Hanning-windowed, otherwise the zeros will distort the resulting spectrum). One of the main practical applications of zero-padding is to fill up the number of data points of a window to a power of 2 so that the FFT algorithm can be applied to the signal data. Suppose that a user has the option of specifying the window length of a signal that is to be Fourier analysed in milliseconds rather than points and suppose a user chooses an analysis window of 3 ms. At a sampling frequency of 16000 Hz, a duration of 3 ms is 48 sampled data points. But if the program makes use of the FFT algorithm, there will be a problem because, as discussed earlier, the FFT requires the window length, *N*, to be an integer power of 2, and 48 is not a power 2. One option then would be to append 24 zero data sample values to bring the window length up to the next power of 2, i.e., to 64.

The command lines to do the zero-padding have been incorporated into the evolving spect() function – in particular, there is an extra argument fftlength which can be greater than the signal length and if so, the signal is appended (or prepended – it doesn't make any difference) with a number of zeros equal to the difference between fftlength and the actual length of the signal, thus:

```
"spect" <-
function(signal,...,  hanning=T, dbspec=T, fftlength=NULL)
{
w <- function(a=0.5, b=0.5, N=512)
{
n =  0: (N-1)
a - b * cos(2 * pi * n/(N-1) )
}
# The number of points in this signal
N <- length(signal)
# Apply a Hanning window
if(hanning)
signal <- signal * w(N=N)
# If fftlength is specified…
if(!is.null(fftlength))
{
# and if fftlength is longer than the length of the signal…
if(fftlength > N)
# then pad out the signal with zeros
signal <- c(signal, rep(0, fftlength-N))
}
# Apply FFT and take the modulus
signal.f <- fft(signal);  signal.a <- Mod(signal.f)
# Discard magnitudes above the critical Nyquist and normalise
if(is.null(fftlength))
signal.a <- signal.a[1:(N/2+1)]/N
else
signal.a <- signal.a[1:(fftlength/2+1)]/N
# Convert to dB
if(dbspec)
```

```
signal.a <- 20 * log(signal.a, base=10)
as.spectral(signal.a, ...)
}
```

In the following, the same 64-point signal is zero-padded to bring it up to a signal length of 256 (i.e., it is appended with 192 zeros). The spectrum of this zero-padded signal is shown as the continuous line in Fig. 8.7 (right) superimposed on the 64-point spectrum:

```
ylim = c(10, 70)
plot(spect(sam64, 16000), type = "p", xlim=c(0, 3500), xlab="Frequency (Hz)",
ylim=ylim)
par(new=T)
plot(spect(sam64, 16000, fftlength=256), type = "l", xlim=c(0, 3500), xlab="",
ylim=ylim)
```

As Fig. 8.7 (right) shows, a spectrum derived from zero padding passes through the same spectral components that were derived without it (thus, zero padding provides additional points of interpolation). An important point to remember is that zero padding does not improve the spectral resolution: zero-padding does not add any new data beyond providing a few extra smooth interpolation points between spectral components that really are in the signal, and sometimes, as Hamming (1989) argues, zero-padding can provide misleading information about the true spectral content of a signal.

### 8.1.8 Preemphasis

In certain kinds of acoustic analysis, the speech signal is pre-emphasised which means that the resulting spectrum has its energy levels boosted by somewhat under 6 dB per doubling of frequency, or (just less than) 6 dB/octave. This is sometimes done to give greater emphasis to the spectral content at higher frequencies, but also sometimes to mimic the roughly 6 dB/octave lift that is produced when the acoustic speech signal is transmitted beyond the lips. Irrespective of these motivations, pre-emphasising the signal can be useful in order to distinguish between two speech sounds that are otherwise mostly differentiated by energy at high frequencies. For example, the difference in the spectra of the release burst of a (syllable-initial) [t] and [d] can lie not just below 500 Hz if [d] really is produced with vocal fold vibration in the closure and release (and often in English or German it is not), but often in a high frequency range above 4000 Hz. In particular, the spectrum of [t] can be more intense in the upper frequency range because the greater pressure build-up behind the closure can cause a more abrupt change in the signal from near acoustic silence (during the closure) to noise (at the release) – and a more abrupt change in the signal always has an effect on the higher frequencies (for example a 1000 Hz sinusoid changes faster in time than a 1 Hz sinusoid and, of course, has a higher frequency). When I explained this [t]-[d] difference to my engineering colleagues during my time at CSTR Edinburgh in the 1980s, they suggested pre-emphasising the bursts because this could magnify even further the spectral difference in the high frequency range (and so lead to a better separation between these stops).

A 6 dB/octave boost in frequency can be brought about by **differencing** a speech signal where differencing has the meaning of subtracting a signal delayed by one data point from itself. You can delay a signal by *k* data points with the shift()

function in the Emu-R library which causes x[n] to become x[n+1] in the signal x, for example:

```
x = c(2, 4, 0, 8, 5, -4, 6)
shift(x, 1)
6  2  4  0  8  5 -4
```

The shifting is done *circularly* by default in this function which means that the last data point becomes the first resulting in a signal of the same length. Other than that, it is clear that the effect of delaying the signal by one point is that that x[2] has moved to the 3rd position, x[3] to the 4th position, and so on. To difference a signal, the signal is subtracted from a delayed version of itself, i.e. the differenced signal is x - shift(x, 1). A spectrum of this differenced signal can be shown to have just less than a 6 dB/octave lift relative to the original signal.

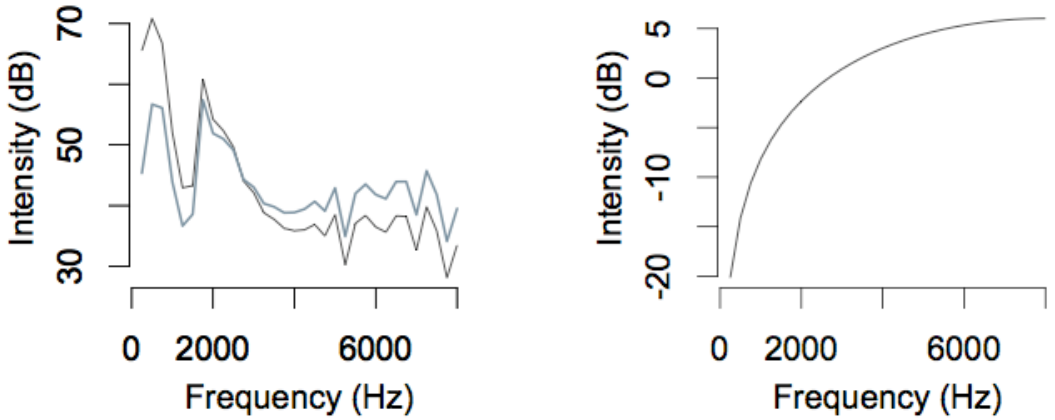



Fig. 8.8: Left: A spectrum of an [i] calculated without (black) and with (gray) first differencing. Right: the difference between the two spectra shown on the left.

This can be verified with the 64 point signal created in 8.1.7, sam64. To see the uncontaminated effect of the boost to higher frequencies, don't apply a Hanning window in the spect() function:

```
# dB-spectrum of  signal
sam64.db = spect(sam64, 16000, hanning=F)
# A one-point differenced signal
dnd = sam64 - shift(sam64, 1)
# The dB-spectrum of the differenced signal
dnd.db = spect(dnd, 16000, hanning=F)
# Superimpose the two spectra
ylim = range(c(sam64.db[-1], dnd.db[-1]))
par(mfrow=c(1,2))
plot(sam64.db[-1], type="l", ylim=ylim, xlab="Frequency (Hz)", ylab="Intensity
(dB)")
par(new=T)
plot(dnd.db[-1], type="l", ylim=ylim, col="slategray", lwd=2)
```

```
# Plot the difference between the spectra of the differenced and original signal
# excluding the DC offset
plot(dnd.db[-1] - sam64.db[-1], type="l", xlab="Frequency (Hz)", ylab="Intensity
(dB)")
```

The effect of differencing is indeed to increase the energy in frequencies in the upper part of the spectrum but also to decrease them in the lower spectral range, i.e. it is as if the entire spectrum is tilted upwards about the frequency axis. The extent of the dB-change can be seen by subtracting the spectrum of the differenced signal from the spectrum of the original signal and this has been done to obtain the spectrum on the right. The meaning of a change of roughly 6 dB/octave is as follows: take any two frequencies such that one is double the other and read off the dB-levels: the difference between these dB-levels will be near 6 dB. For example, the dB-levels at 1000, 2000 and 4000 Hz for the spectrum on the right are -8.17 dB, - 2.32 dB and 3.01 dB. So the change in dB from 1000 to 2000 Hz is very similar to the change from 2000 Hz to 4000 Hz (roughly 8.5 dB). You will notice that the dB-level towards 0 Hz gets smaller and smaller and is in fact minus infinity at 0 Hz (this is why the plot on the left excludes the DC offset at 0 Hz).

### 8.1.9 Handling spectral data in Emu-R

The usual way of getting spectral data into R is not with the functions like `spect()` of the previous section (which was created simply to show some of the different components that make up a dB-spectrum), but with the spectrum option in the Emu-tkassp toolkit (Chapter 3) for calculating spectral data allowing the DFT length, frame shift and other parameters to be set. These routines create signal files of spectral data that can be read into Emu-R with emu.track() which creates a **spectral trackdata object**.

The object plos.dft is such a spectral trackdata object and it contains spectral data between the acoustic closure onset and the onset of periodicity for German /b, d/ in syllable-initial position (see Appendix C for how these objects were created). The stops were produced by an adult male speaker of a Standard North German variety in a carrier sentence of the form *ich muss* /SVCn/ *sagen* ( *I must* /SVCn/ *say*) where S is the syllable-initial /b, d/ and /SVCn/ corresponds to words like /botn/ (*boten*, *bade*), /degn/ (*Degen*, *dagger*) etc. The speech data was sampled at 16 kHz so the spectra extend up to 8000 Hz. The DFT window length was 256 points and spectra were calculated every 5 ms:

| | |
|---|---|
| plos | Segment list /b, d/ from closure onset to the periodic onset of the following vowel |
| plos.l | A vector of labels (b, d) |
| plos.w | A vector of the word labels from which the segments were taken |
| plos.lv | A vector of labels of the following vowels. |
| plos.asp | Vector of times at which the burst onset occurs |
| plos.sam | Sampled speech data of plos |
| plos.dft | Spectral trackdata object of plos |

You can verify that plos.dft is a spectral (trackdata) object by entering is.spectral(plos.dft), which is to ask the question: is this a spectral object? Since the sampling frequency for this dataset was 16000 Hz and since the window size used to derive the data was 256 points, then, following the discussion of the previous section,

there should be 256/2 + 1 = 129 spectral components equally spaced between 0 Hz and the critical Nyquist, 8000 Hz.

This information is given as follows:

```
ncol(plos.dft)
129
trackfreq(plos.dft)
0.0   62.5  128.0  187.5  250.0… 7812.5 7878.0 7937.5 8000.0
```

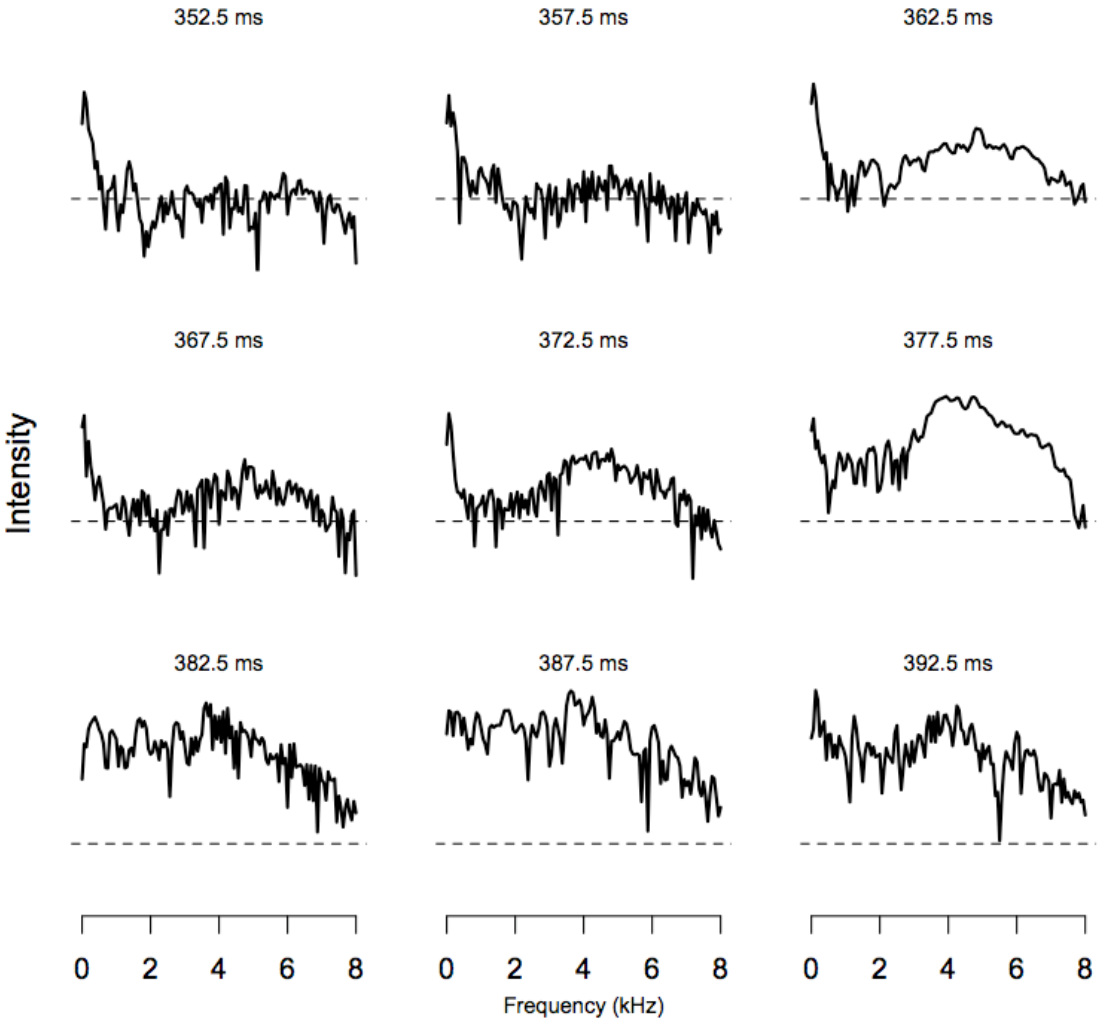



Fig. 8.9: 256-point spectra calculated at 5 ms intervals between the acoustic onset of a closure and the onset of periodicity of a /d/ in /daʊ/. The midpoint time of the window over which the DFT was calculated is shown above each spectrum. The release of the stop is at 378 ms (and can be related to the rise in the energy of the spectrum at 377.ms above 3 kHz). The horizontal dashed line is at 0 dB.

