Use R!

Jérôme Sueur

Sound Analysis and Synthesis with R

Springer

Use R!

Series editors

Robert Gentleman Kurt Hornik Giovanni Parmigiani

More information about this series at http://www.springer.com/series/6991

Jérôme Sueur

Sound Analysis and Synthesis with R

Springer

Jérôme Sueur
Muséum National d'Histoire naturelle
Paris, France

Electronic Supplementary Material The online version of this article (https://doi.org/10.1007/978-3-319-77647-7) contains supplementary material, which is available to authorized users.

ISSN 2197-5736 ISSN 2197-5744 (electronic)
Use R!
ISBN 978-3-319-77645-3 ISBN 978-3-319-77647-7 (eBook)
https://doi.org/10.1007/978-3-319-77647-7

Library of Congress Control Number: 2018939906

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Un livre sans histoires ni paroles
mais écrit avec passion pour

Chloé
Julia
r
o
l
i
n
e

Preface

Sound is virtually always around us, everywhere, all the time. This morning, my day started on a rather unpleasant one: that of the repeated buzzing of my alarm clock. The night had been quiet despite the purr of the central heating, some motorbikes racing down the street, and blackbirds singing in one of our garden's trees at dawn. Now that I go down to the kitchen I can hear my clothes rub against my body and the wooden steps crack under my feet. As soon as I move, I realize that I myself generate sound. Quickly the house wakes up in an explosion of surrounding sounds seeping from the flush, the kettle, the toaster, the fridge, ventilation, and other domestic appliances. Music plays on the radio, but it does not cover the call of a hungry cat and family voices that soon invest and dominate in the acoustic space. My working day is a long, and sometimes exhausting, suite of sounds: metallic train screeches, mobile phone ringtones, office babbles, siren blares, street work roar, and radio tunes but also some amazing tropical sounds I recorded in a remote forest that I play back on my computer to escape that city soundscape. Sound is ubiquitous. It constantly reaches my body, being absorbed or bounced back, received and processed through my ears. But my body is also a sound source. My heart, my blood, my breath, my bones, and my vocal chords generate sound. I am sound in a world of sounds. No air, no life, no sound.

The soundscape I go through any given workday is mainly a city soundscape with very little enjoyable sound. Most of this sound can be considered as noise, which is actually sound conveying no information or overlapping other meaningful sounds. Nice soundscapes are certainly to be found in nature, in the middle of a dark forest or in the depth of an even darker ocean. Wildlife sound can be a bird song, a frog call, an insect hummer, a deer grunting, the exploding sound of a small pistol shrimp, or the amazing whistle of a giant whale. We often refer to the extraordinary diversity of life forms and colours, but life diversity is also to be found in animal vocalizations. Animals can produce rhythmic or continuous, pure-tone or polyphonic, harmonic or dysharmornic, synchronized or cacophonic sounds. Animals can almost play any instrument in any orchestra. Whatever their properties, the sounds emanating from animals are never exactly the same from one song bout to another one, from one individual to another one, and from one species to another one. The variety of animal

sound is so high that audio robots that can identify and interpret a human voice and chat with you on the phone can hardly discriminate the sound of a dolphin from a whale's. This animal acoustic diversity may be a challenge for sound analysis and synthesis, but more importantly, they are a living treasure that has to be enjoyed and preserved.

Animal sound variety is the matter of bioacoustics and ecoacoustics, two closely related life sciences disciplines. As a bioacoustician or ecoacoustician I often have to face the naive but essential question about my research: "Come on, Jérôme, what's your job all about? Is there really a point in listening to cicadas?". Addressing this question is almost the same issue as wondering why we need to name insects and flowers, scrutinize the sky to discover new stars, analyse the old century playwrights style, or understand the physics of a golf ball. Such fundamental research participates in the world's knowledge and bioacoustics. Ecoacoustics are no exception. It is essential to describe and to understand the patterns and processing that determine natural acoustic environments. To me, it is as important a thing to know how a pigeon call is produced as it is to know how a financial index is computed. However, it would be unfair to say that bioacoustics and ecoacoustics have no application in our daily lives. Next time you fly, think that the engines of the plane you are comfortably seated in will not fail after sucking up flying birds owing to the loudspeakers at the end of the runway that play specific alarm sounds and scare them away. These specific sounds were designed by Thierry Aubin, a renowned bioacoustician.

One of the most important soundmark of my city working day is the subtle noise of my fingers on the computer keyboard. As soon as I have settled administration, teaching, curation, and supervision duties I open `R` and play with. But why does `R` has such an important role in my professional life?

I have been reluctant to programming for ages. As a schoolboy and later as a university student I always disliked programming courses as I was confused with FOR, WHILE, IF, THEN, DO, and other mysterious instructions. I surely have always been a software user for work, but I never thought that one day, I would have written a command rather than just clicked on a mouse. It actually took me a long time to get into R and, eventually, to love `R`. I was introduced to `R` by a colleague of mine, Michel Baylac, who is an expert in morphometrics. This was at a morning lab coffee break and here follows the discussion we had some ten years ago. Consider that the original dialogue was in French:

Jérôme: Michel, how did you do your elliptic Fourier analysis in your last paper? I could not find any statistical software that does it.

Michel: I used `R`.

Jérôme: Sorry?

Michel: I programmed the EFA with `R`.

Jérôme: Air? I do not know this software. What is it?

Michel: Well it is a programming language deriving from `S`.

Jérôme: Ace? I do not know that one either. So you can do your own analysis. Sounds great!

Michel: It is. And it is free. No licence to purchase.

A few coffee breaks and some explanations about software names later, I successfully installed `R` and got started with it but I quickly gave up so used I was to graphical user interface. Fortunately, Michel gave an `R`-based statistics Master course a few weeks later so I went back to school and followed his instructions to run `R` multivariate analysis. But I had no data to analyse at that time and a few months later I had forgotten almost everything. A year after, I joined my future wife Caroline Simonis, who is a co-author of `seewave`, for a second session of Michel's course and I joined her again for a course on linear models with `R` organized by Emmanuel Paradis, who wrote the best-seller *`R` for beginners* and the wonderful phylogenetics `ape` package. I was probably more motivated to be with Caroline than to learn `R`, but Caroline understood the interest of `R` much faster than I did and she talked me into starting to write more code rather than simply `lm(y~x)`. This time I had data to look into at hand and free time to play with `R`. I can still remember that winter evening when I could plot my very first pure-tone spectrogram. The image was incomplete and inaccurate, but to me it was a wonderful and shiny plot that motivated me to keep on with `R`. I was so amazed at being able to run such an analysis by myself knowing all the production steps perfectly that in the following weeks I could not stop writing basic sound-analysis dedicated functions. I was lucky to be greatly helped by Caroline and also by Thierry Aubin, my mentor in bioacoustics and the author of the Syntana sofware that inspired the main `seewave` functions.

`R` definitely changed my research. I was no more limited to the utilities provided by prohibitive closed source softwares that my department could not afford. I could do almost everything by myself: I could draw fieldwork observation maps, read, analyse, and change my sound samples. I could collect qualitative and quantitative data, run batch processes, apply multivariate analysis, plot high-quality graphics, and eventually produce a paper combining `R` and LaTeX in a nice layout. The most important thing was that I was not only a user but also a designer. I was able to create, imagine, and share new tools with others.

I'm telling this very personal story as I think this could be the future of anyone who is a bit afraid of software programming. Learning `R` is not that difficult with a little help—which a nice woman like Caroline can do but this is not absolutely necessary. Just keep in mind that it is worth an effort and that the reward will be tremendous.

This book was written for students who are interested in bioacoustics and ecoacoustics, but I really hope that it can help anyone who is willing to dive into

the fantastic area of acoustics and into the endless land of R. A book and a cake are not that different: they both require time and energy to be made, but they are consumed apace. I really hope this book has a nice taste, and I wish you very nice reading and programming nights!

Paris, France Jérôme Sueur

Acknowledgements

I would like first to thank Andreas Wessel who, some years ago, initiated this book by whispering my name in Lars Koerner's ear.

I am deeply indebted to Michel Baylac and Emmanuel Paradis for having taught me `R` at several occasions. Without their help, I would still be using spreadsheet applications to compute an arithmetic mean.

I was extremely lucky to be supervised during my research training by Thierry Aubin and Daniel Robert, my mentors for ever in bioacoustics.

The core of this book is the `seewave` package which was initiated with Thierry Aubin and Caroline Simonis. `seewave` has been growing up during the last 11 years thanks to the contribution of Ethan C. Brown, Marion Depraetere, Camille Desjonquères, François Fabianek, Amandine Gasc, Eric Kasten, Stefanie LaZerte, Laurent Lellouch, Jonathan Lees, Jean Marchal, Sandrine Pavoine, David Pinaud, Alicia Stotz, Luis J. Villanueva-Rivera, Zev Ross, Carl G. Witthoft, and Hristo Zhivomirov. `seewave` and related analyses have been improving thanks to ideas, comments, checks, or bug reports by Andrey Anikin, Charlotte Curé, Stéphane Dray, Denis Dupeyron, Almo Farina, Arnold Fertin, Kurt Hornik, Emiliano A. Laca, Nadia Pieretti, Daniel Ridley-Ellis, Jesse Ross, Pavel Senin, and Arvind Sowmyan. `seewave` has been maintained on CRAN thanks to the crucial help of Kurt Hornik, Uwe Ligges, Brian Ripley, and Simon Urbanek, all members of the `R` core team.

My motivation to complete this book mostly came from the imaginary students I had in mind when coding and writing. I also receive significant support from the students or junior researchers I was lucky to supervise their research in bioacoustics or ecoacoustics: Pablo Bolaños, Marion Depraetere, Camille Desjonquères, Manon Ducrettet, Amandine Gasc, Alexandre Kempf, Laurent Lellouch, Diego Llusia, Felipe Moreno, Alexandra Rodriguez, Alexandra Stotz, and Juan Sebastian Ulloa.

This book is acoustically and visually illustrated thanks to several people who shared sounds and/or images: Laurent Arthur, Thierry Aubin, Renaud Boistel, David Cartmell, Emmanuel Delfosse, Amandine Gasc, Joël Gilbert, Jean-François Julien, Diego Llusia, Ladislav Nagy, Christian Roesti, Frédéric Sèbe, and Andreas Trepte.

Readers of beta versions of the manuscript kindly took their precious time to check and improve the text: Andrey Anikin, Thierry Aubin, Stéphane Dray, Amandine Gasc, Jonathan Katz, Laurent Lellouch, Nathan Merchant, Benoît Obled, and Loïc Ponger.

Contents

Acronyms

A	Maximum amplitude
AM	Amplitude modulation
DC	Direct current voltage
DFT	Discrete Fourier transform
E	Energy (J)
F	Force (N)
FFT	Fast Fourier transform
FM	Frequency modulation (Hz)
FT	Fourier transform
I	Intensity ($\mathrm{W\,m^{-2}}$)
IDFT	Inverse discrete Fourier transform
IFT	Inverse Fourier transform
ISTDFT	Inverse discrete short-time Fourier transform
ISTFT	Inverse short-time Fourier transform
P	Power (W)
Q	Quality factor
RMS	Root-mean-square
S	Area ($\mathrm{m^2}$)
SIL	Sound intensity level (dB)
SPL	Sound pressure level (dB)
STDFT	Short-time discrete Fourier transform
STFT	Short-time Fourier transform
SVL	Sound velocity level (dB)
T	Period (s)
TKEO	Teager-Kaiser energy operator
Z	Acoustic impedance ($\mathrm{N\,s\,m^{-3}}$)
ZCR	Zero crossing rate
a	Instantaneous amplitude
a	Acceleration ($\mathrm{m\,s^{-2}}$)
c	Sound celerity ($\mathrm{m\,s^{-1}}$)
d	Duration (s)

f	Ordinary frequency (Hz)
f_c	Carrier frequency (Hz)
f_d	Dominant frequency (Hz)
f_r	Resonant frequency (Hz)
f_s	Sampling frequency (Hz)
f_N	Nyquist frequency (Hz)
p	Pressure (Pa)
p_0	Reference air pressure at 0 s.l.m (1.1013×10^5 Pa)
p_{ref}	Human auditory threshold in air (2×10^{-5} Pa $= 20\,\mu$Pa $= 0$ dB)
v	Particle velocity ($\text{m}\,\text{s}^{-1}$)
t	Time (s)
ω	Angular frequency (rad)
λ	Wavelength (m)
ρ	Volumetric mass density ($\text{kg}\,\text{m}^{-3}$)
φ	Angular phase (rad)

List of Figures

List of Tables

List of DIY Boxes

Chapter 1
Introduction

1.1 Sound as a Science Material

This book is dedicated to the analysis, and marginally to the synthesis, of sound using the software R. The functions and scripts introduced in the next pages were essentially designed to achieve scientific tasks—even if, interestingly, R appears to be used as a tool for the arts as well.

Sound is the raw material of a long list of scientific disciplines from nano-sciences to astronomy. Sound can be found indeed at the atom level with phonons which are quanta of vibration (Gustafsson et al. 2014) or at the microscale level with the detection of viruses with a microphone-like device (Cooper et al. 2001). The sound of a vast quantity of animal species have been recorded and studied, from the 2 mm aquatic bug *Micronecta scholtzi* (Sueur et al. 2011) to the about 25 m blue whale *Balaenoptera musculus* (Mellinger and Clark 2003). Plants have also drawn the attention of acousticians as plants might use sound for their own needs (Gagliano et al. 2012) or the sound they produce incidentally might be exploited for monitoring purposes (Felisberto et al. 2015). The analysis and use of sound by human is obviously multiple and multi-scale as sound is almost everywhere, in speech, music, engineering, industry, and medicine. Probably less known is the use of sound at a large scale for the monitoring of earthquakes (Sylvander et al. 2007), icequakes (Royer et al. 2015), or volcanic activity (Dziak et al. 2015). Sound does not exist in the vacuum of space, but the feet sensors of the European Space Agency Philae lander recorded vibrations when touching down the comet 67P/Churyumov-Gerasimenko.[1] Later, the European robot recorded the large-amplitude oscillations

Electronic supplementary material: The online version of this chapter (doi:10.1007/978-3-319-77647-7_1) contains supplementary material, which is available to authorized users.

[1] https://soundcloud.com/esa

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_1

of a magnetic field around the comet which could be translated into sound (Richter et al. 2015).

The reason of the importance of sound in science is probably found in the omnipresence of sound—signals and noises—in our everyday life. We each live in a specific audio environment, or niche, made of speech, music, and artificial or natural sounds. The content of this book is significantly biased by my own audio niche and by my experience in science linked to bioacoustics and ecoacoustics. Bioacoustics is a life science discipline that has been working on animal sound communication for more than 70 years. Bioacoustics mainly aims at explaining the neurology and mechanics of animal singing and hearing, at deciphering the nature and quantity of the information shared between communicating individuals, and at the evolution of animal sound through evolutionary time (Fletcher 1992). Ecoacoustics is a more recent discipline which is currently building a bridge between acoustics and ecology (Sueur and Farina 2015). Among others, ecoacoustics tackles ecology questions related to the monitoring of animal populations, animal communities, and landscapes.

The feedback I could receive from `seewave` users showed me that `R` could be used in many other disciplines than bioacoustics and ecoacoustics. I learned that `R` was part of studies related to noise assessment in the city, the detection and prediction of Alzheimer diseases, emotion regulation, space shuttle vibrations, animal monitoring through telemetry, brain speech processing in neurosciences, neurophysiological activation during music listening, and in timber analysis in musical acoustics.

1.2 Layout

The book is organized in 17 chapters and two appendices. The chapters were written one after the other one so that their reading is probably more comfortable in a sequential order. However, cross-references between chapters and sections should help the navigation across pages. Here follows a short overview of each chapter:

Chapter 2 *What Is Sound?* is a short introduction to sound, without any reference to `R`. This chapter can be skipped if the reader knows already the basis of acoustics. Note that this introduction is extremely brief covering a tiny part of acoustics so that readers might like to consult Speaks (1999), Hartmann (1998) and Larsen and Wahlberg (2017) for theory, Rumsey and McCormick (2002) for recording techniques, Fletcher (1992), Bradbury and Vehrencamp (1998), Hopp et al. (1998) for animal acoustics, and Rossing (2007) for almost everything about acoustics.

Chapter 3 *What Is `R`?* is a welcome talk to `R`, without any reference to sound. This chapter does not cover all topics of `R`, but it should contain enough information to understand the `R` code included in the following chapters. The literature about `R` has exploded in the last years: there are now hundreds of blogs and websites, and dozens of books have been published in several languages. Quick-R (Kabacoff 2013) and the R cookbook (Teetor 2011) are excellent references to supplement this short invitation to use `R`.

Chapter 4 *Playing with Sound* is a first contact with sound within R. The objective of this section is to get familiar with sound-specific classes and to learn how to import, to play, and to export sounds.

Chapter 5 *Display of the Wave* shows how to display sound in a time × amplitude plot. It mainly describes how to produce an oscillogram and how to compute and display an amplitude envelope. The important principle of a window sliding along the sound used for smoothing or discretisation is also explained.