Compatibly with the discussion from 8.1.7, the above information shows that the spectral components occur at 0 Hz, 62.5 Hz and so on at intervals of 62.5 Hz up to 8000 Hz. We were also told that the frame shift in calculating the DFT was 5 ms. So how many spectra, or spectral slices, can be expected for e.g., the 47$^{th}$ segment? The 47$^{th}$ segment is a /d/ of duration just over 80 ms:

```
plos.l[47]
"d"
dur(plos[47,])
80.266
```

So for this segment, around 16 spectral slices (80/5) are to be expected. Here are the times at which these slices occur:

```
tracktimes(plos.dft[47,])
317.5 322.5 327.5 332.5 337.5 ... 437.5 392.5
```

They are at 5 ms intervals and as length(tracktimes(plos.dft[47,])) shows, there are indeed 16 of them. So to be completely clear: plos.dft[47,] contains 16 spectral slices each of 129 values and spaced on the time-axis at 5 ms intervals. A plot of the first eight of these, together with the times at which they occur is shown in Fig. 8.9:

```
dat <- frames(plos.dft[47,])[8:16,]
times = tracktimes(dat)
par(mfrow=c(3,3)); par(mar=c(2, 1, .4, 1))
ylim =c(-20, 50)
for(j in 1:nrow(dat)){
plot(dat[j,],  bty="n", axes=F, ylim=ylim, lwd=2)
if(any(j == c(7:9)))
{
freqs = seq(0, 8000, by=2000)
axis(side=1, at=freqs, labels=as.character(freqs/1000))
}
abline(h=0, lty=2, lwd=1)
mtext(paste(times[j], "ms"), cex=.5)
if(j == 8)
mtext("Frequency (kHz)", 1, 2, cex=.5)
if(j == 4)
mtext("Intensity", 2, 1, cex=.75)
}
```

Since these spectral data were obtained with a DFT of 256 points and a sampling frequency of 16000 Hz, the window length is equal to a duration of (256 x 1000/16000) = 256/16 = 16 ms. The temporal midpoint of the DFT window is shown above each spectral slice. For example, the 6$^{th}$ spectral slice has a time stamp of 377.5 ms and so the DFT window that was used to calculate this spectrum extends from 377.5 – 8 = 369.5 ms to 377.5 + 8 = 388.5 ms

The dcut() function can be used to extract trackdata between two time points, or at a single time point, either from millisecond times or proportionally (see 5.5.3). Thus spectral data at the temporal midpoint of the segment list is given by:

```
plos.dft.5 = dcut(plos.dft, .5, prop=T)
```

For spectral trackdata objects and spectral matrices, what comes after the comma refers not to column numbers directly, but to **frequencies**. The subscripting pulls out those spectral components that are closest to the frequencies specified. Here are some examples:

```
# Trackdata object 2000-4000 Hz
spec = plos.dft[,2000:4000]
```

The frequencies can be verified as follows:

```
trackfreq(spec)
2000.0 2062.5 2128.0 2187.5 2250.0 2312.5 2378.0 2437.5 2500.0 2562.5
2628.0 2687.5 2750.0 2812.5 2878.0 2937.5 3000.0 3062.5 3128.0 3187.5
3250.0 3312.5 3378.0 3437.5 3500.0 3562.5 3628.0 3687.5 3750.0 3812.5
3878.0 3937.5 4000.0
```

Notice that spec now has 33 columns ( ncol(spec) ) with each column including data from the frequencies listed above. Some further examples of handling spectral objects in Emu-R are as follows:

```
# Trackdata object 0-3000 Hz
spec = plos.dft[,0:3000]

# As above, but of segments 1-4 only
spec = plos.dft[1:4,0:3000]
dim(spec)
4 49

# Trackdata object of the data at 1400 Hz only
spec = plos.dft[,1400]
trackfreq(spec)
1375
ncol(spec)
1

# Trackdata object of all frequencies except 0-2000 Hz and except 4000-7500 Hz
spec = plos.dft[,-c(0:2000, 4000:7500)]
trackfreq(spec)
2062.5 2128.0 2187.5 2250.0 2312.5 2378.0 2437.5 2500.0 2562.5 2628.0
2687.5 2750.0 2812.5 2878.0 2937.5 3000.0 3062.5 3128.0 3187.5 3250.0
3312.5 3378.0 3437.5 3500.0 3562.5 3628.0 3687.5 3750.0 3812.5 3878.0
3937.5 7562.5 7628.0 7687.5 7750.0 7812.5 7878.0 7937.5 8000.0
```

Use -1 to get rid of the DC offset, i.e., of the spectral component at 0 Hz:

```
# DC offset (0 Hz), segments 1-10:
spec = plos.dft[1:10,0]
# All spectral components of segments 1-10, except the DC offset
spec = plos.dft[1:10,-1]
```

Exactly the same machinery works on spectral matrices that are the output of dcut(). Thus:

```
# Spectral data at the temporal midpoint
plos.dft.5 = dcut(plos.dft, .5, prop=T)
# As above, but 1-3 kHz
spec = plos.dft.5[,1000:3000]
# This therefore gives a plot in the spectral range 1-3 kHz
# at the temporal midpoint for the 11th segment
```

```
plot(spec[11,], type="l")
```

Notice that, if you pull out a **single** row from a spectral matrix, the result is no longer a (spectral) **matrix** but a (spectral) **vector**: in this special case, just put the desired frequencies inside square brackets without a comma (as when indexing any vector). Thus:

```
spec = plos.dft.5[10,]
class(spec)
"numeric"   "spectral"
# plot the values between 1000-2000 Hz
plot(spec[1000:2000])
```

**8.2. Spectral average, sum, ratio, difference, slope.**

The aim in this section is to consider some of the ways in which spectra can be parameterised for distinguishing between different phonetic categories. A spectral parameter almost always involves some form of data reduction of the usually very large number of spectral components that are derived from the Fourier analysis of a waveform. One of the simplest forms of spectral data reduction is averaging and summing energy in frequency bands. The data to which these parameters will be applied is given below: these are German fricatives produced by an adult male speaker in read sentences taken from the `kielread` database. The fricatives were produced in an intervocalic environment and are phonetically (rather than just phonologically) voiceless and voiced respectively and they occur respectively in words like [bø:zə] (*böse*, *angry*) and [vasə] (*Wasser*, *water*):

| | |
|---|---|
| fric | Segment list of intervocalic [s,z] |
| fric.l | Vector of (phonetic) labels |
| fric.w | Vector of word labels from which the fricatives were taken |
| fric.dft | Spectral trackdata object (256 point DFT, 16000 Hz sampling freq.) |

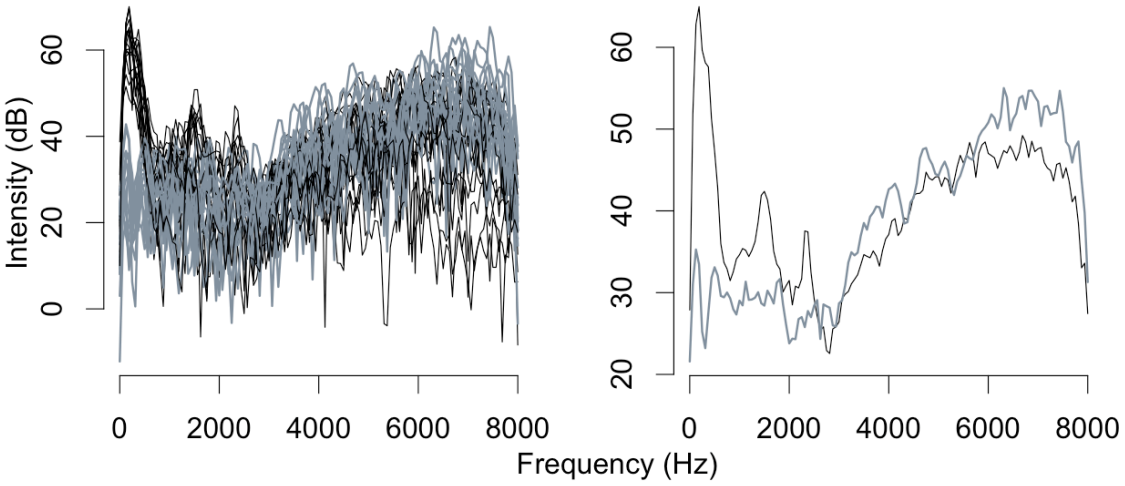



Fig. 8.10: Spectra (left) and ensemble-averaged spectra (right) of [s] (gray) and [z] (black).

A ensemble plot of all the [s] and [z] fricatives at the temporal midpoint was produced as follows:

```
fric.dft.5 = dcut(fric.dft, .5, prop=T)
plot(fric.dft.5, fric.l)
```

It can also be helpful to look at averaged spectral plots per category. However, since decibels are logarithms, then addition and subtraction really mean multiplication and division respectively; consequently, any arithmetic operation which, like averaging, involves summation, cannot (or should not) be directly applied to logarithms. For example, the average of 0 dB and 10 dB is not 5 dB. Instead, the values first have to be converted back into a linear scale where the averaging (or other numerical operation) takes place; the result can then be converted back into decibels. Since decibels are essentially power ratios, they could be converted to the (linear) power scale prior to averaging. This can be done by dividing by 10 and then taking the anti-logarithm (i.e., raising to a power of 10). Under this definition, the average of 0 dB and 10 dB is calculated as follows:

```
dbvals = c(0, 10)
# Convert to powers
pow = 10^(dbvals/10)
# Take the average
pow.mean = mean(pow)
# Convert back to decibels
10 * log(pow.mean, base=10)
7.403627
```

For plots of spectral data, this conversion to powers before averaging is done within the plot() function by including the argument power=T. Thus the ensemble averaged spectra are given by:

```
plot(fric.dft.5, fric.l, fun=mean, power=T)
```

There is clearly a greater amount of energy below about 500 Hz for [z] (Fig. 8.10, right), and this is to be expected because of the influence of the fundamental frequency on the spectrum in the case of voiced [z].

One way of quantifying this observed difference between [s] and [z] is to sum or to average the spectral energy in a low frequency band. This can be done with the fapply() function, which applies a function to the spectral separately per segment. For example, the mean spectral energy (ignoring the issue to do with power discussed above) at the temporal midpoint of the 10th segment is mean(fric.dft.5[10,]). It is equivalently given by fapply(fric.dft.5[10,], mean). The same function can be used to calculate the mean spectral levels *separately for each segment*: thus fapply(fric.dft.5[1:3,], mean) returns three values, one mean value per segment. The first argument to fapply() can be any spectral object and the second argument any function that gives sensible results when applied to a spectral vector, i.e., to a single spectrum. Thus since in the above example mean() can be applied to a single spectrum mean(fric.dft.5[10,]), then the same function can also be used inside fapply() to calculate the mean spectral level for each segment.

In the above example, the first argument to fapply() was a spectral *matrix* (of spectra at the temporal midpoint with one spectrum per segment) but the first argument could also be a spectral t*rackdata* object. For example, m = fapply(fric.dft[8,], mean) calculates the energy in each spectrum of the 8th segment. More specifically, fric.dft is a spectral trackdata object that contains spectra between the start and end times of each segment in the segment list fric. For the 8th segment, there are 9 such spectra (as shown by nrow(frames(fric.dft[8,])) ) and the times at which these spectra occur are:

```
tracktimes(fric.dft[8,])
787.5 792.5 797.5 802.5 807.5 812.5 817.5 822.5 827.5
```

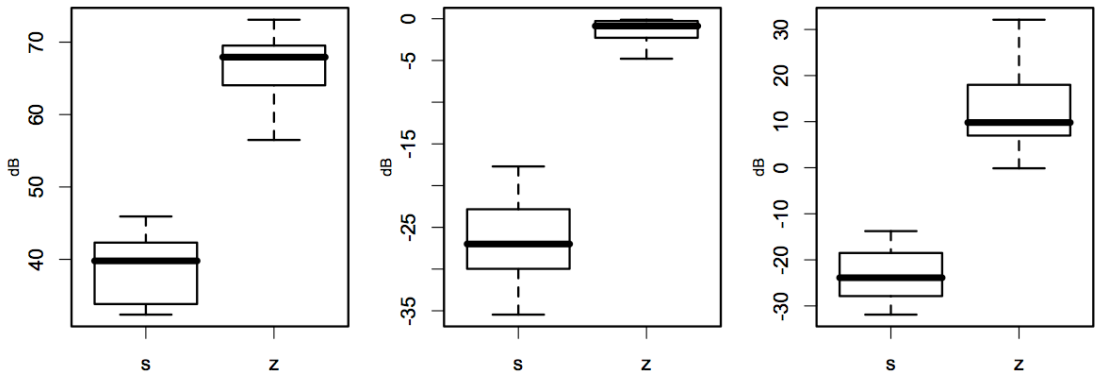



Fig. 8.11: Distribution of [s] and [z] on: summed energy in the 0-500 Hz region (left), the ratio of energy in this region to that in the total spectrum (middle) and the ratio of energy in this region to the summed energy in the 6000 -7000 Hz range (right).

Consequently, the command m = fapply(fric.dft[8,], mean) returns 9 mean values, one at each of the above times and with one mean per spectrum. Moreover m is a trackdata object (because it stores signals as a function of time) and is amenable to all of the usual functions for manipulating trackdata objects: thus plot(m) is a plot of the mean level of the spectrum for the 8th segment between its start and end time, frames(m) returns the 9 mean values and tracktimes(m) returns the times at which they occur.

The function fapply() also has an optional argument power=T that converts from decibels to power values: for the reasons discussed earlier, this should be done when averaging or summing energy in a spectrum. Finally, then, the sum and mean energy in the spectral matrix containing spectra, one per segment, of the [s] and [z] segments in the frequency band 0-500 Hz is given by:

```
s500 <- fapply(fric.dft.5[,0:500], sum, power=T)
m500 <- fapply(fric.dft.5[,0:500], mean, power=T)
```

A boxplot (Fig. 8.11) for the first of these, the summed energy in the 0-500 Hz band, shows that this parameter distinguishes between [s,z] very effectively:

```
boxplot(s500 ~ fric.l, ylab="dB")
```

When two categories are compared on amplitude or intensity levels, there is usually an implicit assumption that they are not artificially affected by variations in loudness caused, for example, because the speaker did not keep at a constant distance

from the microphone, or because the speaker happened to produce some utterance with greater loudness than others. This is not especially likely for the present corpus, but one way of reducing this kind of artefact is to calculate the **ratio of energy levels**. More specifically, the ratio of energy in the 0-500 Hz band relative to the total energy in the spectrum should not be expected to vary too much with, say, variations in speaker distance from the microphone. The sum of the energy in the entire spectrum (0-8000 Hz) is given by:

```
stotal <- fapply(fric.dft.5, sum, power=T)
```

To get the desired energy ratio, subtract one from the other in decibels (which is equivalent to dividing the power of one by the power of the other, i.e., equivalent to calculating a ratio of powers):

```
s500r <-  s500 - stotal
# Middle panel Fig. 8.11
boxplot(s500r ~ fric.l, ylab="dB")
```

The energy ratio could be calculated between two separate bands, rather than between one band and the energy in the entire spectrum, and this too would have a similar effect of reducing some of the artificially induced differences in the amplitude level discussed earlier. Since Fig. 8.10 shows that [s] seems to have marginally more energy around 6500 Hz, then the category differences might also emerge by calculating the ratio of summed energy in the 0-500 Hz band to that in the 6000 - 7000 Hz band (Fig. 8.11, right panel):

```
shigh <- fapply(fric.dft.5[,6000:7000], sum, power=T)
s500tohigh <- s500 - shigh
boxplot(s500tohigh ~ factor(fric.l), ylab="dB")
```

Another way of normalising for artificially induced differences in the amplitude level is to calculate **difference spectra**, obtained as its name suggests by subtracting one spectrum from another, usually in the same syllable, word, or phrase. (A difference spectrum has already been calculated in the discussion of pre-emphasis in 8.1.8). But independently of this consideration, difference spectra have been shown to be valuable in distinguish place of articulation in both oral (e.g., Lahiri et al., 1984) and nasal (e.g., Kurowski & Blumstein, 1984; Harrington, 1994) stops. A difference spectrum shows how the energy in the spectrum changes between two time points. For the following example, difference spectra will be calculated for the database of plosives considered earlier. More specifically, spectra are to be calculated (a) during the closure 20 ms before the release of the stop and (b) 10 ms after the release of the stop; a difference spectrum will then be obtained by subtracting (a) from (b). Because /d/ tends to have a lot more energy in the upper part of the spectrum at stop release than /b/, then such differences between these two categories should show up above about 4000 Hz in the difference spectrum. The ensemble averaged plot of the difference spectrum as well as the summed energy values in the difference spectrum between 4-7 kHz confirms this (Fig. 8.12):