Chapter 6 *Edition* shows how R can be used for sound manipulations, such as resampling, channels management, time edition, or amplitude changes. However, note that these changes are in most cases operated without graphical control so that other software could be preferred for full visual sound edition, such as the free and multi-platform solution Audacity.[2]

Chapter 7 *Amplitude Parametrization* lists the options to assess the amplitude features of a sound. This, in particular, includes details about signal-to-noise ratio, dB scales, and calibration.

Chapter 8 *Time-Amplitude Parametrization* reviews the techniques than can be invoked to take time measurements, that is, basically signal and pause duration, either through a manual process or with the help of an automatic process. Because time variations are tidily linked to amplitude variations, the chapter also describes how to assess amplitude modulations properties through a Fourier analysis.

Chapter 9 *Introduction to Frequency Analysis: The Fourier Transformation* is a modest introduction to the theory of the Fourier transformation that connects the time and frequency domains. The different transforms are quickly presented so that the next chapters can be read. Each equation is supported with raw R code to show that the Fourier transformation and its derivatives can be easily obtained. Much more details about Fourier mathematics can be found in Hartmann (1998) and Das (2012).

Chapter 10 *Frequency, Quefrency, and Phase in Practice* is a direct application of the preceding chapter as it mainly consists in explaining how to obtain, display, and describe the frequency spectrum of the Fourier transformation. The chapter also goes through the quefrency cepstrum and phase portrait. These sound transformation and visualization processes can be used to describe sound in the frequency domain, for instance, to estimate the dominant, the fundamental, and the harmonic frequency bands.

Chapter 11 *Spectrographic Visualization* is fully dedicated to the visualization of sound through the short-time discrete Fourier transform. The chapter details the different options to obtain, tune, and print a spectrogram, including a 2D spectrogram, a 3D spectrogram, and a waterfall display. The realization of a mean spectrum and a soundscape spectrum, which are computed on the short-time Fourier transform, is also introduced.

Chapter 12 *Mel-Frequency Cepstral and Linear Predictive Coefficients* deals with two features, mel frequency cepstral coefficients and linear predictive coef-

[2] http://www.audacityteam.org/

ficients that are mainly used in speech analysis. The first coefficients operate a sort of data compression when the second is used to separate the source from the filter of speech production. A thorough review of speech analysis can be found in Quatieri (2002).

Chapter 13 *Frequency and Energy Tracking* shows how to follow the time variation of particular frequency bands, such as the dominant frequency, the fundamental frequency, or the speech formants. This chapter also refers to the Hilbert analytic signal and the zero-crossing method that can be used to estimate the instantaneous frequency.

Chapter 14 *Frequency Filters* indicates how to apply frequency filters to remove unwanted sounds. The chapter covers filters with predefined frequency transfer functions, as the preemphasis and Butterworth filters, and filters which cut off frequencies can be defined by the user as FIR filters.

Chapter 15 *Other Modifications* adds options for modifying sound not only in the frequency domain but also in the time and amplitude domains. Solutions are, for instance, given to add echoes, to change the frequency content without altering the time content, or to change the time $\times$ frequency features of a sound section.

Chapter 16 *Indices for Ecoacoustics* is a chapter for those who are interested in the acoustic indices developed for ecoacoustics. The main α and β indices are reviewed one by one, and statistic solutions are provided to treat dissimilarity matrices built with β indices.

Chapter 17 *Comparison and Automatic Detection* is dedicated to the methods that compare two sounds. Cross-correlation of amplitude envelopes, frequency spectra, and spectrograms are evoked together with the computation of the frequency coherence. The dynamic time warping technique, which seeks for the best alignments of time series of unequal length, is also covered. Finally, the main functions of the package `monitoR` are arranged in a recipe so that a supervised automatic identification can be proceed.

Chapter 18 *Synthesis* quits the domain of sound analysis to get into sound synthesis. Sound synthesis in `R` is not highly developed, and sound designers might be a bit frustrated by the options currently offered. Nonetheless, the chapter shows that it is possible to generate *de novo* different types of noises, pulses, square signals, sawtooth signals, triangle signals, pure tones, chirps, harmonics sounds, amplitude, and/or frequency-modulated sounds. Associated with edition functions seen in Chap. 6, the additive and modulation synthesis functions of `R` are undoubtedly useful to forge animal vocalizations, musical instrument notes, and human voice.

Appendix A *List of `R` Functions* contains a table of the functions cited in the book. The functions are grouped by themes, and a reference to related chapter is provided.

Appendix B *Sound Samples* lists the sounds used as examples. The sounds are stored in a directory named `sample` that can be downloaded as supplementary material at the following address: https://doi.org/10.1007/978-3-319-77647-7_1.

1.3 Convention for Notation and Code

Notations for continuous and discrete mathematical objects differ. Continuous objects are written with brackets. For instance, a time series will be written $x(t)$. Discrete objects, that is, objects that are made of separated values or samples, are written with square brackets or with subscripts. Discrete objects are defined with an index, either i or n. A time series could be then written $x[i]$ or x_i ($x[n]$ or x_n).

Operations follow similar rules. For instance, the sum is written in its continuous form following:

$$S = \int_{-\infty}^{\infty} x(t)$$

and its discrete form as:

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x[i]$$

or with an index n:

$$\sum_{n=1}^{N} x_n = \sum_{n=1}^{N} x[n]$$

A derivative according to time will be written in its continuous form according to:

$$\dot{x}(t) = \frac{\partial x}{\partial t}$$

and in its discrete form as:

$$x = x[i+1] - x[i] = x_{i+1} - x_i$$

or with an index n:

$$x = x[n+1] - x[n] = x_{n+1} - x_n$$

By convention `R` code is formatted following. Note that the prompt sign `>` has been intentionally removed to facilitate copying and pasting code:

```
input code
output (results)
```

Pseudo-code and terminal commands (shell) are formatted following:

```
peudo-code
or
shell command
```

In text, `R` functions are referred with parentheses with a type writer style following `foo()`. Packages, objects, arguments, and attributes are also written with typewriter style as in `MASS`, `wave`, or `plot`. Character strings are double quoted as `"hello moon"`

Boxes tagged with the acronym DIY[3] detail how the reader could have written already existing functions or could develop new tasks by himself/herself.

1.4 Book Compilation

The book was generated using free and open-source tools only published under the GNU licence. The manuscript was built with `knitr` (Xie 2013), an `R` package that allows to interlace instructions written in `R` and in the typesetting system LaTeX.[4] The manuscript was written with the text editor `emacs`[5] working with the modes `ESS`[6] and `AucTeX`.[7] The literature references were managed with `BibTeX`.[8] All illustration, except one handmade drawing, a few diagrams, and the photographies, was generated with `R` without any image post-processing.

The book was compiled on January 10, 2018 with `R` version 3.4.3 Kite-Eating Tree so that any changes appearing in the packages after this date and this version may affect the repeatability of the codes provided.

[3] Do it yourself!

[4] http://www.latex-project.org/

[5] http://www.gnu.org/software/emacs/

[6] http://ess.r-project.org/

[7] http://www.gnu.org/software/auctex/

[8] http://www.bibtex.org/

Chapter 2
What Is Sound?

2.1 A Debate Under a Dangerous Tree

Starting with a definition is never straightforward, and the easiest solution is probably to open a dictionary, or to be even more lazy, to visit an online dictionary, in order to check what official linguists say. If we consult the entry "sound" of the Oxford Dictionary,[1] we find a definition referring indirectly to the main three steps of information theory, that is, emission, propagation and reception of sound (see Sect. 2.5). Oxford Dictionary says:

> vibrations that travel through the air or another medium and can be heard when they reach a person's or animal's ear.

But would this definition be unclear if we would remove the last part dedicated to hearing? Oxford Dictionary makes also the distinction between a *person* and an *animal*, a discrimination that is difficult to admit when working with evolution—but this is another matter of discussion.

For Collins Dictionary[2] it seems that there is no need to refer to hearing as for Collins dictionary sound is:

> a periodic disturbance in the pressure or density of a fluid or in the elastic strain of a solid, produced by a vibrating object. It has a velocity in air at sea level at 0°C of 331 metres per second (741 miles per hour) and travels as longitudinal waves.

Collins definition is therefore close to the definition to be found in a physics textbook when Oxford definition focuses on perception. In other terms, these two dictionaries answer differently to the question debated for a long time, and that is difficult to avoid here, regarding the reality of sound:

> If a tree falls in a forest and no one is around to hear it, does it make a sound?

[1] http://www.oxforddictionaries.com/

[2] http://www.collinsdictionary.com/

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_2

Collins answers *no* very clearly, Oxford answers *yes* by omission.

The falling tree sound question opens essential thoughts in metaphysics, but we will introduce sound mainly in terms of physics, statistics, and electronics, neglecting philosophical questions. We will also refer to sound as a solution to transfer information between animals including human mammals.

A falling tree in a forest does not produce a single sound but rather a complex mix of frightening sounds that may be very difficult to describe either literally or mathematically. It is probably easier to start with a simple sound like the pure musical tone emanating from an A-flat tuning fork used to tune a guitar. A tuning fork is a two-pronged metal fork that vibrates at a specific resonating frequency, here generating a A-flat musical note. The tuning fork is a sound **source**. When stroked, each tine oscillates symmetrically in the plane of the fork inducing air particle, that was in an equilibrium state, to move. The oscillations of the tuning fork are **transduced** into air oscillations. The oscillation of air particles induces cyclic compressions and rarefactions of air. These variations of air particle motion and air pressure are in essence sound (Fig. 2.1). The air is the **medium** where sound propagates. No air, no sound. However, the tuning fork can be also plunged into the water or can be in contact with a wooden box. The tuning fork will make the water or the wood vibrate and, therefore, produce sound. Any gas, liquid, or solid can be a medium for sound, but sound properties will differ significantly depending on the medium.

If we now place an elastic membrane maintained within a rigid frame at a certain distance from the tuning work, the membrane will move sympathetically with air and vibrate at the same rate as the tuning fork. This simple membrane is a **receiver**;

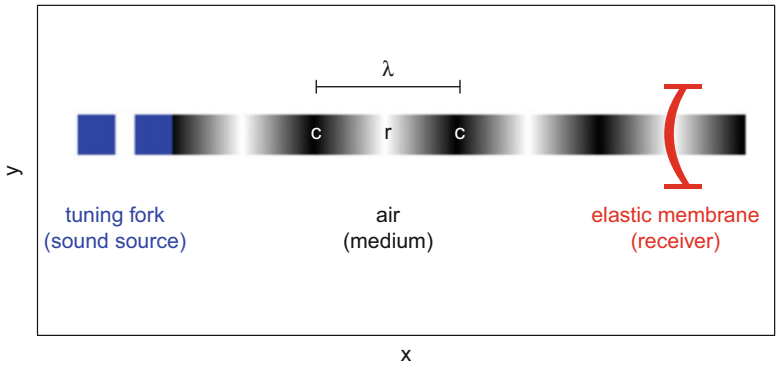


Fig. 2.1 Sound emanating from a tuning fork. The two tuning fork hinges are represented from above with two blue squares. Their vibrations generate a sound that propagates as a longitudinal wave in air. Sound is represented along a single direction with an alternation of air rarefaction (r) and compression (c) with a wavelength λ. A simple framed elastic membrane at a fixed position in the (x, y) space vibrates sympathetically with sound. This is an oversimplified representation of sound propagation around a tuning fork; see Russell et al. (2013) and Russell (2000) for a complete description

it **transduces** air vibrations into solid vibrations. Such a simple receiver is actually the main first step of animal hearing and microphone systems.

Sound radiation around a tuning fork is actually more complex (Russell et al. 2013; Russell 2000), but the essential is here. To have sound, we need a source, a medium, and, optionally, a receiver.

2.2 Sound as a Mechanical Wave

2.2.1 Air Particle Motion

When a sound is produced in air by a driven tuning fork, air particles move. This motion is an oscillation around a resting position, something similar to the oscillations of a pendulum oscillation around a position of equilibrium. Particles do not travel; they just move around their initial position. There are three main derived quantities to describe air particle motion:

1. Displacement (x): this is the distance in meters a particle moves away from its resting position.
2. Velocity (v): this is the amount of displacement (x) per unit time (t) or the time rate of displacement. Velocity includes both magnitude and direction of the displacement. Velocity is the first derivative of x with respect to t, expressed, for instance, in $\mathrm{m\,s^{-1}}$:

$$v(t) = \frac{\delta x}{\delta t}$$

3. Acceleration (a): this is the change of velocity (v) per unit time (t) or the time rate of speed. Acceleration is the first derivative of v with respect to t and the second derivative of x with respect to t, expressed, for instance, in $\mathrm{m\,s^{-2}}$:

$$a(t) = \frac{\delta v}{\delta t} = \frac{\delta^2 x}{\delta t^2}$$

These derived quantities do not vary in phase: when displacement is maximum, velocity is null, and acceleration is minimum (i.e., deceleration is maximum). In other words, there is a $+\pi \div 2$ rad or $+90°$ phase shift between displacement and velocity and $+\pi$ rad or $+180°$ phase shift between displacement and acceleration. However, it is important to note that these phase shifts are true only in the near-field of the source. In the far-field, the pressure and the particle velocity vary in phase. The transition between the near- and far-field occurs at a specific distance, known as the Fraunhofer distance, $r = 0.16 \times \lambda$, where λ is the wavelength of the sound (see Sect. 2.2.2 for a definition of wavelength).

These derived quantities are rarely considered, but they explain air pressure, one of the most important quantities used in acoustics.

2.2.2 Air Pressure Variation

Force (F) is a quantity that we refer almost every day but that we may not understand so well. We apply a force on a static object when we try to change its position by pushing or pulling it. For instance, we apply a positive force on a door when opening it and a negative force when closing it. Pushing or pulling a door is a way to change the velocity (v) of the door in respect to time. We saw that a change of velocity with time is the derived quantity acceleration (a). It is also quite intuitive to understand that a metallic door will be more difficult to open than a wood door: the force depends on the mass (m) of the door. Hence, a force F results of the product between mass and acceleration as given by the Newton's second law of motion:

$$F = m \times a$$

with F in newtons (N).

For sound, the force of one moving particle on another depends on particle acceleration and particle mass. However, we almost never measure this tiny force but the sum of many particle forces acting perpendicularly on a surface, like on the small elastic membrane we placed away from the vibrating tuning fork. This sum is the pressure (p), and sound is usually described by the pressure variations that occur in the medium. A pressure is simply the amount of force per unit area (S) obtained by computing the following ratio:

$$p = \frac{F}{S}$$

with p in newtons per square meters ($\mathrm{N\,m^{-2}}$) or Pascals (Pa)

This means that a force F applied on a small area will generate a greater pressure than when applied on a large area.

Knowing that F is the product of mass and acceleration and that acceleration is the second derivative of displacement with respect to time, the pressure can also be written with respect to displacement:

$$p = \frac{m}{S} \times \frac{\delta^2 x}{\delta t^2}$$

This means that pressure does not vary in phase with displacement but in phase with acceleration, i.e., with a phase shift of $+\pi$ rad or $+180°$ with displacement.

Air pressure at sea level is $p_0 = 1.1013 \times 10^5$ Pa. A sound will then change p_0 to lower and higher values. These changes are in the range of 10^{-3} and 10 Pa.

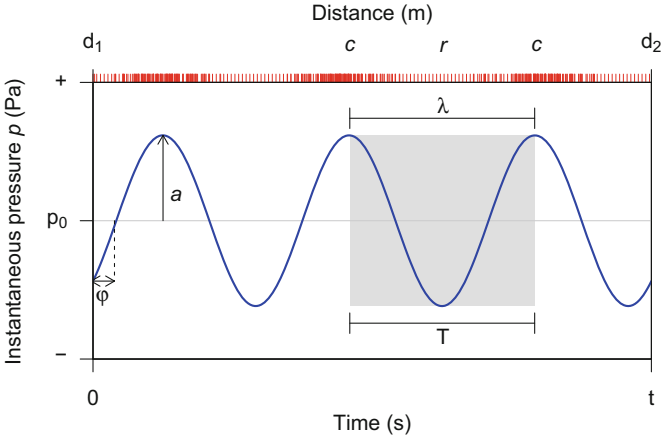


Fig. 2.2 Sound pressure (p) and amplitude variations. The sound was recorded at time $t = 0$ and at distance d_1 from the source with a $-\pi \div 4$ rad or $-45°$ phase shift φ. The bottom x-axis shows the time t in seconds, the top x-axis shows the distance in meter and the y-axis is the instantaneous pressure p in Pascal. In this ideal case, air pressure oscillates cyclically as a sinusoidal function around p_0. The gray rectangle delimits one cycle. In the time domain, the interval between two compression peaks is the period (T). In the space dimension, the distance between two compression peaks is the wavelength (λ). The red vertical bars on the top x-axis represent the density of air particles. Low and high air particle density corresponds to air rarefaction (r) and compression (c), respectively

Sound can be considered as a longitudinal pressure wave (or transversal pressure wave in solids) (Fig. 2.2). We saw that air particles do not travel, but the pressure wave does. The wave starts at the surface of the source and propagates through the medium. The distance between two maximal compressions is the wavelength (λ).

The small membrane that has a fixed position in the space will receive a succession of sound waves as long as the tuning fork vibrates. This means that the membrane will be pushed forward when the air will be compressed (high pressure) and pulled backward when the air will be rarefied (low pressure). If we measure the displacement of the membrane around its resting position, we will observe a cyclic displacement repeated 440 times per second corresponding to the pitch of an A-flat musical note.