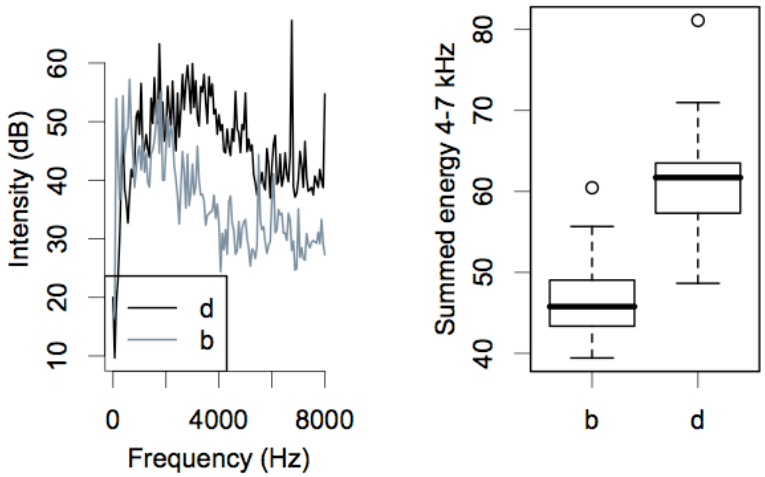



Fig. 8.12: Left: Ensemble-averaged difference spectra for [b] and [d] (left) calculated from spectra taken 20 ms and 10 ms after the stop release. Right: the distributions of [b] and [d] on the change in summed energy before and after the burst in the 4000 – 7000 Hz range.

```
# (a) Spectra 20 ms before the stop release
before = dcut(plos.dft, plos.asp-20)
# (b) Spectra 10 ms after the stop release
after = dcut(plos.dft, plos.asp+10)
# Difference spectra: (b) - (a)
d = after - before
# Ensemble-averaged plot of the difference spectra separately for /b, d/
par(mfrow=c(1,2))
plot(d, plos.l, fun=mean, power=T, xlab="Frequency (Hz)", ylab="Intensity (dB)",
leg="bottomleft")
# Summed energy in the difference spectra 4-7 kHz
dsum = fapply(d[,4000:7000], sum, power=T)
# Boxplot (Fig. 8.12)
boxplot(dsum ~ factor(plos.l), ylab="Summed energy 4-7 kHz")
```

The (linear) **spectral slope** is a parameter that reduces a spectrum to a pair of values, the intercept and the slope of the (straight) line of best through the spectral values. There is evidence that these differences in spectral slope are important for distinguishing between places of articulation in oral stops (e.g., Blumstein & Stevens, 1979, 1980). The R function lm() can be used to compute the straight line of best fit (a linear regression based on least squares), as follows:

```
# Spectrum of the 1st segment 10 ms after the stop release (Fig. 8.13, left)
plot(after[1,], xlab="Frequency (Hz)", ylab="Intensity (dB)")
# Calculate the coefficients of the linear regression equation
m = lm(after[1,] ~ trackfreq(after))
# Superimpose the regression equation
abline(m)
m$coeff
# The intercept and slope
```

```
   Intercept            X
54.523549132 -0.003075377
```

The slope is negative because, as Fig. 8.13 (left panel) shows, the spectrum falls with increasing frequency.

A first step is to look at ensemble-averaged spectra of [b,d] in order to decide roughly over which frequency range the slopes should be calculated:

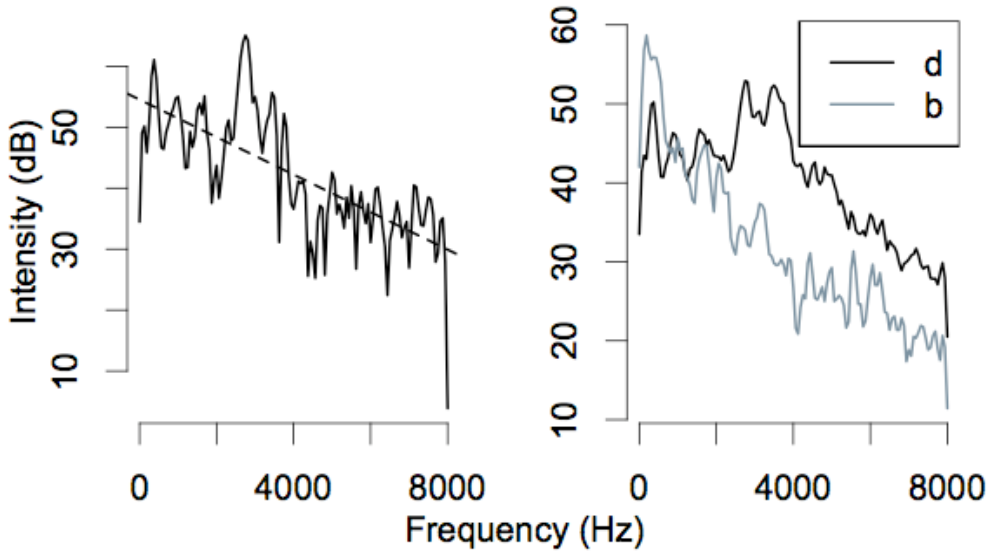



Fig. 8.13: Left: A spectrum of a [d] 10 ms after the stop release showing the line of best fit (dotted) based on least squares regression. Right: Ensemble-averaged spectra for [b] and [d] calculated 10 ms after the stop release.

```
# Plot averaged spectra per category (Fig. 8.13, right panel)
plot(after, plos.l, fun=mean, power=T, xlab="Frequency (Hz)",
ylab="Intensity (dB)")
```

Spectral slopes will now be calculated in the frequency range 500 – 4000 Hz, because as the right panel of Fig. 8.13 shows, this is where the slope differences between the categories emerge most clearly. In order to calculate slopes for the entire matrix of spectral data (rather than for a single segment, as done in the left panel of Fig. 8.13), the fapply(x, fun) function can be used again, where fun is now a function that calculates the slope per spectrum. First of all, the function for calculating the slope has to be written:

```
slope <- function(x)
{
# Calculate the intercept and slope  in a spectral vector
lm(x ~ trackfreq(x))$coeff
}
```

The coefficients of the regression through the spectrum for the first segment could be obtained from:

```
slope(after[1,])
   Intercept            X
54.523549132 -0.003075377
```

The same function is used inside fapply() in order to calculate the slopes for all spectra in the 500-4000 Hz range:

```
m <- fapply(after[,500:4000], slope)
```

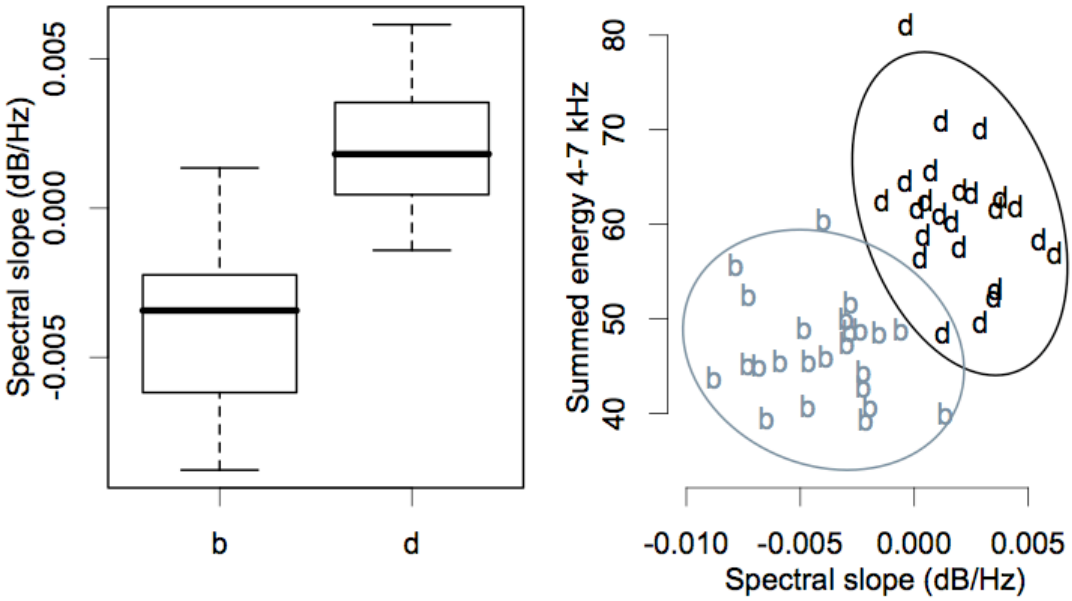



Fig. 8.14: Left: Distribution of [b] and [d] on the slope of the spectrum in the 500-4000 Hz range calculated 10 ms after the stop release. Right: 95% ellipse plots on this parameter (*x*-axis) and the summed energy in the 4-7 kHz (*y*-axis) range also calculated 10 ms after stop release.

The object `m` is a two-columned matrix with the intercept and slope values per segment in columns 1 and 2 respectively. So the extent to which [b,d] are distinguished by spectral slope calculated in the 500-4000 Hz range from a spectrum taken 10 ms after the stop release can be displayed as follows (Fig. 8.14, left panel):

```
boxplot(m[,2] ~ plos.l, ylab="Spectral slope (dB/Hz)")
```

As figure 8.14 shows, the slope for [b] in this frequency range is, consistently with e.g. Blumstein & Stevens (1979, 1980), predominantly negative whereas for [d] it is mostly positive.

It is interesting to consider the extent to which this parameter and the one calculated earlier, the sum of the energy from a difference spectrum, both contribute to the [b,d] distinction. The eplot() function could be used to look at the distribution of these categories on these parameters together (Fig. 8.14, right panel):

```
eplot(cbind(m[,2], dsum), plos.l, dopoints=T, xlab="Spectral slope (dB/Hz)",
ylab="Summed energy 4-7 kHz")
```

In fact, both parameters together give just about 100% separation between [b,d]. However, closer inspection of the right panel of Fig. 8.14 shows that most of the 'work' in separating them is done by the spectral slope parameter: if you draw a vertical line at around -0.00168 (abline(v= -0.00168)) , then you will see that only 2-3 [b] tokens fall on the wrong , i.e.[d]-side, of the line. So the slope parameter is clearly very effective, and more so than the difference parameter.

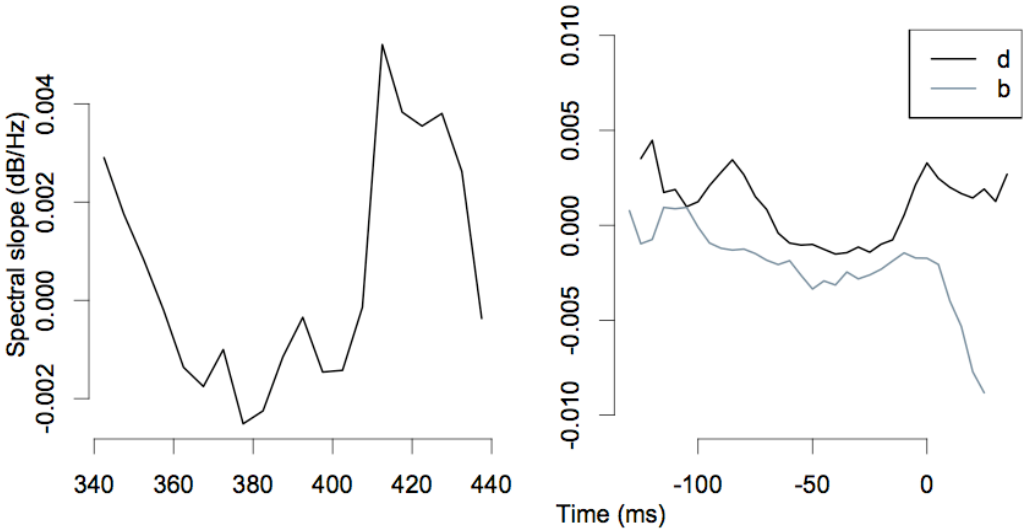



Fig. 8.15: Left: The spectral slope in the 500-4000 Hz range plotted as a function of time from closure onset to the burst offset/vowel onset for a [d] token. Right: The spectral slope over the same temporal extent averaged separately across all [b] and [d] tokens, after synchronisation at the burst onset ($t$ = 0 ms).

Finally, these functions have been applied so far to spectral slices at a single point in time, but as discussed earlier, the slopes at multiple points in time between from the onset to the offset of each segment by applying fapply() to a spectral trackdata object. For example, the following command calculates the spectral slope for each spectrum between the segment onset and offset:

```
allvals = fapply(plos.dft[,500:4000], slope)
```

allvals is now a two-dimensional trackdata object containing the intercept and slope for each spectrum at 5 ms intervals per segment. So it is possible to inspect how the spectral slope changes between the onset and offset of e.g., the 10$^{th}$ segment as follows (left panel, Fig. 8.15):

```
dplot(allvals[,2], plos.l, plos.asp, prop=F, average=T, xlab="Time (ms)",
ylab="Spectral slope (dB/Hz)")
```

The right panel of Fig. 8.15 shows an ensemble plot of these data averaged separately by phonetic category after synchronisation at the burst onset: these data show very clearly how, beyond the burst onset at $t$ = 0 ms, labials have falling, but alveolars rising spectral slopes. This plot of the spectral slope as a function of time was produced with:

```
dplot(allvals[,2], plos.l, plos.asp, ylab="Spectral slope (dB/Hz)", xlab="Time (ms)")
```

**8.3. Spectral moments**

The types of parameters discussed in the preceding section can often effectively distinguish between spectra of different categories. Another useful way of quantifying spectral differences is to reduce the spectrum to a small number of parameters that encode basic properties of its shape. This can be done by calculating what are often called **spectral moments** (e.g., Forrest et al., 1988). The function for calculating moments is borrowed from statistics in which the first four moments describe the **mean**, **variance**, **skew**, and **kurtosis** of a probability distribution.

Before looking at *spectral* moments, it will be helpful to consider (statistical) moments in general. The matrix bridge includes some hypothetical data of counts that were made on three separate days of the number of cars crossing a bridge at hourly intervals. It looks like this:

```
bridge
   Mon Tues Wed
0    9    1   0
1   35    1   1
2   68    5   7
3   94    4  27
...
```

The first row shows that between midday and 1 p.m., 9 cars were counted on Monday, one on Tuesday, and none on Wednesday. The second row has the same meaning but is the count of cars between 1 p.m. and 2 p.m. Fig. 8.16 shows the distribution of the counts on these three separate days:

```
par(mfrow=c(1,3))
barplot(bridge[,1], ylab="Observed number of cars", main="Monday")
barplot(bridge[,2], xlab="Hours", main="Tuesday")
barplot(bridge[,3], main="Wednesday")
```

There are obviously overall differences in the shape of these distributions. The plot for Monday is skewed to the left, the one for Tuesday is a mirror-image of the Monday data and is skewed to the right. The data for Wednesday is not as dispersed as for the other days: that is, it has more of its values concentrated around the mean.

Leaving aside kurtosis for the present, the following predictions can be made:

- Monday's mean (1st moment) is somewhere around 4-5 p.m. while the mean for Tuesday is a good deal higher (later), nearer 8 or 9 p.m. The mean for Wednesday seems to be between these two, around 6-7 p.m.

- The values for Wednesday are not as spread out as for Monday or Tuesday: it is likely therefore that its variance (2nd moment) will be lower than for those of the other two days.

- As already observed, Monday, Tuesday, and Wednesday are all likely to have different values for skew (3rd moment).

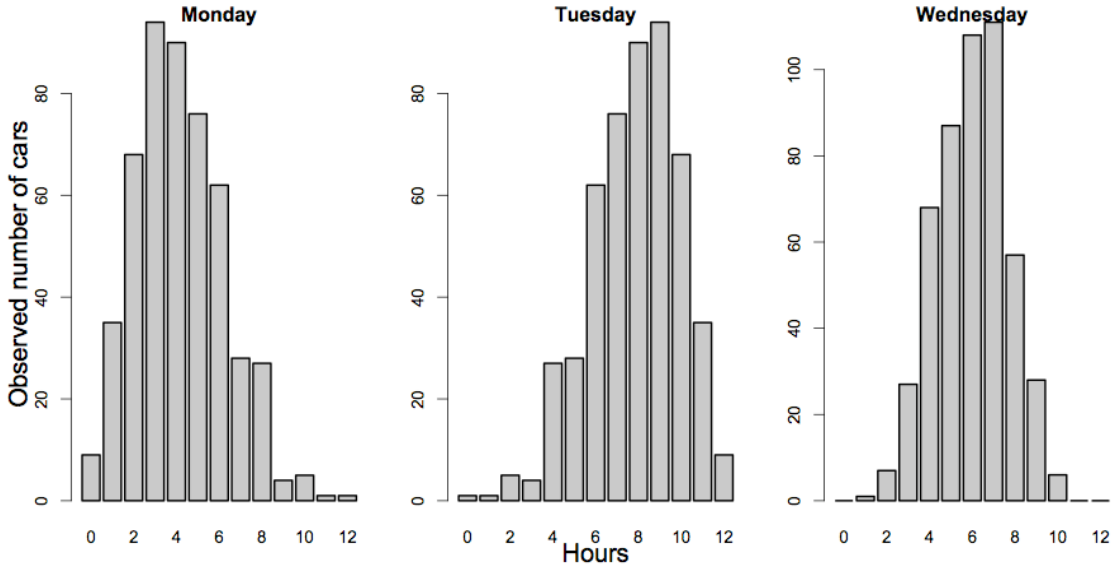



Fig. 8.16: Hypothetical data of the count of the number of cars crossing a bridge in a 12 hour period.

The core calculation of moments involves the formula:

$$\frac{\sum f(x-k)^m}{\sum f}$$

In which *f* is the observed frequency (observed number of cars in this example) *x* is the class (hours from 0 to 12 in our example), *m* is the moment (*m* =1, 2, 3, 4) and *k* is a constant (see also Harrington, 2009). The above formula can be translated directly into R as:

```
sum(f * (x - k)^m) / sum(f)
```

This formula can be put into a function that defaults to calculating the first moment with m defaulting to 1 and k to 0 (the constant is zero when calculating the first moment):

```
mfun <- function(x, f, k = 0, m = 1)
{
sum(f * (x - k)^m) / sum(f)
}
```

To get the mean or first moment, the class, x, has to be created which for the present bridge data consists of the integers 0 through to 12:

```
hours = 0:12
```

So the first moment for the Monday data is

```
first = mfun(hours, bridge[,1])
```

For the other moments, the constant, k, is set equal to the first moment that has just been calculated, and m, the moment number to be calculated, is set to 2, 3, 4 respectively. Thus for the Monday data:

```
second = mfun(hours, bridge[,1], first, 2)
third = mfun(hours, bridge[,1], first, 3)
fourth = mfun(hours, bridge[,1], first, 4)
```

Two more adjustments need to be made. The third moment has to be divided by the second moment raised to the power 1.5:

```
third = third/second^1.5
```

and the 4th moment is divided by the square of the second moment and then 3 is subtracted from the result (the subtraction of 3 is done to make a normal distribution have zero kurtosis):

```
fourth = fourth/second^2 - 3
```

There is a function in the Emu-R library, moments(count, x) that carries out all of the above calculations. In this function, count is the observed frequency and x the class. So the first four moments for the Monday data are given by moments(bridge[,1], hours)[3] while all four moments for Monday, Tuesday, Wednesday are given by:

```
t(apply(bridge, 2, moments, hours))
      [,1]     [,2]        [,3]        [,4]
Mon  4.172 4.422416  0.47063226  0.08290827
Tues 7.828 4.422416 -0.47063226  0.08290827
Wed  8.992 2.851936 -0.07963716 -0.39367681
```

(t() is the transpose function and does nothing more than turn the resulting matrix the other way round so that the days of the week appear as rows, and the first four moments are in the columns).

As expected, the first moment (column 1) is at about 4-5 p.m. for Monday, close to 6 p.m for Wednesday and higher (later) than this for Tuesday. Also, as expected, the variance (second moment), whose unit in this example is *hours*$^2$, is least for Wednesday.

The skew is a dimensionless number that varies between -1 and 1. When the skew is zero, then the values are distributed evenly about the mean, as they are for a Gaussian normal distribution. When the values are skewed to the left so that there is a longer tail to the right, then kurtosis is positive (as it is for the Monday data); the skew is negative when the values are skewed to the right (as for the Tuesday data).

---

[3] The x argument can be omitted: if it is missing, then it defaults to 0:(*N*-1), where *N* is the length of count. So moments(bridge[,1]) gives the same result.

Finally, the kurtosis is also a dimensionless number that is zero for a normal Gaussian distribution. Kurtosis is often described as a measure of how 'peaked' a distribution is. In very general terms, if the distribution is flat – that is, its shape looks rectangular – then kurtosis is negative, whereas if the distribution is peaked, then kurtosis is typically positive. However, this general assumption only applies if the distributions are not skewed (skewed distributions tend to have positive kurtosis) and kurtosis depends not just on the peak but also on whether there are high values at the extremes of the distribution (see Wuensch, 2006 for some good examples of this). For all these reasons - and in particular in view of the fact that spectra are not usually symmetrical about the frequency axis - it is quite difficult to use kurtosis to make predictions about the spectral differences between phonetic categories.

When *spectral* moments are calculated, then x and f in the above formula (and function) are the frequency in Hz and the corresponding dB values (and not the other way round!). This can be understood most easily by having another look at Fig. 8.16 and pretending it is a spectrum with a horizontal axis of frequency in Hz and a vertical axis of dB. On this assumption, the calculation of the 1st spectral moment results in a value in Hz (analogous to a value in hours for the worked example above), and the second spectral moment a value in $Hz^2$, while the 3rd and 4th spectral moments are dimensionless, as before.

Spectral moments will be investigated in the matrix of spectra at the temporal midpoint of the [s,z] fricatives extracted earlier with:

```
fric.dft.5 = dcut(fric.dft, .5, prop=T)
```

To apply the moments(count, x) function, count is a vector of dB values and x, the class, contains the frequencies at which these dB values occur. Since for e.g. the 3rd segment, the dB values are given by fric.dft.5[3,] and their frequencies by trackfreq(fric.dft.5) then the moments for this spectrum must be:

```
moments(fric.dft.5[3,], trackfreq(fric.dft.5))
```

However, the above command may sometimes fail. This is because some of the dB values can be negative and yet the calculation of moments assumes that the values for the observations are positive (it would never be possible, for example, to have a negative value in counting how many cars crossed the bridge in an hourly time interval!). To overcome this problem, the dB values are typically rescaled in calculating moments so that the minimum dB value is set to zero (as a result of which all dB values are positive and the smallest value is 0 dB). The moments() function does this whenever the argument minval=T is included. Thus:

```
moments(fric.dft.5[3,], trackfreq(fric.dft.5), minval=T)
```

Finally, since the moments() function can work out for itself the frequencies if it is supplied with spectral data, the second argument can be dropped. So the spectral moments for the 3rd segment are equivalently and more simply given by:

```
moments(fric.dft.5[3,], minval=T)
```

In order to calculate spectral moments not just for the 3rd but for all the segments, use the fapply(x, y) function as before, thus:

```
m <- fapply(fric.dft.5, moments, minval=T)
```

i.e. m[3,] gives the back the same result as moments(fric.dft.5[3,], minval=T).

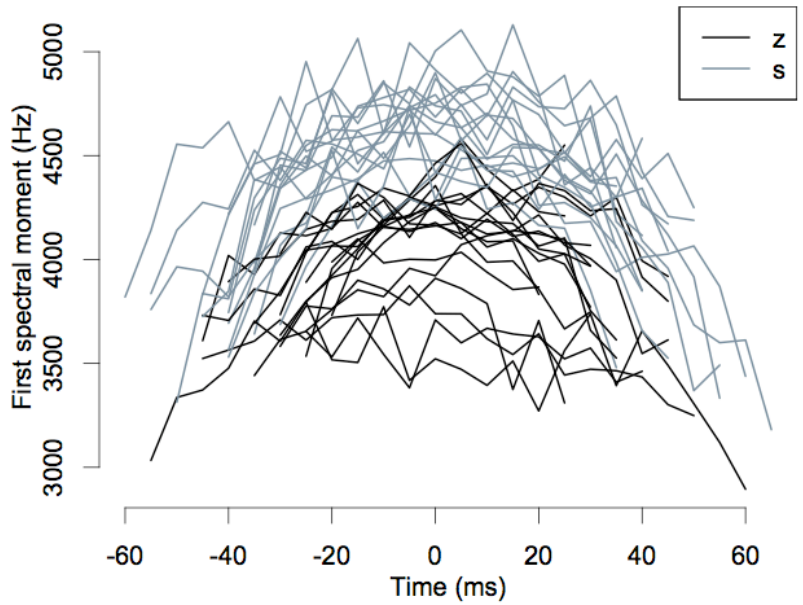



Fig. 8.17: First spectral moment as a function of time for [s] (gray) and [z] (black). The tracks are synchronised at $t$ = 0 ms, the segment midpoint.

Since fapply() can be used to apply a function to a trackdata object, then a plot the 1st spectral moment from the onset to the offset of the third segment is given by:

```
m <- fapply(fric.dft, moments, minval=T)
plot(m[3,1], xlab="Time (ms)", ylab="1st spectral moment (Hz)", type="l")
```

and an ensemble plot of the 1st spectral moment for each segment between the acoustic onset and offset, synchronised at the temporal midpoint and coded for phonetic category (Fig. 8.17) is given by:

```
dplot(m[,1], fric.l, 0.5, xlab="Time (ms)", ylab="First spectral moment (Hz)")
```

The first spectral moment for [s] is higher than for [z] because, as Fig. 8.10 had shown, [s] has less energy at low frequencies and slightly more energy at high frequencies than [z].

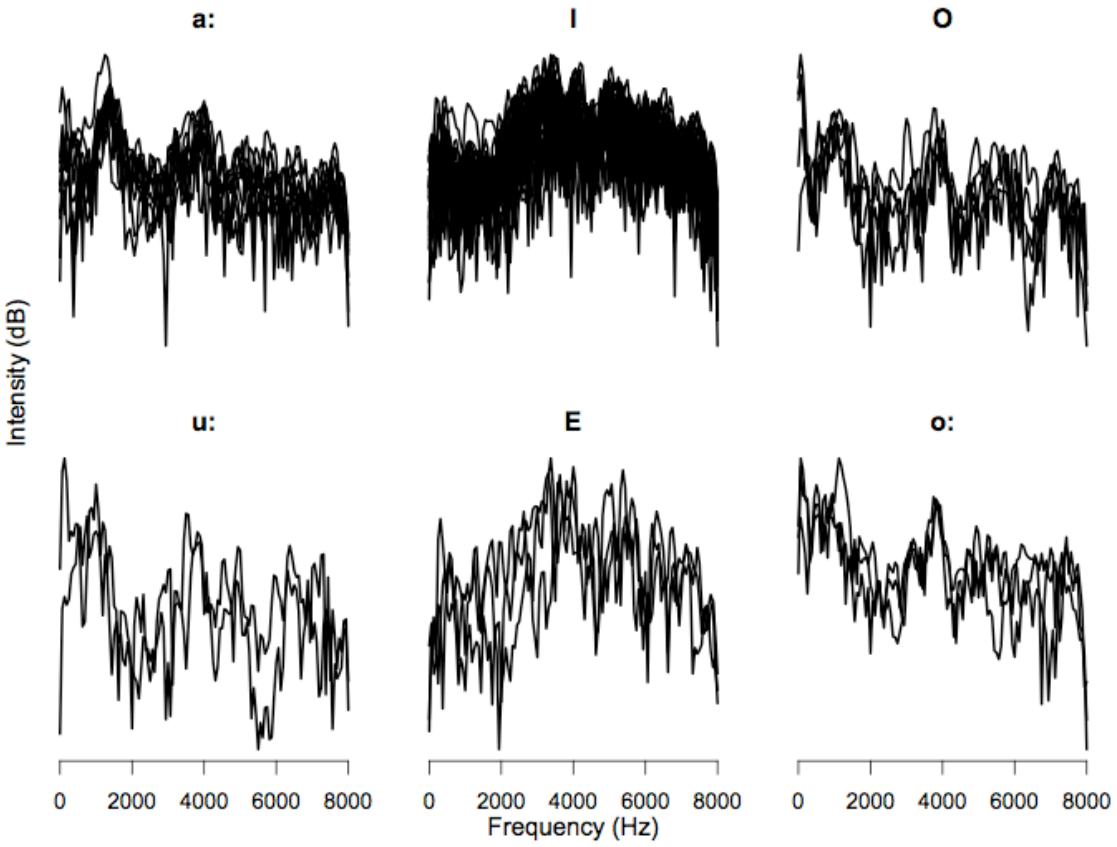



Fig. 8.18: Spectra calculated at the temporal midpoint of post-vocalic voiceless dorsal fricatives in German shown separately as a function of the preceding vowel context (the vowel context is shown above each spectral plot).

Finally spectral moments will also be used to assess the extent to which palatal and velar fricatives in German are separated which was a theme presented using electropalatographic data in the preceding Chapter. The dorsal fricatives in the Kiel Corpus of Read Speech were transcribed with either a front allophone [ç] (MRPA/SAMPA `C`) or with a back allophone [x] (MRPA/SAMPA `x`). Recall from the EPG analysis in Chapter 7 that there seems to be articulatory evidence for a categorical distinction between these two fricatives. The same problem will be analysed by comparing the fricative spectra of dorsal fricatives following vowels differing in phonetic backness. The dataset in this case is acoustic data of a female speaker taken from the Kiel corpus of read speech:

```
dorfric      # Segment list, postvocalic German dorsal fricatives
dorfric.l    # A vector of their labels: C or x
dorfric.lv   # A vector of the vowel labels preceding these fricatives
dorfric.w    # A vector of the word labels containing these fricatives
dorfric.dft  # Spectral trackdata object, 256 point DFT, 16 kHz
             sampling frequency
```

The preceding vowel types arranged from phonetically front to back are "I", "E", "a:", "O", "o:", "u:" corresponding to IPA [ɪ, ɛ, a:, ɔ, o:, u:] (the long [a:] and short [a] that distinguish German *Lamm* (*lamb*) and *lahm* (*lame*) have been collapsed into a single

category because there was only one [a] token). As you will see from unique(paste(dorfric.lv, dorfric.l, sep=".")), the dorsal fricatives were transcribed in the Kiel Corpus with the palatal [ç] following [ɪ,ɛ] and with the velar [x] following the other vowels: the aim in the present analysis is to establish whether there is any acoustic justification for this. It will, as always, help to plot all the fricative spectra at the temporal midpoint separately by vowel category (Fig. 8.18):

```
# Fricative spectra at the temporal midpoint
dorfric.dft.5 = dcut(dorfric.dft, .5, prop=T)
# Overlaid spectra separately by vowel category
par(mfrow=c(2,3)); par(mar=rep(2, 4))
for(j in unique(dorfric.lv)){
temp = dorfric.lv==j
plot(dorfric.dft.5[temp,], main=j)
}
```

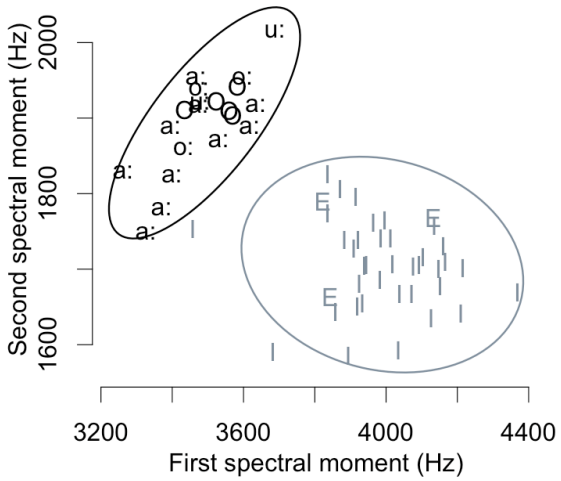



Fig. 8.19: 95% confidence ellipses for [ç] (gray) and [x] (black) in the plane of the first two spectral moments. The data were calculated at the fricatives' temporal midpoints. The labels of the vowels preceding the fricatives are marked at the fricatives' data points.