As the membrane displacement is related to air pressure, it can be used to measure air pressure fluctuations. This is how sound is the most often visualized. We will now see how to describe this wave.

Sound can be described in two reference frames. The first reference frame is made of two dimensions (t, a), time and amplitude. This reference frame is used when the sound is observed—recorded—at a fixed position in the space. This is the case of the small membrane recording the tuning fork. In this case, the main parameters are the time, or duration, (t), the instantaneous amplitude (a), and the cycle period (T) which is the time taken by the wave to return to its initial state and to complete a cycle.

The second reference frame is more complex as it is made of five dimensions (x, y, z, t, a), with (x, y, z) a three-dimensional frame defining space, t time, and a amplitude. This reference frame is often used to map sound propagation in the medium. The distance between peaks of pressure or amplitude in this reference frame is the wavelength (λ). This wavelength depends on the period (T) and sound celerity (c), following:

$$\lambda = c \times T = \frac{c}{f}$$

with λ in m. Sound celerity greatly varies from one medium to another one and with medium properties. Celerity of a 1000 Hz sound is 337 m s^{-1} in air and 1447 m s^{-1} in freshwater at 10 °C.

The A-flat tuning fork produces at 20 °C a sound with wavelength of 0.777 m.

It is important not to mistake wavelength (λ) for particle displacement (x) and particle velocity (v) for sound celerity (c), respectively.

To better visualize sound, wonderful animations are available at the websites of Daniel Russell (Pennsylvania State University, USA)[3] and of the Institute of Sound and Vibration Research (University of Southampton, UK).[4]

2.2.3 Amplitude

The amplitude can be measured in four different main ways (Fig. 2.3):

1. Instantaneous amplitude (a): amplitude measured at a time t,
2. Maximum amplitude (A): the maximum of the absolute value of the amplitude,
3. Peak-to-peak amplitude (pk-pk): the range of amplitude between the minimum and maximum values,
4. Root-mean-square amplitude (rms, RMS): the root-mean-square or quadratic mean is the square root of the mean of the squares. In other words, the sound wave is first squared, then the mean of the squared values is computed, and finally the square root of this mean is computed:

$$\text{rms} = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} {x_i}^2}$$

The rms is commonly used as it provides an estimation of the amplitude average.

[3] http://www.acs.psu.edu/drussell/demos.html

[4] http://resource.isvr.soton.ac.uk/spcg/tutorial/tutorial/Tutorial_files/Web-basics.htm, accessed on 2017-02-06.

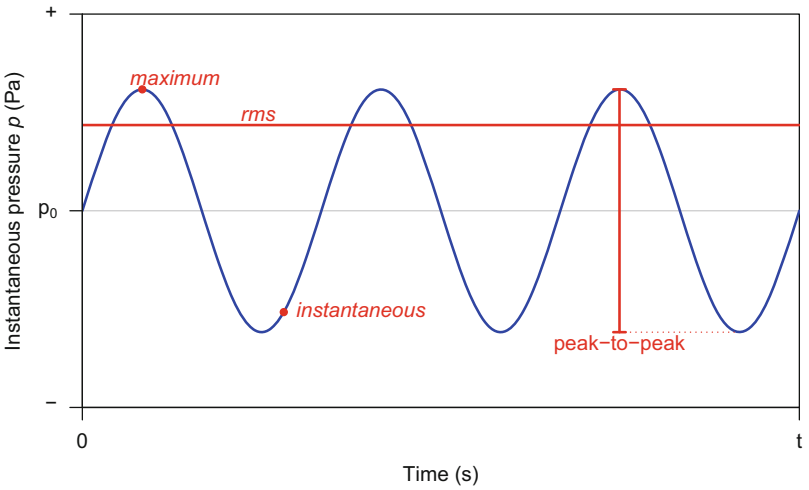


Fig. 2.3 Amplitude (A). The three main amplitude quantities of a sound: the instantaneous, the maximum, the peak-to-peak, and the average (root-mean-square, rms) amplitude

Sounds transfer **energy** to the medium. The amount of energy carried by the sound wave per unit time is the **sound power**. Power is therefore obtained by:

$$P = \frac{E}{t}$$

with E the energy in Joule and t time in seconds and P in $\mathrm{J\,s^{-1}}$ or Watts (W).

However, it is almost impossible to measure sound power at every point in space; it is therefore more convenient and usual to measure power only at the location of the receiver, here the small vibrating membrane. This leads to the definition of sound intensity (I) that is sound power that interacts with the area (S) of the receiver perpendicular to sound:

$$I = \frac{P}{S}$$

with I in $\mathrm{W\,m^{-2}}$.

Displacement, pressure, energy, power, and intensity measure amplitude variations varying along an absolute scale. However, sound amplitude is often expressed along a relative scale. A relative scale is a scale built in reference to one or more specific values. For instance, the Celsius degree used to measure temperature is a relative scale with two references: 0 °C for water freezing point and 100 °C for water boiling point.

The sound pressure that can be received by a human ear extends from 2×10^{-5} Pa to about 200 Pa covering a range of 2×10^{8} Pa. This hearing range is so large that

it was thought that a logarithm scale would be more adapted to measure sound amplitude, in particular sound pressure. The logarithm function to the base 10 has the main properties to enhance low values and to reduce high values, compressing the dynamic range of data. Combining relative measure and logarithm to the base 10, a relative unit, named Bel (B) after the name of Alexander Graham Bell, was coined by computing a ratio between an observed value x and a reference value x_{ref}:

$$\mathrm{B} = \log_{10}\left(\frac{x}{x_{ref}}\right)$$

A deciBel or dB is ten times a B:

$$\mathrm{dB} = 10 \times \log_{10}\left(\frac{x_1}{x_2}\right)$$

Applying this expression to sound pressure, we can define sound pressure level (SPL) in dB as:

$$\mathrm{SPL} = 20 \times \log_{10}\left(\frac{p_1}{p_2}\right)$$

If the pressure of the first sound p_1 is twice higher than the pressure of the second sound (p_2), we then have a $\approx$6 dB ratio between the two sounds:

$$\begin{aligned}\mathrm{dB} &= 20 \times \log_{10}\left(\frac{2}{1}\right)\\ &= 6.02\end{aligned}$$

Table 2.1 provides some usual dB ratios, and Fig. 2.4 shows the dB function with respect to the linear ratio of two pressures.

However, such a measure is relative and difficult to use in most cases. It is better to refer to a reference pressure p_{ref} to obtain values that can be compared across

Table 2.1 dB ratios

dB	Linear ratio
0	1
+3	$\sqrt{2}$
+6	2
+20	10
+40	100
+60	1000

Equivalent between dB and linear multiplication ratios

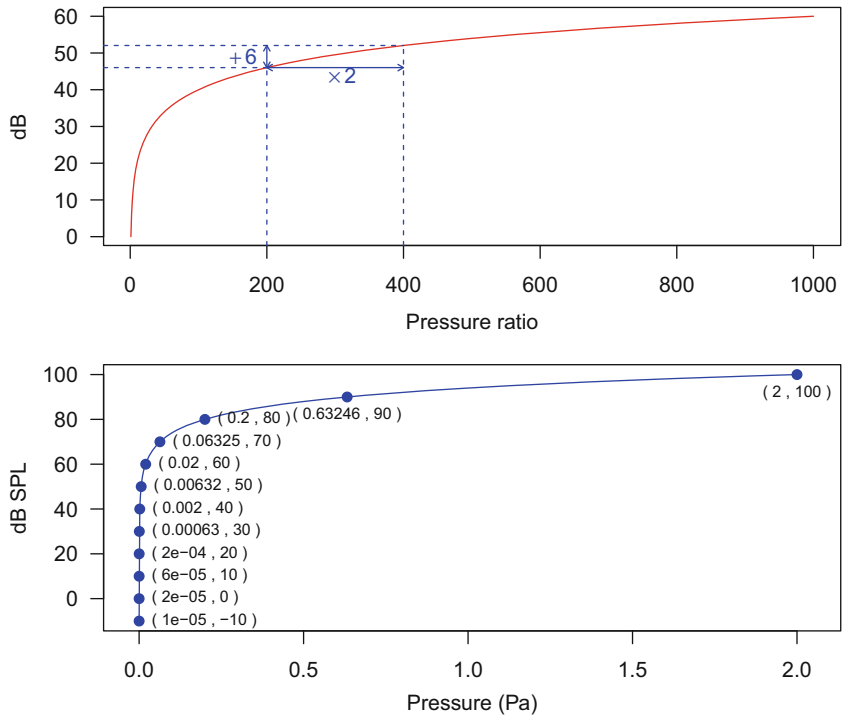


Fig. 2.4 dB scale. Top: relation between the ratio of two pressures and the corresponding dB value. Doubling the pressure is equivalent to an addition of 6 dB. Bottom: from pressure in Pa to sound pressure level (SPL) in dB. Values are given for every 10 dB

studies. We can then adapt the formula following:

$$\text{SPL} = 20 \times \log_{10}\left(\frac{p}{p_{ref}}\right)$$

For sound in air, the reference pressure is $p_{ref} = 2 \times 10^{-5}$ Pa corresponding to the human hearing threshold at 1000 Hz. In water, the reference is $p_{ref} = 10^{-6}$ Pa. The reference pressure can change; it is thus absolutely necessary to specify it with the distance of measure when reporting SPL values. For instance, we should say "a 440 Hz tuning fork produces a sound with a pressure level of 60 dB *re* 2×10^{-5} Pa at a distance of 1 m" and not only "a 440 Hz tuning fork produces a sound with a pressure level of 60 dB." SPL is not an easy scale as it is based on a ratio and on a logarithmic function. The most important thing to remember is that the scale is not linear and that doubling the pressure anywhere along the pressure scale always results in adding +6 dB (Fig. 2.4).