There do indeed seem to be differences in accordance with the transcriptions. After [ɪ, ɛ] there is a concentration of energy around 3-4 kHz whereas after the other vowels, the spectra fall with increasing frequency and there is not the same concentration of energy at any one frequency. Based on these plots, it seems likely that [ɪ, ɛ] have a lower second spectral moment, because the spectral energy is not so diffuse or spread along the frequency axis as it is after the other vowel categories. It is also possible that the mean, or first spectral moment, is higher for [ɪ, ɛ] because the other fricatives have proportionally slightly more energy in the lower part (0-2000 Hz) of the spectrum.

These predictions can be tested by calculating spectral moments for the fricatives shown in Fig. 8.18. In calculating moments, researchers sometimes leave out at least the DC offset (frequency at 0 Hz) which is just the average amplitude of the spectrum multiplied by the signal length, *N*; and it is also a good idea to cut out the frequencies near the Nyquist frequency because these are often not very reliable. In the example, the frequencies below 500 Hz and above 7000 Hz were removed. The 2nd spectral moment has also been converted into the spectral standard-deviation by taking

its square root, in order to have more manageable values in Hz (rather than values in the region of $10^5$ for $Hz^2$).

```
# Spectral moments 500 – 7000 Hz range
m = fapply(dorfric.dft.5[,500:7000], moments, minval=T)
# Spectral standard deviation in Hz
m[,2] = sqrt(m[,2])
```

Fig. 8.19 shows ellipse plots for these calculated spectral moments but showing the vowel labels at the data points:

```
eplot(m[,1:2], dorfric.l, dorfric.lv, dopoints=T, xlab="First spectral moment (Hz)",
ylab="Second spectral moment (Hz)")
```

This Figure shows that there does indeed seem to be a very good separation between the tokens labelled as `C` and `x`. Also, the relative positions according to context are roughly in accordance with the predictions made earlier from the spectra: the dorsal fricatives following the front vowels [ɪ,ɛ] have a high first spectral moment; and [ɔ, o:, u:] have a higher second spectral moment than [ɪ,ɛ] with the open vowel [a:] falling roughly between these vowel groups on this parameter.

### 8.4. The discrete cosine transformation

The discrete cosine transformation (DCT) is a mathematical operation that is very much like a discrete Fourier transform: it decomposes a signal into a set of sinusoids such that, when these are summed the same signal is reconstructed. One of the main differences between the two is that in the DCT the sinusoids are at *half*-cycles, i.e. at $k = 0, 0.5, 1, 1.5 \ldots ½(N – 1)$ rather than, as for the DFT, at *integer* cycles ($k = 0, 1, 2, \ldots N\text{-}1$). Another is that the output of the DCT are sinusoids with no phase. But any sinusoid with no phase is a cosine wave, so we may say that a DCT decomposes a signal into a set of cosine waves at frequencies $k = 0, 0.5, 1., 1.5, \ldots ½(N – 1)$; and hence the name, discrete *cosine* transformation.

The amplitudes of these cosine waves are called **DCT coefficients** and they are usually labelled from 0 to $N$-1. So the $0^{th}$ coefficient, DCT-0, is the amplitude of the $k = 0$ cosine wave; DCT-1, the $1^{st}$ coefficient, is the amplitude of the $k = 0.5$ cosine wave, and so on. Now it turns out that these DCT coefficients encode global properties of the signal's shape: in particular, as will be shown below, DCT-0, DCT-1, DCT-2 are proportional to the signal's mean, slope, and curvature respectively. For this reason, they serve the same important function as do spectral moments that were discussed in the previous section: DCT-coefficients, like spectral moments, reduce the quantity of information in a spectrum to a handful of values and importantly in such a way that different phonetic categories are often quite well separated (assuming these categories have differently shaped spectra).

The DCT has another useful application in phonetics: it can be used to smooth a signal. The way that this works is as follows. Suppose you are driving along a road and your task is to draw a crescent or an arc on a sheet of paper. Just as you draw the arc, the car goes over a long cattle-grid that produces a series of bumps throughout the car. You find that your drawing looks like an arc (assuming that size of the bumps is not too big), but also has a minor deviations from an arc that are caused by the regularly spaced bumps caused by the cattle grid. It turns that if you make a spectrum of what you drew, then the bits of the signal that are due to the bumps would show up at high frequencies.

This is to be expected: the bumps cause the pencil to change rapidly up and down above the arc that you are trying to draw. But anything that changes rapidly also shows up as a high frequency in the spectrum. Now we said that when a DCT is applied to a signal, then the signal is decomposed into cosine waves of progressively increasing frequency ($k$ = 0, .5, 1…). Therefore, if a DCT is applied to the bumpy arc, then the bumpy part should show up at high cosine frequencies. If all of the cosine waves are summed, then the same bumpy arc is reconstructed, but if only the first few frequencies are summed, then the influence of the bumps, which only affect the high frequencies, should be more or less removed: the net result is a smoother arc than the one that was drawn (more like the one you intended to draw), which can be called a DCT-smoothed signal. Moreover, the fewer cosine waves that are summed, the smoother the result: so a summation of the first three cosine waves at frequencies $k$ = 0, 0.5, 1 is going to produce a smoother result than summing cosine waves at frequencies $k$ = 0, 0.5, 1, 1.5, 2 cycles.

Now dB-spectra of speech derived using the Fourier analysis techniques discussed in this Chapter often have just this of property of bumpiness superimposed on a trend line. In voiced speech, the trend line is due to the formants which are responsible for a number of large peaks and troughs up and down the frequency axis. But, as discussed in 8.1, there is also a superimposed jaggedness or bumpiness that is the result of the harmonics due to vocal fold vibration that are spaced on the frequency axis at roughly pitch frequency. Thus in speech production, the filter is due to the shape of the vocal tract and produces a fairly smooth trend line in the spectrum while the source causes short-term changes (the bumpiness or saw-tooth effect). Following the earlier analogy, if a DCT is applied to a spectrum of speech but only the lower frequency cosine waves are summed, then the result will essentially be to filter out much of the bumpy part due to the source leaving predominantly the 'trend-line' due to the filter i.e, the spectrum that is due to the shape of the vocal tract.

Finally, before looking in closer detail at how the DCT can be applied in Emu-R, it should be mentioned that there is more or less an equivalence between the application of a DCT to a spectrum and **cepstral analysis**. In speech technology research, the output of a DCT applied to a spectrum is considered to be (a very close) approximation to cepstral analysis (e.g., Milnar & Shao, 2006), but the differences between the two are negligible for most kinds of phonetic and indeed speech signal processing analysis. Thus leaving these minor differences aside, the amplitudes of the ½ cycle cosine waves into which a spectrum is decomposed by a DCT analysis are the DCT-coefficients which are essentially **cepstral coefficients**. A plot of the DCT-coefficients as a function of the coefficient number (a plot of DCT-0, DCT-1, DCT-2.. as a function of 0, 1, 2..) is a **cepstrum**; and a DCT-smoothed spectrum is a cepstrally smoothed spectrum. In automatic speech recognition, speech scientists often parameterise the signal every 5 or 10 ms in terms of what they call **mel-scaled cepstral coefficients**. These are DCT coefficients that are derived by applying a discrete cosine transformation to a Mel-scaled spectrum. This point will be explored in more detail at the end of this Chapter.

### 8.4.1 Calculating DCT-coefficients in EMU-R

In order to emphasise the very important point that the DCT is a transformation that is not specific to speech, the first example will be of a DCT applied to some of the data in bridge (a hypothetical count of cars crossing a bridge between midday and midnight). In the example below, the DCT is applied to the data in column 1 which is initially stored (for convenience) in a vector x. The function for calculating DCT

coefficients is the Emu-R function dct(). (Formulae for the DCT are not given here, but see Harrington et al, 2008, Harrington, 2009; Nossair & Zahorian, 1991; Watson & Harrington, 1999).

```
x = bridge[,1]
# DCT coefficients
x.dct = dct(x)
# round to 2 places for convenience
round(x.dct, 2)
```

```
     0      1      2      3      4      5      6      7      8      9     10
 54.39  29.54 -26.13 -24.65  -8.77  -2.96  -0.58  -0.06   0.59   2.14  -1.75
    11     12
 -3.31   3.26
```

x.dct contains the DCT-coefficients: remember that these are amplitudes of ½-cycle cosine waves. The cosine waves into which the signal has been decomposed using the DCT can be inspected using this function:

```
cfun <- function(A, j=0)
{
# A: DCT-coefficients (amplitude of ½ cycle cosine wave)
# j: frequency (cycles) of ½ cycle cosine wave
N = length(A)
n = 0:(N-1)
A[1] = A[1]/sqrt(2)
k = seq(0, by=.5, length=N)
# the cosine wave corresponding to DCT-j
A[j+1] * cos(2 * pi * k[j+1] * (n+0.5)/N)
}
```

Here is a plot of the first four cosine waves i.e., of the cosine waves corresponding to DCT-0, DCT-1, DCT-2, DCT-3 (Fig. 8.21):

```
par(mfrow=c(2,2)); par(mar=rep(2,4))
for(j in 0:3){
plot(cfun(x.dct, j), type="l", xlab="", ylab="", main=paste("DCT-", j, sep=""))
}
```

DCT-0 has a frequency of 0 cycles and so, for the reasons discussed earlier (Fig. 8.2), it is a straight line. Also its amplitude is equal to the mean of the signal to which the DCT was applied. The figure also shows that cosine waves are produced at frequencies of ½, 1, and 1½ cycles for the next higher coefficients respectively (the reason why DCT-2 and DCT-3 are upside-down cosine waves is because these DCT-coefficients are negative). Notice that the peak amplitudes of these cosine waves are equal to the corresponding DCT-coefficients (see round(x.dct, 2) given above). This is to be expected since DCT-coefficients are just the (peak) amplitudes of these cosine waves.

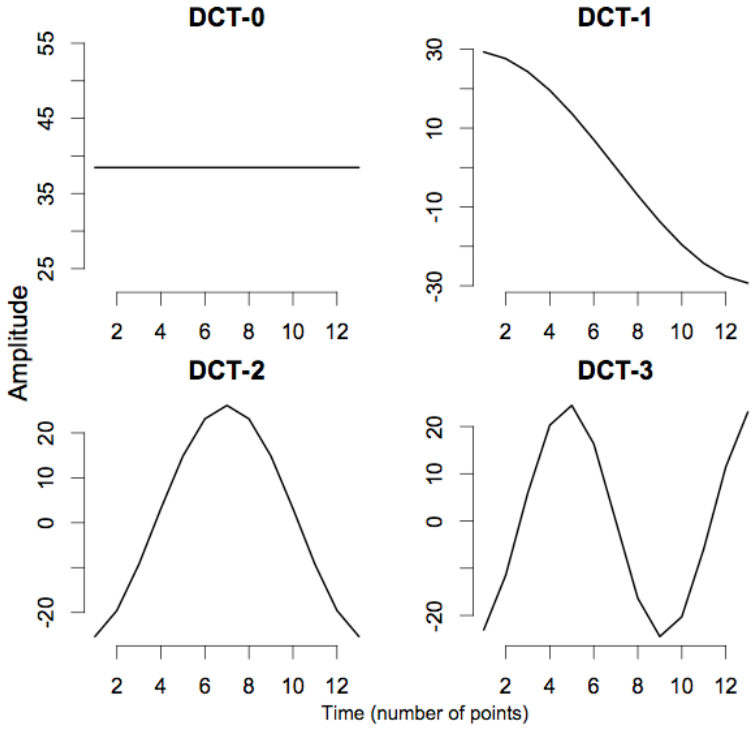



Fig. 8.20: The first four half-cycle cosine waves that are the result of applying a DCT to the signal with data points shown on the right

If *all* of these half-cycle cosine waves are summed, then the result is the original signal to which the DCT transformation has just been applied:

```
N = length(x.dct)
mat = rep(0, length(x.dct))
for(j in 0:(N-1)){
mat = mat+cfun(x.dct, j)
}
```

Apart from rounding errors, mat, the reconstructed signal, is the same as the original signal x, as the following shows:

```
round(mat-x, 5)
0  1  2  3  4  5  6  7  8  9 10 11 12
0  0  0  0  0  0  0  0  0  0  0  0  0
```

Following the reasoning in 8.4.1 above, if only the few *lowest* frequencies are summed, e.g., the cosine waves corresponding to DCT-0 through to DCT-3 shown in Fig. 8.20, then the result is a **DCT-smoothed signal**, i.e. a smoother version of the original signal. The summation could be accomplished with the for-loop given above. Alternatively, and more conveniently, the same dct() function can be used not just for dct-*analysis* but also for DCT-*synthesis* to add up the cosine waves into which the signal was decomposed. The following adds up the first four half-cycle cosine waves shown in Fig. 8.20:

```
# Sum to DCT-3
dctsum = dct(x, 3, T)

# The above is the same as:
dctsum= rep(0, length(x.dct))
for(j in 0:3){
dctsum = dctsum+cfun(x.dct, j)
}
```

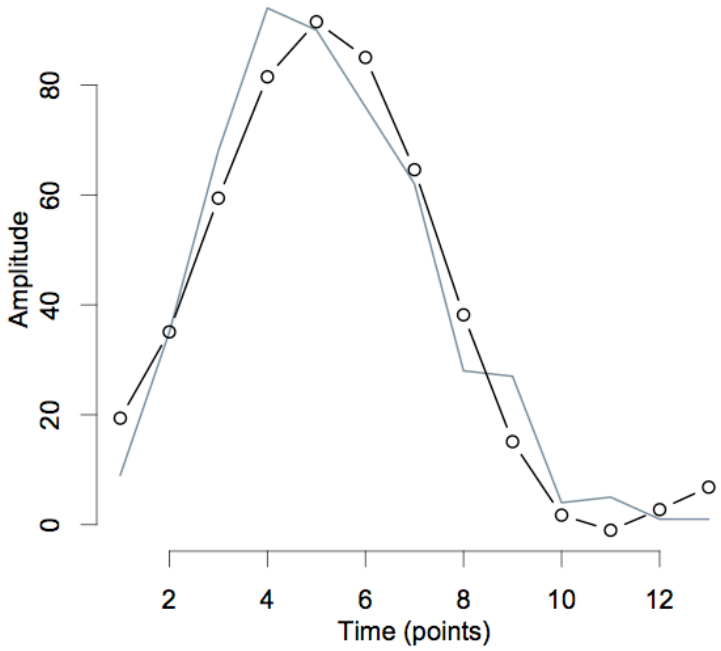



Fig. 8.21: The raw signal (gray) and a superimposed DCT-smoothed signal (black showing data points) obtained by summing the four ½ cycle cosine waves shown on the left.

A plot of the original signal and DCT-smoothed signal using coefficients 0-3 is shown in Fig. 8.21 and is given by the following commands:

```
ylim = range(c(dctsum, x))
plot(x, type="l", ylim=ylim, xlab="Time (points)", ylab="", col="slategray", axes=F)
par(new=T)
plot(dctsum, type="b", ylim=ylim, xlab="Time (points)", ylab="Amplitude")
```

The more coefficients that are used, the closer the DCT-smoothed signal approximates the original signal.

### 8.4.2 DCT-coefficients of a spectrum

The leftmost panel of Fig. 8.22 shows a 512-point dB-spectrum calculated at the temporal midpoint of an [ɛ] vowel sampled at 16000 Hz (the midpoint of the first segment in vowlax as it happens) and plotted with plot(e.dft, type="l"). Following the discussion earlier in this Chapter, the 512 point window is easily wide enough so that harmonics appear: there is a gradual rise and fall due to the presence of formants and

superimposed on this is a jaggedness produced by the fundamental frequency and its associated harmonics. The DCT-coefficients of this spectrum can be calculated following the procedure in 8.4.1. Such a calculation produces the amplitudes of the ½ cycle cosine waves and a plot of them as a function of the corresponding DCT-coefficient number (middle panel, Fig. 8.22) is a cepstrum:

```
# DCT coefficients
e.dct = dct(e.dft)
N = length(e.dct); k = 0:(N-1)
# cepstrum
plot(k, e.dct,  ylim=c(-5, 5), type="l", xlab="Time (number of points",
ylab="Amplitude of cosine waves")
```

In the earlier example of trying to draw an arc while driving over a cattle-grid, it was argued that the deviations caused by the bumps show up at high frequency cosine waves and that analogously so would the oscillations due to the harmonics caused by vocal fold vibration (the source) that produce the jaggedness in a spectrum. In the present example, their effect is visible as the pronounced spike in the cepstrum between 100 and 150 points. The spike occurs at the 107$^{th}$ DCT-coefficient (DCT-107). With this information, the fundamental frequency of the signal can be estimated: 107 points corresponds to 0.0066875 s at the sampling frequency of 16000 Hz and therefore to a fundamental frequency of 1/0.0066875 = 149.5 Hz. The estimated f0 can be checked against the spectrum. For example, the 4$^{th}$ harmonic in the spectrum in the left panel of Fig.8.22 is associated with a peak at 593.75 Hz which means that the fundamental frequency is 593.75/4 = 148.4 Hz, i.e., a value that is within about 1 Hz of the f0 estimated from the cepstrum. So this demonstrates another use of DCT (cepstral) analysis: it can be used to estimate whether or not the signal is voiced (whether there is/is not a spike) and also for estimating the signal's fundamental frequency.

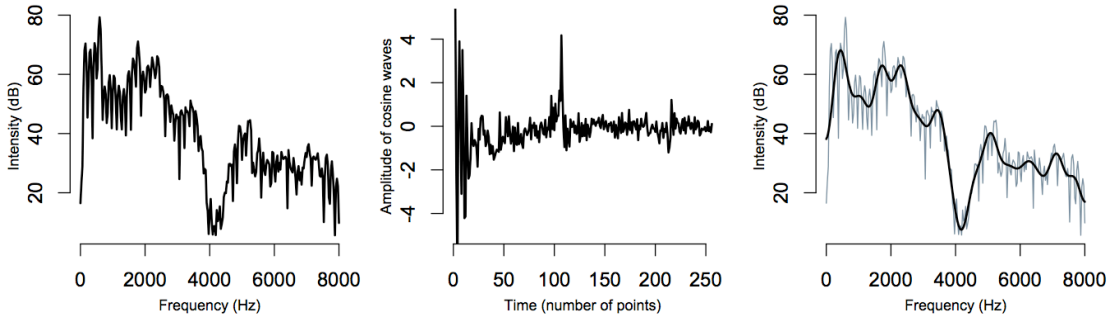



Fig. 8.22: Left: a spectrum of an [ɛ] vowel. Middle the output of a DCT-transformation of this signal (a cepstrum). Right: a DCT-smoothed signal (cepstrally smoothed spectrum) superimposed on the original spectrum in the left panel and obtained by summing the first 31 half-cycle cosine waves.

A DCT or cepstrally-smoothed version of the spectrum that excludes the contribution from the source signal can be obtained as long as the summation does not include the higher frequency cosine waves around DCT-107 that encode the information about the the fundamental frequency and harmonics. Beyond this, there can be no guidelines about how many cosine waves should be summed: the more that are summed, the more the resulting signal approximates the original spectrum. In the right

panel of Fig. 8.22, the first 30 coefficients have been summed and the result superimposed on the original raw spectrum as follows:

```
# Carry out DCT analysis then sum from DCT-0 to DCT-30
coeffto30 = dct(e.dft, 30, T)
# We have to tell R that this is spectral data at a sampling frequency of 16000 Hz
coeffto30 = as.spectral(coeffto30, 16000)
ylim = range(coeffto30, e.dft)
# Raw dB-spectrum
plot(e.dft, ylim=ylim, xlab="", ylab="", axes=F, col="slategray", type="l")
par(new=T)
# Superimposed DCT-smoothed (cepstrally-smoothed) spectrum
plot(coeffto30, ylim=ylim, xlab="Frequency (Hz)", ylab="Intensity (dB) ")
```

The smooth line through the spectrum, a cepstrally-smoothed spectrum, has none of the influence due to the source. Finally, if you want to derive a cepstrally smoothed spectra from either a spectral matrix or trackdata object[4], then this can be done using dct() with the argument fit=T inside fapply(). For example, a plot of cepstrally smoothed spectra using 5 coefficients for a spectral matrix of stop bursts is given by:

```
smooth = fapply(keng.dft.5, dct, 5, fit=T)
plot(smooth, keng.l)
```

### 8.4.3 DCT-coefficients and trajectory shape

The lowest three DCT-coefficients are, as has already been mentioned, related to the mean, slope, and curvature respectively of the signal to which the DCT transformation is applied. DCT-0 in the DCT-algorithm that is implemented here (and discussed in Watson & Harrington, 1999) is the mean of the signal multiplied by $\sqrt{2}$. DCT-1 is directly proportional to the linear slope of the signal. This relationship can be verified by calculating the linear slope using the slope() function created in 8.2 and then correlating the slope with DCT-1. For example, for the dorsal fricative data:

```
# Here is the slope function again:
slope <- function(x)
{
# Calculate the intercept and slope  in a spectral vector
lm(x ~ trackfreq(x))$coeff
}

# Spectra at the temporal midpoint
dorfric.dft.5 = dcut(dorfric.dft, .5, prop=T)
# Spectral slope – NB the slope is stored in column 2
sp = fapply(dorfric.dft.5, slope)
# Coefficients up to DCT-1 – NB DCT-1 is in column 2
d = fapply(dorfric.dft.5, dct, 1)
# How strongly is the linear slope correlated with DCT-1?
```

[4] Calculating cepstrally smoothed spectra on a spectral trackdata object is very slow in R. There is a signal processing routine in Emu-tkassp for calculating cepstrally smoothed spectra directly from the audio signals (LP type CSS in the spectrum pane).

```
cor(sp[,2], d[,2])
-0.9979162
```

The above shows that there is almost complete (negative) correlation between these variables, i.e. greater positive slopes correspond to greater negative DCT-1 values and *vice-versa* (this is clearly seen in plot(sp[,2], d[,2]) where you can also see that when the linear slope is zero, so is DCT-1).

DCT-2 is most closely related to the signal's curvature, where curvature has the definition given in 8.6 of the chapter on vowels, i.e., it is the coefficient $c_2$ in a parabola $y = c_0 + c_1 x + c_2 x^2$. Recall that the coefficient $c_2$ can be calculated as follows:

```
# c2 for F1 data, lax vowels. c2 is stored in coeffs[,3]
coeffs= trapply(vowlax.fdat[,1], plafit, simplify=T)
```

The DCT-coefficients: DCT-2 is stored in d[,3]

```
d = trapply(vowlax.fdat[,1], dct, 3, simplify=T)
```

The correlation between $c_2$ and DCT-2 is very high:

```
cor(coeffs[,3], d[,3])
[1] 0.939339
```

In general, there will only be such a direct correspondence between curvature in a parabola and DCT-2 as long as the signal has a basic parabolic shape – if not, then the relationship between the two is likely to be a good deal weaker.

### 8.4.4 Mel- and Bark-scaled DCT (cepstral) coefficients

The Bark scale has already been discussed in the chapter on vowels: it is a scale that warps the physical frequency axis in Hz into one which corresponds more closely to the way in which frequency is processed in the ear. Another auditory scale that was more commonly used in phonetics in the 1970s and which is used in automatic speech recognition research today is the Mel scale. As discussed in Fant (1968), the Mel scale is obtained in such a way that a doubling on the Mel scale corresponds roughly to a doubling of perceived pitch. Also, 1000 Mel = 1000 Hz. If you want to see the relationship between Mel and Hz, then enter:

```
plot(0:10000, mel(0:10000), type="l", xlab="Frequency (Hz)", ylab="Frequency
(mels)")
```

In fact, the Bark and Mel scale warp the frequency scale in rather a similar way, especially for frequencies above about 1000 Hz.

There are two main ways to see what a spectrum looks like when its frequency axis is converted to an auditory scale. The first just converts trackfreq(x) from Hz into Mel or Bark (where x is a spectral object). Since the auditory scales are approximately linear up to 1000 Hz and quasi-logarithmic thereafter, the result of the first method is that there are more data points at higher frequencies in the auditorily-scaled spectra: this is because the interval in Bark or Mel for the same frequency width in Hz becomes progressively smaller with increasing frequency (compare for example the difference in Bark between 7000 Hz and 6000 Hz given by bark(7000) - bark(6000) with the Bark difference between 2000 Hz and 1000 Hz ). The second method uses a linear interpolation technique (see 6.6 and Fig. 6.18) so that the data points in the spectrum

are spaced at equal Mel or Bark intervals along the frequency axis: so with this second method, there is the same number of data points between 1 and 2 Bark as between 3 and 4 Bark and so on. Both methods give more or less the same spectral shape, but obviously some of the detail is lost in the high frequency range with the second method because there are fewer data points. Here finally are the two methods for the spectrum of the [ɛ] vowel considered earlier:

```
# Method 1
plot(e.dft, freq=bark(trackfreq(e.dft)), type="l", xlab="Frequency (Bark)")
# Method 2
plot(bark(e.dft), type="l", xlab="Frequency (Bark)")
```

A Bark-scaled DCT-transformation is just a DCT transformation that is applied to a spectrum after the spectrum's frequency axis has been converted into Bark (or into Mel for a Mel-scaled DCT-transformation). Only the second method, in which the data points represent equal intervals of frequency is available for DCT-analysis, and not the first[5]. This is because the DCT-analysis is predicated on the assumption that the digital points are at equal intervals (of time or of frequency).

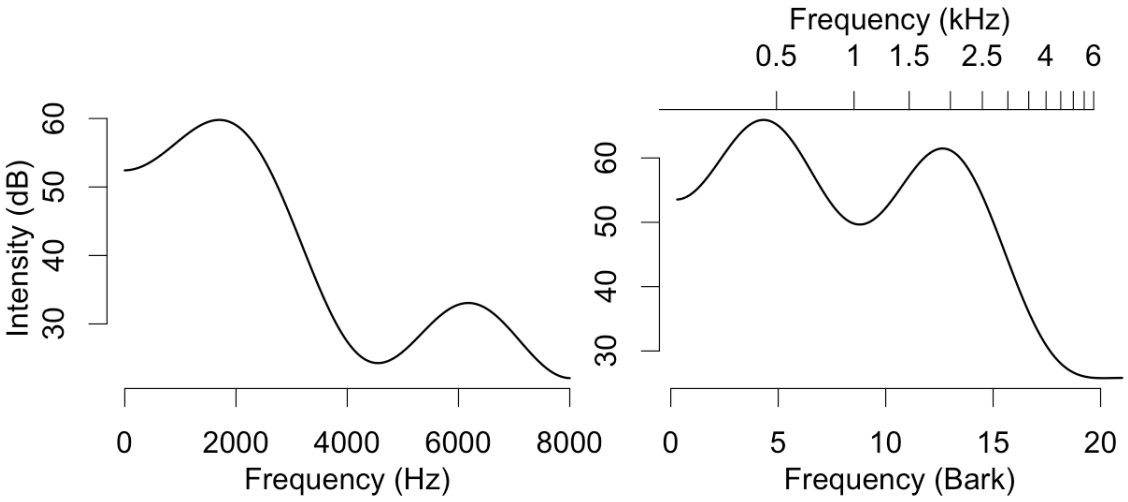



Fig. 8.23: Left: a DCT-smoothed Hz spectrum of [ɛ]. Right: A DCT-smoothed, Bark-scaled spectrum of the same vowel. Both spectra were obtained by summing the first six coefficients, up to DCT-5. For the spectrum on the right, the frequency axis was converted to Bark with linear interpolation before applying the DCT.

The motivation for converting to an auditory scale is not just that this scale is obviously more closely related to the way in which frequency is perceived, but also because, as various studies in automatic speech recognition have shown, fewer Bark- or Mel-scaled DCT (cepstral) coefficients are needed to distinguish effectively between different phonetic categories than when DCT coefficients are derived from a Hz scale. In order to illustrate this point, calculate a DCT-smoothed spectrum with and without

[5] In fact, there is a third method. In automatic speech recognition, energy values are often summed in a filter-bank at intervals of 1 Bark. This reduces a spectrum to a series of about 20 values, and it is this data-reduced form that is then subjected to a DCT (e.g. Milner & Shao, 2006). The filter bank approach is discussed in Chapter 9.

auditory scaling using only a small number of coefficients (six in this example, up to DCT-5), as follows:

```
# DCT (cepstrally) smoothed Hz spectrum with 6 coefficients
hz.dft = dct(e.dft, 5, T)
hz.dft = as.spectral(hz.dft, trackfreq(e.dft))
# DCT (cepstrally) smoothed Bark  spectrum with 6 coefficients
bk.dft = dct(bark(e.dft), 5, T)
bk.dft = as.spectral(bk.dft, trackfreq(bark(e.dft)))
# Plots
par(mfrow=c(1,2))
plot(hz.dft, xlab="Frequency (Hz)", ylab="Intensity (dB)")
plot(bk.dft, xlab="Frequency (Bark)")
# Superimpose a kHz axis up to 3.5 kHz
values = seq(0, 6000, by=500)
axis(side=3, at=bark(values), labels=as.character(values/1000))
mtext("Frequency (kHz)", side=3, line=2)
```

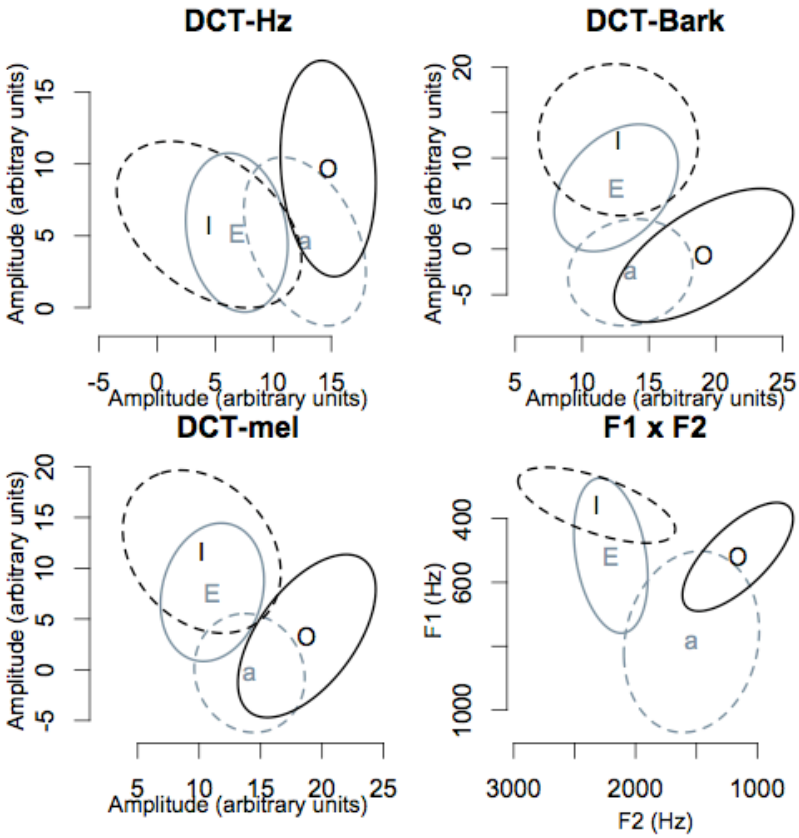



Fig. 8.24:  95% confidence ellipses around the data at the temporal midpoint of German lax vowels produced  by a female speaker.  Top left: DCT 1 x DCT-2 derived from Hz-spectra. Top right: DCT-1 x DCT-2 derived from Bark-spectra. Bottom left: DCT-1 x DCT-2 derived from mel-spectra. Bottom right: F1 x F2.