In the same way, the intensity can be expressed along a dB level scale leading to sound intensity level (SIL). Intensity is linearly proportional to pressure following the relation:

$$\begin{aligned} I &= \frac{p^2}{Z} \\ &= \frac{p^2}{\rho c} \end{aligned}$$

with Z the acoustic impedance of the medium that is obtained by multiplying the volumetric mass density (ρ) by sound celerity (c). The acoustic impedance is a concept that allows to measure the resistance of the medium to sound propagation. A high impedance indicates that sound will be highly damped. Knowing this, we can write a relation between sound pressure level and sound intensity level:

$$\begin{aligned} \text{SPL} &= 20 \times \log_{10}\left(\frac{p}{p_{ref}}\right) \\ &= 20 \times \log_{10}\left(\frac{IZ}{I_{ref}Z}\right)^{\frac{1}{2}} \\ &= 10 \times \log_{10}\left(\frac{I}{I_{ref}}\right) \\ &= \text{SIL} \end{aligned}$$

with $I_{ref} = 10^{-12}\,\text{W}\,\text{m}^{-2}$ in air, and $I_{ref} = 6.7 \cdot 10^{-19}\,\text{W}\,\text{m}^{-2}$ in water.

This means that SPL and SIL measure the same quantity. They are equivalent.

There is also another sound level measure, less commonly used, that estimates the level of particle velocity level or sound velocity level (SVL):

$$\text{SVL} = 20 \times \log_{10}\left(\frac{v}{v_{ref}}\right)$$

with $v_{ref} = 5 \times 10^{-8}\,\text{m}\,\text{s}^{-1}$ for air.

The dB is a logarithm unit that follows specific arithmetic laws making it difficult to compute a mean, a standard deviation, or any statistical parameter. It is therefore often necessary to convert back dB values to linear values. Absolute sound pressure p, sound intensity I, and particle velocity v can be recovered from a dB value by applying the following equations:

$$p = p_{ref} \times 10^{\frac{\text{SPL}}{20}}$$

$$I = I_{ref} \times 10^{\frac{\text{SIL}}{10}}$$

$$v = v_{ref} \times 10^{\frac{\text{SVL}}{20}}$$

For instance, a 60 dB value corresponds to:

$$\begin{aligned} p &= 2 \times 10^{-5} \times 10^{\frac{60}{20}} \\ &= 0.02\,\text{Pa} \\ I &= 10^{-12} \times 10^{\frac{60}{10}} \\ &= 10^{-6}\,\text{W}\,\text{m}^{-2} \\ v &= 5 \times 10^{-8} \times 10^{\frac{60}{20}} \\ &= 5 \times 10^{-5}\,\text{m}\,\text{s}^{-1} \end{aligned}$$

The dB is a unit based on human hearing threshold, but it is not totally adapted to human frequency sensitivity. The human ear does not perceive frequency in a linear way: a sound produced at 1000 Hz at 80 dB SPL will appear louder than a 440 Hz sound produced at exactly the same level. A solution to take into account this nonlinearity is to apply a weight to the dB in relation to frequency. There are four main dB weightings named dB(A), dB(B), dB(C), and dB(D) which curves slightly differ according to frequency (Fig. 2.5). These particular dB are commonly used in sound level meters for noise measurement.

Sound energy, and therefore sound amplitude measured in dB, decreases when sound propagates through the medium. Sound attenuation is due to three main factors: (1) spreading losses, (2) medium absorption (heat conduction, shear viscosity, molecular relaxation losses), and (3) scattering (reflection, refraction, diffraction, and absorption of the wave due to impedance changes in the medium).

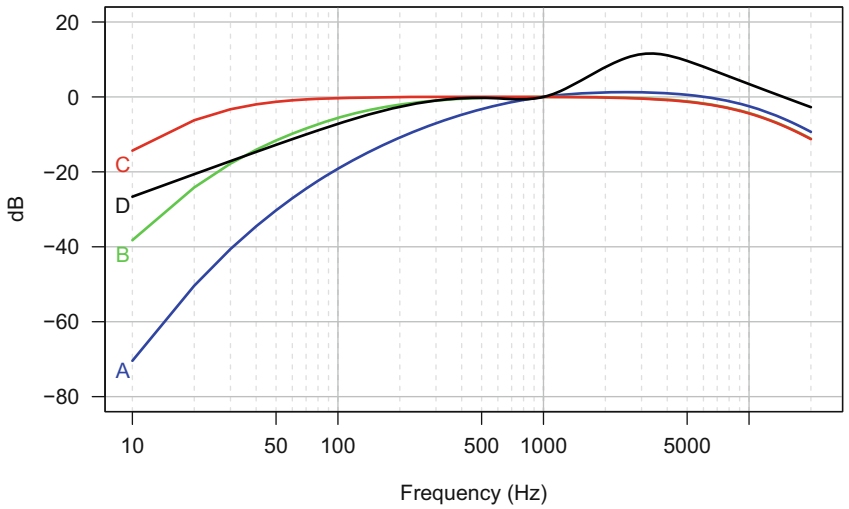


Fig. 2.5 dB weighting curves. The weightings curves of dB(A), dB(B), dB(C), and dB(D) according to frequency. The code used to produce this figure is given in Sect. 7.2.2

Medium absorption and scattering are rather difficult to modelize as they depend on several parameters such as frequency, humidity, temperature, and pressure and by the physical properties of the obstacles generating scattering. Spreading losses are more easy to predict in particular in the case of a spherical sound source propagating through a free and unbounded medium. In these conditions, the intensity I at a distance d from the source of power P is given by:

$$I = \frac{P}{4\pi d^2}$$

This equation is obtained as the surface area of the sound sphere has an area of $4\pi d^2$. This means that the intensity decreases with the inverse square of the distance from the source; this is known as the inverse square law.

If the intensity of a spherical sound can be measured at a certain distance, we can refer to a reference intensity I_{ref} and a reference distance d_{ref}. This measure, obtained, for instance, with a sound level meter, can be used to predict what would be the intensity I further away, at a distance d from the source. We have:

$$I_{ref} = \frac{P}{4\pi d_{ref}^2}$$

such that the ratio of the intensities is:

$$\frac{I}{I_{ref}} = \left(\frac{d_{ref}}{d}\right)^2$$

giving:

$$I = I_{ref} \times \left(\frac{d_{ref}}{d}\right)^2$$

that can also be written:

$$I = \frac{I_{ref}}{D^2}$$

with

$$D = \frac{d}{d_{ref}}.$$

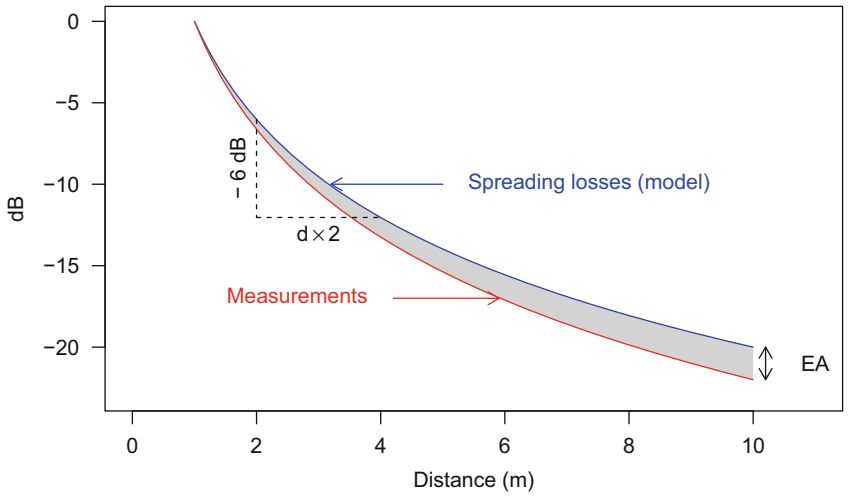


Fig. 2.6 Sound attenuation for a spherical source. Curves of dB attenuation with distance due to spreading losses in a free and unbounded medium (model) and of what could be measured in the medium (measurements). The difference between the two curves due to medium absorption and scattering is named excess of attenuation (EA). The measurement curve is here still idealized as scattering effects will produce an irregular curve