The DCT-smoothed Hz spectrum (left panel, Fig. 8.23) is too smooth: above all it does not allow the most important information that characterises an [ɛ] vowel, i.e. F1 and F2 to be distinguished. The DCT-smoothed Bark spectrum seems to be  as smooth and is

perhaps therefore just as ineffective as the Hz spectrum for characterising the salient acoustic properties of [ɛ]. But a closer inspection shows that this is not so. There are evidently two broad peaks in the DCT-smoothed Bark spectrum that are at 4.32 Bark and 12.63 Bark respectively. The conversion bark(c(4.32, 12.64), inv=T), shows that these Bark frequencies are 432 Hz and 1892 Hz – in other words the frequency location of these peaks is strongly influenced by the first two formant frequencies[6]. So the DCT-smoothed Bark-spectrum, in contrast to the DCT-smoothed Hz-spectrum, seems to have given greater prominence to just those attributes of [ɛ] that are most important for identifying it phonetically.

A comparison can now be made of how the raw and auditorily-transformed DCT-coefficients distinguish between the same German lax vowel categories that were the subject of analysis in Chapter 6. For this purpose, there is a spectral object vowlax.dft.5 which contains 256-point dB-spectra at the temporal midpoint of the segment list vowlax. The relevant objects for the present investigation include:

```
vowlax.dft.5        # Matrix of dB-spectra
vowlax.l            # Vector of vowel labels
vowlax.spkr         # Vector of speaker labels
vowlax.fdat.5       # F1-F4 formant frequency data at the temporal midpoint
```

Given that the salient acoustic information for distinguishing between vowel categories is typically between 200-4000 Hz, the first few DCT coefficients will be calculated in this frequency range only:

```
# First four DCT-coefficients calculated on Hz spectra
dcthz = fapply(vowlax.dft.5[,200:4000], dct, m=3)
# ...on Bark-scaled spectra
dctbk = fapply(bark(vowlax.dft.5[,200:4000]), dct, m=3)
# on Mel-scaled spectra.
dctml = fapply(mel(vowlax.dft.5[,200:4000]), dct, m=3)
```

Remember that at least 6 or 7 auditorily scaled DCT-coefficients are usually necessary to obtain a discrimination between vowel categories that is as effective as the one from the first two formant frequencies. Nevertheless, there is a reasonably good separation between the vowels for female speaker 68 in the plane of DCT-1 x DCT-2 (the reader can experiment with other coefficients pairs and at the same time verify that the separation is not as good for the male speaker's data on these coefficients). The same vowels in the formant plane are shown for comparison bottom right in Fig. 8.24.

```
temp = vowlax.spkr == "68"
par(mfrow=c(2,2))
eplot(dcthz[temp,2:3], vowlax.l[temp], centroid=T, main="DCT-Hz")
eplot(dctbk[temp,2:3], vowlax.l[temp], centroid=T, main="DCT-Bark")
eplot(dctml[temp,2:3], vowlax.l[temp], centroid=T, main="DCT-mel")
eplot(dcut(vowlax.fdat[temp,1:2], .5, prop=T), vowlax.l[temp], centroid=T, form=T,
main="F1 x F2")
```

[6] Recall that this spectrum was taken at the onset of the first vowel in vowlax. The peaks in the Bark-smoothed cepstrum on the right of Fig. 8.23 are quite close to the F1 and F2 in the first few frames of the calculated formants for this vowel given by frames(vowlax.fdat[1,1:2]).

There are a couple of interesting things about the data in Fig. 8.24. The first is that in all of the DCT-spaces, there is a resemblance to the shape of the vowel quadrilateral, with the vowel categories distributed in relation to each other very roughly as they are in the formant plane. This is perhaps not surprising given the following three connected facts:

- a DCT transformation encodes the overall shape of the spectrum
- The overall spectral shape for vowels is predominantly determined by F1-F3
- F1 x F2 are proportional to phonetic height and backness respectively, and therefore to the axes of a vowel quadrilateral.

Secondly, the vowel categories are distinguished to a slightly greater extent in the auditorily transformed DCT-spaces (Bark and Mel) than in the DCT-Hertz spaces - this is especially so as far as the overlap of [a] with [ɪ, ɛ] is concerned.

Finally, one of the advantages of the DCT over the formant analysis is that there has been no need to use complicated formant tracking algorithms and above all no need to make any corrections for outliers. This is one of the reasons why they are preferred in automatic speech recognition. Another is that, while it makes no sense to track formants for voiceless sounds, the same DCT coefficients, or auditorily-transformed DCT coefficients, can be used for quantifying both voiced and voiceless speech.

## 8.5. Questions

1. This question is about digital sinusoids.

1.1. Use the crplot() function in the Emu-R library to plot the alias of the cosine wave of length 20 points and with frequency 4 cycles.

1.2. Use the crplot() function to plot a sine wave.

1.3. The alias also requires the phase to be opposite in sign compared with the non-aliased waveform. Use crplot() to plot the alias of the above sine wave.

1.4. The cr() function produces a plot of $A\cos(\frac{2\pi kn}{N}+\phi)$ where $A$, $k$, $\phi$ are the cosine's amplitude, frequency (in cycles) and phase (in radians) respectively. Also, $N$ is the length of the signal and $n$ is a vector of integers, 0, 1, 2, … $N$-1. Convert the equation into an R function that takes $A$, $k$, $f$, $N$ as its arguments, and verify that you get the same results as from cr() for any choice of amplitude, frequency, phase, and $N$. (Plot the cosine wave from your function against $n$ on the $x$-axis).

1.5. What is the effect of adding to a cosine wave another cosine wave that has been phase-shifted by π radians (180 degrees)? Use the cr() function with values=T (and round the result to the nearest 4 places) to check your assumptions.

2. According to Halle, Hughes & Radley (1957), the two major allophones of /k/ before front and back vowels can be distinguished by $a - b$ , where $a$ and $b$ have the following definitions:

$a$ the sum of the dB-values in the 700 Hz – 9000 Hz range.

*b* the sum of the dB-values in the 2700 – 9000 Hz range.

Verify (using e.g., a boxplot) whether this is so for the following data:

| | |
|---|---|
| keng | Segment list of the aspiration of syllable-initial Australian English /k/ before front /ɪ,ɛ/ (e.g., *kin*, *kept*) and back /ɔ:, ʊ/ vowels (e.g., *caught*, *could*). |
| keng.dft.5 | Spectral matrix of the above at the temporal midpoint of the segment. |
| keng.l | Labels of the following vowel (`front` or `back`) |

3. If vowel lip-rounding has an anticipatory coarticulatory influence on a preceding consonant in a CV sequence, how would you expect the spectra of alveolar fricatives to differ preceding unrounded and rounded vowels? Plot the spectra of the German syllable-initial [z] fricatives defined below at their temporal midpoint separately in the unrounded and rounded contexts to check your predictions.

| | |
|---|---|
| sib | Segment list, syllable-initial [z] preceding [i:, ɪ, u:, ʊ], one male and one female speaker. |
| sib.l | A vector of labels: `f`, for [z] preceding front unrounded [i:, ɪ], `b` for [z] preceding back rounded [u:, ʊ] |
| sib.w | A vector of word labels. |
| sib.dft | Spectral trackdata object (256 point DFT) from the onset to the offset of [z] with a frame shift of 5 ms. |

Apply a metric to the spectra that you have just plotted to see how effectively you can distinguish between [z] before unrounded and rounded vowels.

4. Here are some F2-data of Australian English and Standard German [i:] vowels, both produced in read sentences each by one male speaker.

| | |
|---|---|
| f2geraus | Trackdata object of F2 |
| f2geraus.l | Vector of labels: either `aus` or `ger` corresponding to whether the F2-trajectories in f2geraus were produced by the Australian or German speaker. |

4.1 It is sometimes said that Australian English [i:] has a 'late target' (long onglide). How are the trajectories between the languages likely to differ on skew?

4.2 Produce a time-normalised, averaged plot of F2 colour-coded for the language to check your predictions.

4.3 Quantify these predictions by calculating moments for these F2 trajectories (and e.g., making a boxplot of skew for the two language categories).

5. Sketch (by hand) the likely F2-trajectories of [aɪ, aʊ, a] as a function of time. How are these F2-trajectories likely to differ on skew? Check your predictions by calculating F2-moments for [aɪ, aʊ] and [a] for speaker 68. Use the following objects:

| | |
|---|---|
| dip.fdat | Trackdata object of formants containing the diphthongs |

| | |
|---|---|
| dip.l | Vector of diphthong labels |
| dip.spkr | Vector of speaker labels for the diphthongs |
| vowlax.fdat | Trackdata object of formants containing [a] |
| vowlax.l | Vector of vowel labels |
| vowlax.spkr | Vector of speaker labels |

Make a boxplot showing the skew for these three categories.

6. The features *diffuse* vs. compact are sometimes used to distinguish between sounds whose energy is more distributed (diffuse) as opposed to concentrated predominantly in one region (compact) in the spectrum.

6.1 On which of the moment parameters might diffuse vs. compact spectra be expected to differ?

6.2 In their analysis of stops, Blumstein & Stevens (1979) characterise (the burst of) velars as having a *compact* spectrum with mid-frequency peaks as opposed to labials and alveolars for which the spectra are *diffuse* in the frequency range 0-4000 Hz. Check whether there is any evidence for this by plotting ensemble-averaged spectra of the bursts of [b,d,ɡ] overlaid on the same plot (in the manner of Fig. 8.10, right). All of the data is contained in a spectral matrix calculated from a 256-point DFT centered 10 ms after the stop release and includes the same [b,d] spectral data as after derived in 8.2 as well as [ɡ]-bursts before the non-back vowels [i:, e:, a:, aʊ].

| | |
|---|---|
| stops10 | spectral matrix, 256-point DFT |
| stops10.lab | vector of stop labels |

6.3 Calculate in the 0-4000 Hz range whichever moment you think might be appropriate for distinguishing [ɡ] from the other two stop classes and make a boxplot of the chosen moment parameter separately for the three classes. Is there any evidence for the diffuse ([b,d]) vs. compact ([ɡ]) distinction?

7. A tense vowel is often phonetically more peripheral than a lax vowel, and acoustically this can sometimes be associated with a greater formant curvature (because there is often a greater deviation in producing a tense vowel from the centre of the vowel space).

7.1 Verifying whether there is any evidence for this using dplot() to produce time-normalised, ensemble-averaged F2-trajectories as a function of time of the German tense and lax [i:, ɪ] vowels produced by male speaker 67. The data to be plotted is from a trackdata object dat with a parallel vector of labels lab that can be created as follows:

```
temp = f2geraus.l == "ger"
# F2-trackdata of tense i:
dati = f2geraus[temp,]

# A parallel vector of labels
labi = rep("i:", sum(temp))

temp = vowlax.l == "I" & vowlax.spkr == "67"
```

```
# F2-trackdata of lax [ɪ]
datI = vowlax.fdat[temp,2]
# A parallel vector of labels
labI = rep("I", sum(temp))

# Here are the data and corresponding labels to be plotted
dat = rbind(dati, datI)
lab = c(labi, labI)
```

7.2. Quantify the data by calculating DCT-2 and displaying the results in a boxplot separately for [i:] and [ɪ].

8. This question is concerned with the vowels [ɪ, ʊ, a] in the `timetable` database. The following objects are available from this database in the Emu-R library:

```
timevow                 # Segment list of these three vowels
timevow.dft             # Spectral trackdata object of spectra between
                        the start and end times of these vowels
timevow.l               # Vector of labels
```

8.1 Make an ensemble-averaged spectral plot in the 0-3000 Hz range (with one average per vowel category) of spectra extracted at the temporal midpoint of these vowels. Produce the plot with the *x*-axis proportional to the Bark scale. Look at the global shape of the spectra and try to make predictions about how the three vowel categories are likely to differ on Bark-scaled DCT-1 and DCT-2.

8.2 Calculate Bark-scaled DCT-1 and DCT-2 for these spectra and make ellipse plots of the vowels in this plane. To what extent are your predictions in 9.1 supported?

8.3 Produce for the first [ɪ] at its temporal midpoint a Bark-spectrum in the 0-4000 Hz range overlaid with a smoothed spectrum calculated from the first 6 Bark-scaled DCT-coefficients. Produce the plot with the axis proportional to the Bark scale.

## 8.6 Answers

1.1.

```
crplot(k=16, N=20)
```

1.2.

```
crplot(p=-pi/2)
```

1.3.

```
crplot(k=15, p=pi/2)
```

1.4.

```
cfun <- function(A, k, p, N)
{
n = 0:(N-1)
A * cos((2 * pi * k * n)/N + p)
}
```

For example:

```
res = cfun(1.5, 1.4, pi/3, 20)
n = 0:19
plot(n, res, col=2)
par(new=T)
cr(1.5, 1.4, pi/3, 20, type="l")
```

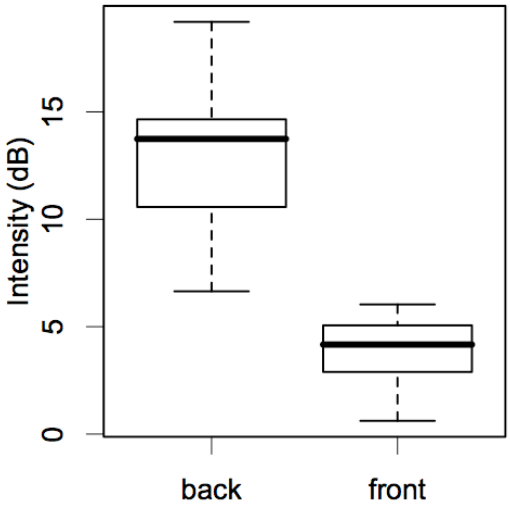



Fig. 8.25: The difference in energy between two frequency bands calculated in the burst of back and front allophones of /k/.

1.5. The sinusoids cancel each other out.

```
o = cr(p=c(0, pi), values=T)
round(o, 4)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

2.

```
a = fapply(keng.dft.5[,700:9000], sum, power=T)
b = fapply(keng.dft.5[,2700:9000], sum, power=T)
d = a - b
boxplot(d ~ factor(keng.l), ylab="dB-difference")
```

Yes, this parameter separates the front and back allophones of /k/ quite well (Fig. 8.25).

3. You would expect an overall lowering of the spectral energy, i.e. lip-rounding should produce a decrease in the spectral centre of gravity or first spectral moment. See e.g., Mann & Repp (1980).

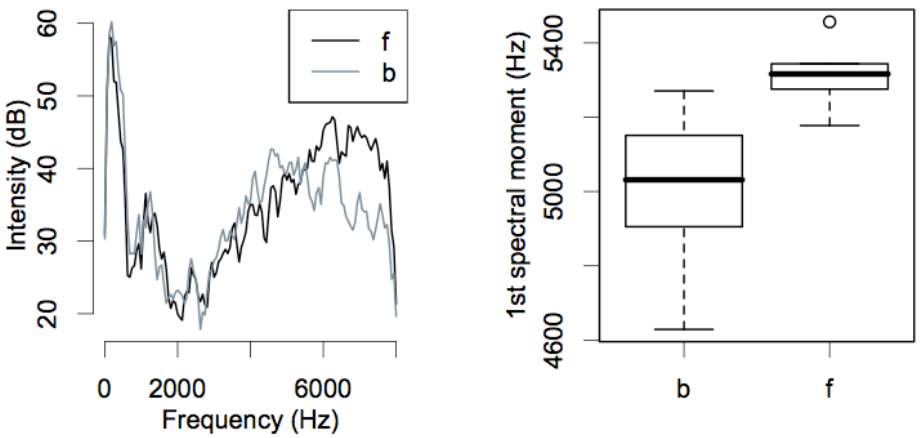



Fig. 8.26: Left: Averaged spectra of [z] preceding front unrounded (`f`) and back rounded (`b`) vowels. Right: boxplot of the first spectral moment of the same data calculated in the 2000-7700 Hz range.