If the intensity is expressed as a sound intensity level, the equation becomes:

$$\begin{aligned} \text{SIL} &= 10 \times \log_{10}\left(\frac{I_{ref}}{D^2}\right) \\ &= 10 \times \log_{10}(I_{ref}) - 20 \times \log_{10}(D) \\ &= \text{SIL}_{ref} - 20 \times \log_{10}\left(\frac{d}{d_{ref}}\right) \end{aligned}$$

This equation stands as well for sound pressure as we have seen above that SIL and SPL level are equivalent. This equation tells us that when doubling the distance between a reference and a measure, that is when $D = 2$ the attenuation is ≈ -6 dB as:

$$-20 \times \log_{10}(2) = -6.0206$$

Figure 2.6 is a graphical display of this spreading losses model. The residuals between the observed values obtained with direct measures achieved in the medium and the values expected by the model are known as the excess of attenuation (EA) and are due to medium absorption and scattering.

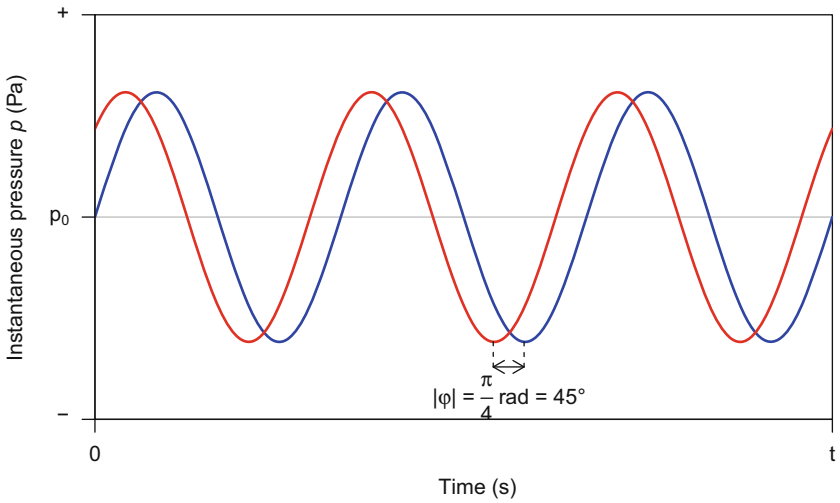


Fig. 2.7 Phase (φ). Two sounds with similar amplitude and frequency but different phase. There is a $\pi \div 4$ rad or 45° shift between the two waves

2.2.4 Phase

The phase (φ) is the horizontal translation of a cyclic function with respect to time. For a sine sound, phase is therefore a temporal translation of $\sin(t)$ with respect to t. Two sine sounds with different phases will not start with the same instantaneous amplitude. For instance, a sound with a phase $\varphi = 0$ starts with an instantaneous amplitude $a_{t=0} = \sin(0) = 0$ when a similar sound with a phase $\varphi = \pi \div 4\,\text{rad} = 45°$ starts with an instantaneous amplitude $a_{t=0} = \sin(0+\pi \div 4) = \sqrt{2} \div 2 \simeq 0.707$ (Fig. 2.7). The sign of the phase shift between two sounds depends on the sound used as a reference. In Fig. 2.7, the blue wave is delayed with respect to the red wave; the phase shift is negative. Respectively, the red wave is forward the blue one; the phase shift is in that case positive. The absolute value of the phase shift, as indicated on Fig. 2.7, does not indicate any phase direction but just phase shift magnitude. Phase might appear not crucial; nonetheless it is actually one of the quantities that is used by hearing systems to localize a sound source, and phase can be very important when synthesizing a sound (see Sect. 18.4).

2.2.5 Duration

Sound duration is rather simple to understand: this is the time interval between the start and end of a sound. Figure 2.8 shows two sine waves of different duration. The duration is in that case easy to extract, but measuring the exact duration of a sound

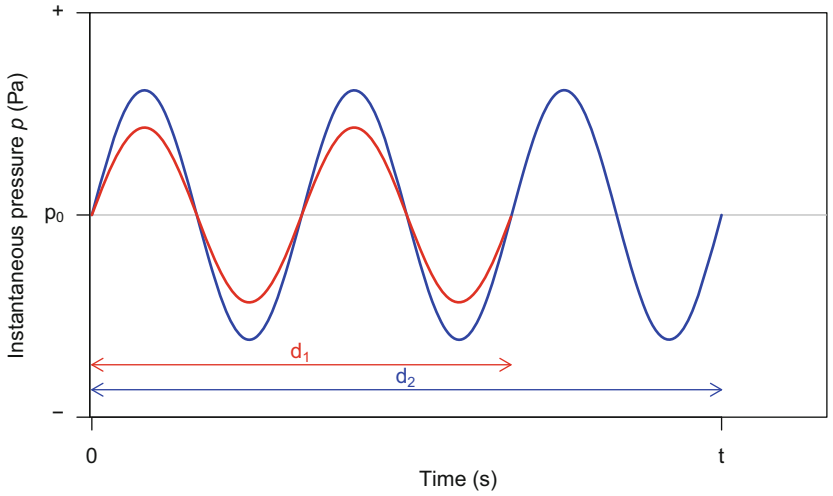


Fig. 2.8 Duration (d). Two sounds of different duration, the red sound being a third shorter than the blue one ($d_1 = 2 \div 3 \times d_2$). The amplitudes of the two sounds were set to different values to allow comparison

recorded outdoor might that be arduous as it is not always clear when a sound begins and ceases due to background noise and echoes (see Sect. 8.1).

2.2.6 *Frequency*

In a broad sense, frequency, or rate, is a quantity that measures the number of time an event (n) occurs during a specific time interval (Δt). Frequency can be then written as:

$$f = \frac{n}{\Delta t}.$$

For a sine wave, frequency is the number of time an amplitude cycle is repeated during 1 s. The A-flat tuning forks generate a sine wave with 440 cycles/s. The frequency is then 440 cycle s^{-1} or, more commonly, 440 Hz. When we refer to the sine function within the unit circle, the frequency of a cycle is named the angular frequency (ω). When considering a wave along the time axis, the frequency is named the ordinary frequency or, more simply, the frequency (f). Rather than counting the number of cycles produced by second, it is usual to measure the duration of one cycle and to compute the inverse of this duration, that is, the reverse of the period (T). The period of the A-flat tuning fork is about 0.00227 s, multiplying this value

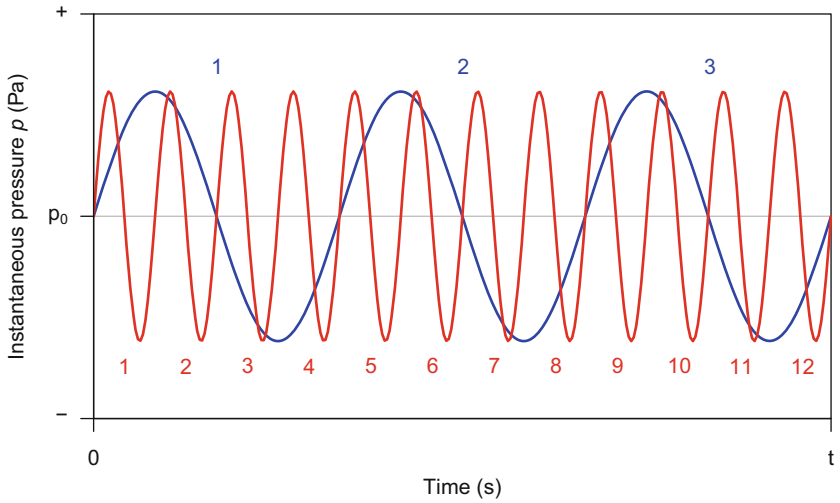


Fig. 2.9 Frequency (f). Two sounds with different frequencies: the red sound has a frequency four times higher than the blue one. In other words, there are three blue cycles and twelve red cycles, or there are four red cycles for a single blue cycle. If $t = 1$ s, then the frequency of the blue wave is 3 Hz, and the frequency of the blue wave is 12 Hz

by 440 returns 1. We then end up with the following basic formulae:

$$f = \frac{1}{T}$$
$$= \frac{\omega}{2 \times \pi}$$

Even if time and frequency are very often considered as separate domains, it is absolutely clear that frequency is just another way to measure a time quantity. This time-frequency tight connection is at the origin of most of the difficulties in time-frequency representation and analysis (see Chaps. 11 and 13). The shorter the cycle period is, the higher the frequency is (Fig. 2.9). In theory, there are neither lower nor upper limits for frequency ranging from a few Hz (low-frequency sound or infrasound) to several kHz or MHz frequency (high-frequency sound or ultrasound).