```
sib.mid= dcut(sib.dft, .5, prop=T)
plot(sib.mid, sib.l, fun=mean, power=T, xlab="Frequency (Hz)", ylab="Intensity (dB)")
```

The parameter chosen here is the 1st spectral moment in the 2000-7700 Hz range.

```
m = fapply(sib.mid[,2000:7700], moments, minval=T)
boxplot(m[,1] ~ factor(sib.l), ylab="1st spectral moment (Hz)")
```

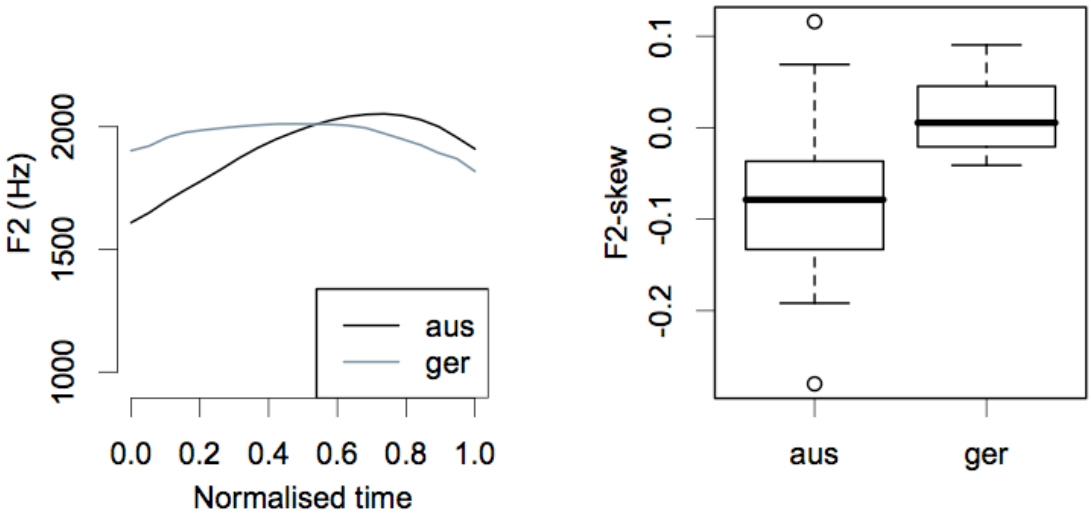



Fig. 8.27: Left: averaged, time-normalised plots of F2 as a function of time for Australian English (black) and Standard German (gray) vowels. Right: boxplots of the 3rd spectral moment calculated across the vowel trajectories from their acoustic onset to their acoustic offset.

4.1 The F2-trajectories for late target vowels should be skewed to the right.

4.2 The plots are in Fig. 8.27.

```
par(mfrow=c(1,2))
dplot(f2geraus, f2geraus.l, average=T, normalise=T, xlab= "Normalised time",
ylab="F2 (Hz)", leg="bottomright")
```

4.3

```
m = trapply(f2geraus, moments, simplify=T)
skew =m[,3]
boxplot(skew ~ factor(f2geraus.l), ylab="F2-skew")
```

5. [aɪ] has a late F2 peak, [aʊ] has a late F2-trough (and thus effectively an early peak), and [a] has its F2 peak near the temporal midpoint. The skew for these three categories should therefore be negative, positive, and close to zero respectively (Fig. 8.28).

```
temp1 = dip.l %in% c("aI", "aU") & dip.spkr == "68"
temp2 = vowlax.l == "a" & vowlax.spkr == "68"
dat = rbind(dip.fdat[temp1,2], vowlax.fdat[temp2,2])
labs = c(dip.l[temp1], vowlax.l[temp2])
coeffs = trapply(dat, moments, simplify=T)
# Fig. 8.28
boxplot(coeffs[,3] ~ labs, ylab="F2-skew")
```

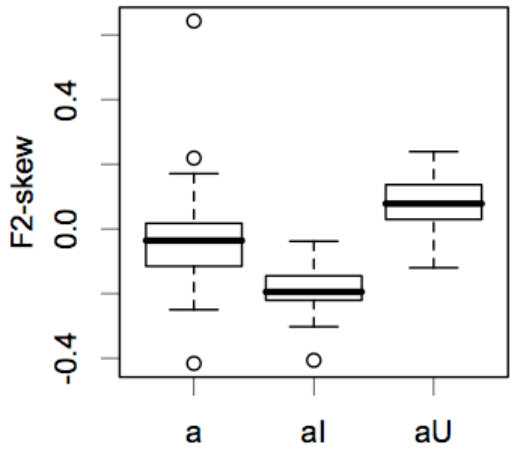



Fig. 8.28: Boxplot of third moment calculated across F2 trajectories of one female speaker separately for two diphthongs and a monophthong.

6.1 Diffuse and compact should differ on the 2nd moment (i.e., variance).

6.2

```
plot(stops10[,0:4000], stops10.lab, fun=mean, power=T, xlab="Frequency (Hz)",
ylab="Intensity (dB)", leg="bottomleft")
```

The averaged spectra in Fig. 8.29 do indeed seem to show a greater concentration of energy in the mid frequency range for [g] than for the other stop categories and so the spectral variance should be less.

6.3

```
m = fapply(stops10[,0:4000], moments, minval=T)
boxplot(sqrt(m[,2]) ~ stops10.lab, ylab="Square root of 2nd moment (Hz)")
```

There is some evidence that the spectral variance is less for [g] in the right panel of Fig. 8.29. However, the data for [g] is dispersed as shown by the large inter-quartile range

(the extent of the rectangle) in comparison with [b, d]: this probably comes about because [ɡ] bursts occur before both front and back vowels which, of course, induce a good deal of variation in initial [ɡ].

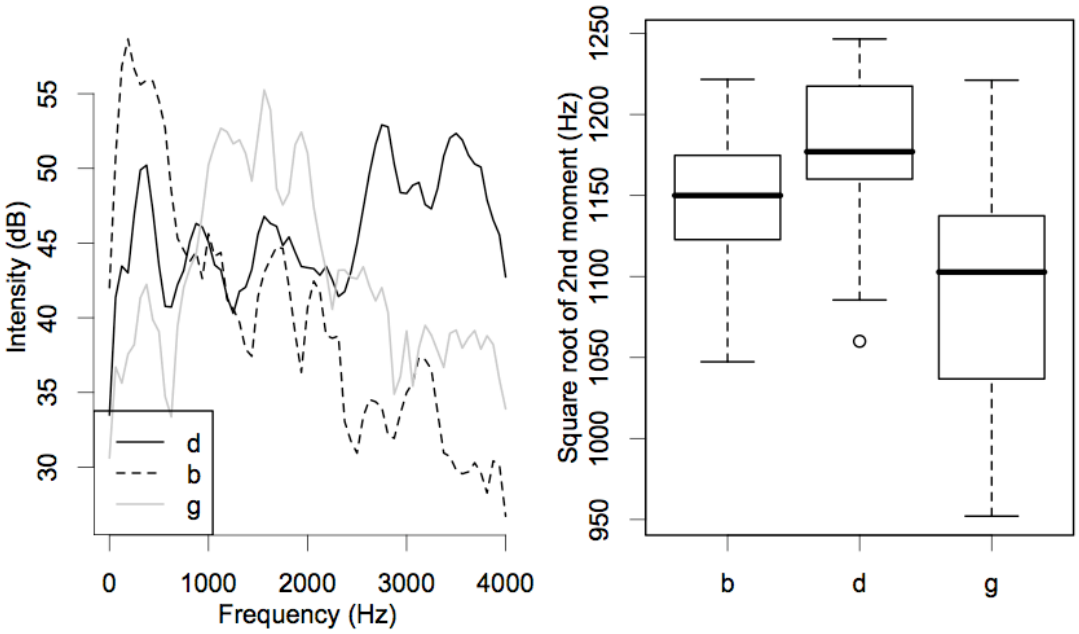



Fig. 8.29: Left: averaged spectra of the bursts of [b, d, ɡ] in isolated words produced by an adult male German speaker. The bursts were calculated with a 256 point DFT (sampling frequency 16000 Hz) centered 10 ms after the stop release. Right: The square root of the second spectral moment for these data calculated in the 0-4000 Hz range.

7.1. There is evidence from Fig. 8.30 (left) for greater curvature in [i:] and this is associated with more extreme DCT-2 values shown on the right.

7.1

```
dplot(dat, lab, ylab="F2 (Hz)", norm=T, average=T, ylim=c(1600, 2100))
```

7.2

```
m = trapply(dat, dct, 2, simplify=T)
boxplot(m[,3] ~ factor(lab), ylab="DCT-2", ylim=c(-150,60))
```

8.1.

```
mid = dcut(timevow.dft, .5, prop=T)
plot(bark(mid[,0:3000]), timevow.l, fun=mean, power=T, xlab="Frequency (Bark)",
ylab="Intensity (dB)"
```

Since the slope of the spectrum for [ʊ] in the left panel of Fig. 8.31 falls more steeply than for the other vowels, it should be distinguished from them on DCT-1. The spectra for [ɪ, a] have a ∪-shape and ∩-shape respectively whereas that of [ʊ] shows less evidence of a parabolic shape. Therefore, DCT-2 should certainly distinguish [ɪ] and [a] (they might have roughly the same value on DCT-2 but one will be positive and the other negative) and both of these should be distinguished from the comparatively more flat [ʊ] spectra.

8.2

```
mid.dct = fapply(bark(mid[,0:3000]), dct, 2)
eplot(mid.dct[,2:3], timevow.l, centroid=T, xlab="Bark-scaled DCT-1", ylab="Bark-scaled DCT-2")
```

The predictions from 8.1 are more or less supported.

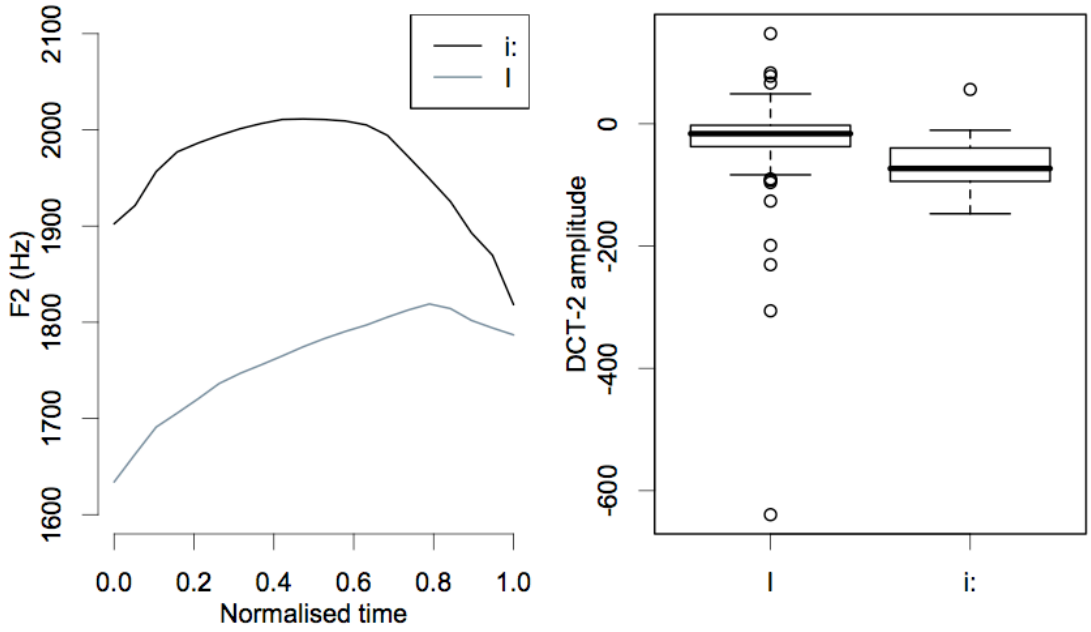



Fig. 8.30: Left: Linearly time-normalised and then averaged F2-trajectories for German [i:] and [ɪ]. Right: DCT-2 amplitudes shown separately for [i:] and [ɪ] calculated by applying a discrete cosine transformation from the onset to the offset of the F2-trajectories.

8.3

```
# Logical vector for [ɪ]
temp = timevow.l == "I"
# n is an index of the first [ɪ]
n = 1:length(timevow.l)
n = n[temp][1]
# Bark spectrum for this [ɪ] at the segment's temporal midpoint
ispec = bark(dcut(timevow.dft[n,0:4000], .5, prop=T))
# Smoothed-Bark spectrum
smooth = dct(ispec, 5, fit=T)
# Equivalently
# smooth = fapply(ispec, dct, 5, fit=T)
# Row bind the raw and smooth spectra
both = rbind(ispec, smooth)
# Make both into a spectral object with the same frequency range
fs = 2 * max(trackfreq(ispec))
both = as.spectral(both, fs)
```

```
# A plot of the raw and overlaid smoothed spectrum
plot(both, c("raw", "smooth"), xlab="Frequency (Bark)", ylab="Intensity (dB)")
```

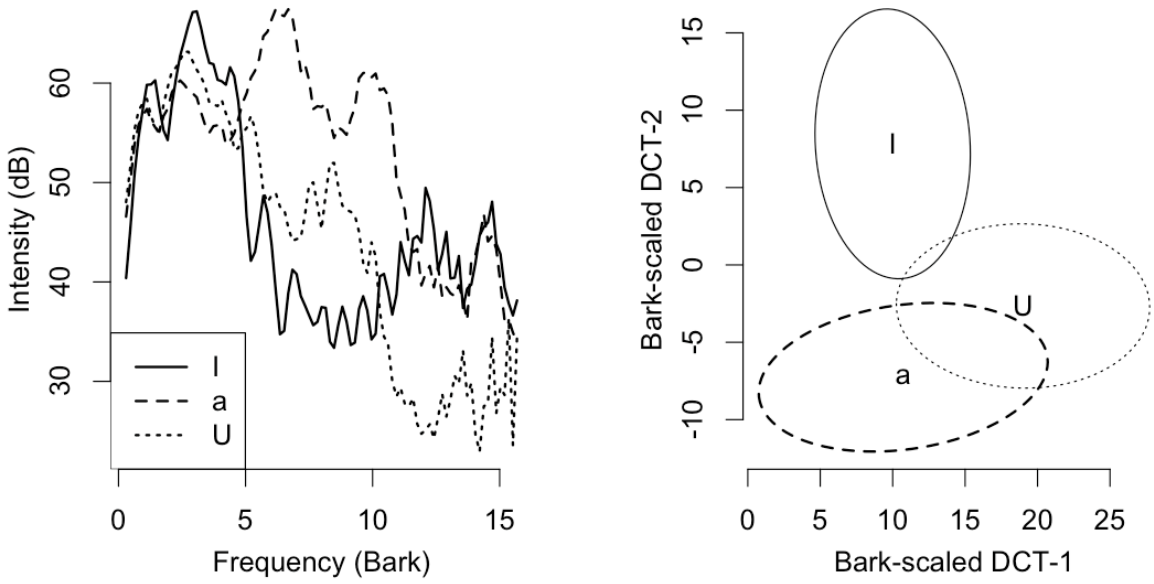



Fig. 8.31. Ensemble-averaged spectra (left) at the temporal midpoint the vowels [ɪ, a, ʊ] (solid, dashed, dotted) and a plot of the same vowels in the plane of Bark-scaled DCT-1 and DCT-2 calculated over the same frequency range.