All the sounds we have looked at so far were simple sound waves or pure tones, that is, they were made of single pure frequency following a sine or cosine function. However, sound is, in most cases, complex being made of a combination of several tones, i.e., made of an addition of several sine or cosine functions. There is a specific terminology for the complex sine waves:

fundamental f_0, the greatest common divisor of the set of frequencies composing the sound wave, this is the sine wave with the lowest frequency. The fundamental is sometimes considered as the first harmonic or first partial.

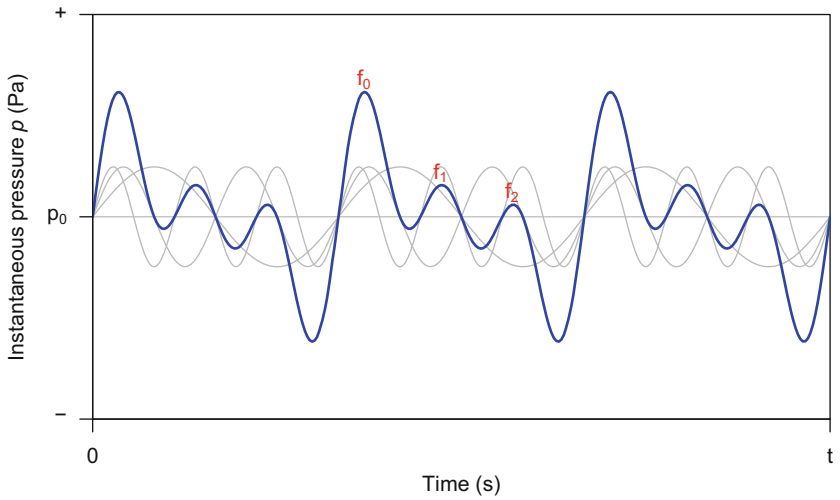


Fig. 2.10 Harmonics. Sound made of three tones with a harmonic ratio: the fundamental (f_0), the first harmonic (f_1), and the second harmonic (f_2). The light gray lines correspond to these three tones isolated

overtone, any sine wave with a frequency higher than the fundamental. Synonym: partial.

harmonic f_i, an overtone that is a perfect integer multiple of the fundamental. f_i is the ith harmonic.

inharmonic, an overtone that is not a perfect integer multiple of the fundamental frequency.

resonant frequency, frequency of a resonator.

formant, a group of frequencies amplified by a resonator.

dominant frequency, the frequency of highest amplitude. This can be the fundamental or an overtone.

instantaneous frequency, the frequency evaluated at a specific time t of the time wave; this is the time derivative of the instantaneous phase.

carrier frequency, a term initially used in telecommunications for the frequency of the main sinusoidal waveform (carrier wave or signal wave) used to convey information.

A harmonic series results from the addition of a fundamental and several harmonics. The amplitude, or importance, of each harmonic can be set by a multiplying factor or weight A_i (Fig. 2.10). The following harmonic series sound is the result of a fundamental and the addition of three harmonics:

$$\begin{aligned} s(t) &= (A_0 \times f_0) + (A_1 \times f_1) + (A_2 \times f_2) + (A_3 \times f_3) \\ &= (A_0 \times f_0) + (A_1 \times 2 \times f_0) + (A_2 \times 3 \times f_0) + (A_3 \times 4 \times f_0) \end{aligned}$$

A harmonic cannot appear if it has a null amplitude; for instance, the harmonic f_2 is no more produced in the following sound wave:

$$s(t) = (A_0 \times f_0) + (A_1 \times f_1) + (A_3 \times f_3)$$

Similarly, the fundamental frequency cannot be apparent resulting in sound made of a series of harmonics only:

$$s(t) = (A_1 \times f_1) + (A_2 \times f_2) + (A_3 \times f_3)$$

In that case, the fundamental can be determined by finding the greatest common divisor of f_i or by simply computing the difference between f_{i+1} and f_i. A sound made of two harmonics $f_1 = 400\,\text{Hz}$ and $f_2 = 600\,\text{Hz}$ has a fundamental frequency $f_0 = f_2 - f_1 = 600 - 400 = 200\,\text{Hz}$.

Any combination of harmonics and inharmonics can be recorded or synthesized, but the sound sources of animals and instruments usually produce typical harmonics series.

So far, we considered only sine wave, but waves might follow a square, triangle, or sawtooth periodic function as shown in Fig. 2.11. These waves are actually not seen in animal vocalizations, but they still can be useful for synthesis purposes (see Sect. 18.3). As they are not sine waves, these functions do not consist of a single frequency component but of a series of harmonics. The sawtooth function

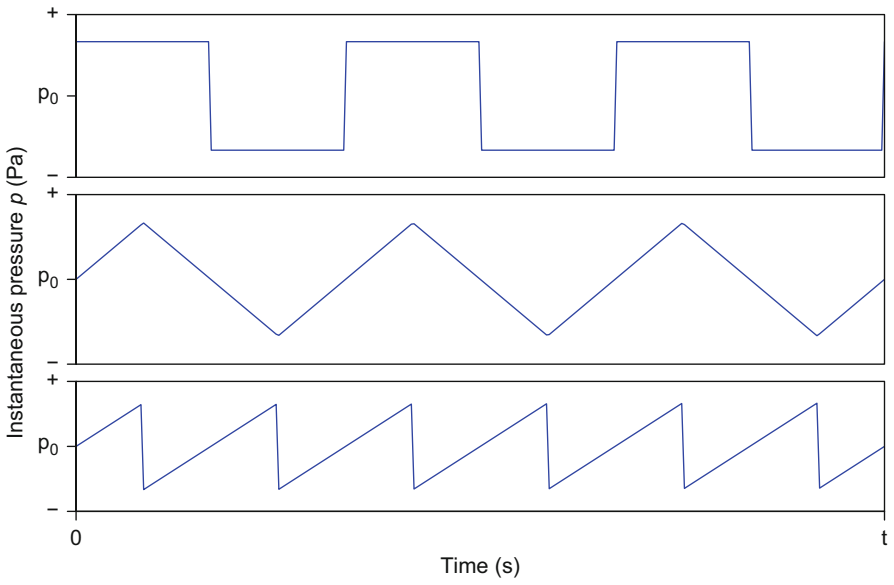


Fig. 2.11 Square (top), triangle (middle), and sawtooth (bottom) waves. These periodic functions consist of harmonics series

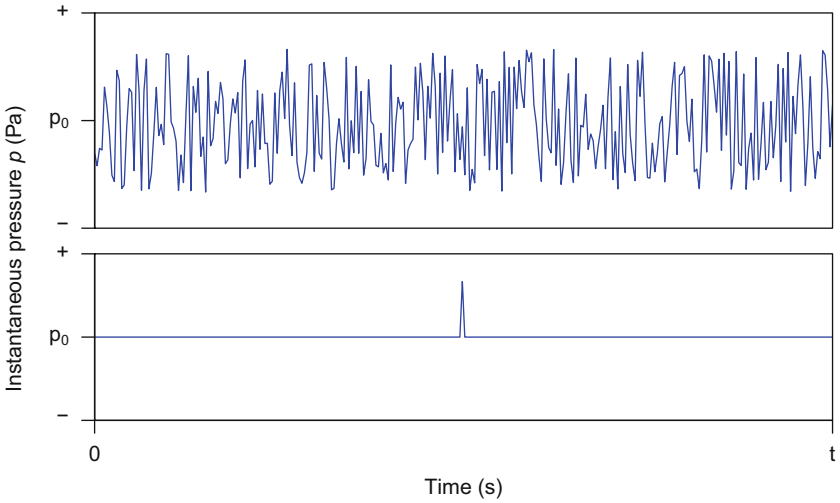


Fig. 2.12 Noise (top) and Dirac pulse (bottom) waves. These functions do not produce either harmonics or inharmonics overtones

is, for instance, made of a harmonic series with amplitudes steadily decreasing. Some sounds are not built with periodic functions. This is particularly the case of a random wave, or noise *sensu stricto*, and Dirac pulse (Fig. 2.12). These sounds do not contain any frequency-structured components.

2.2.7 Writing Sound with a Simple Equation

Sounds are nice to hear and to visualize as waves, but they are also wonderful when written with maths. Now that we know all parameters describing a simple sine sound wave, we can write the pressure equation to generate it. It is an elegant sine formula:

$$\begin{aligned} p(t) &= A \times \sin(\omega t + \varphi) \\ &= A \times \sin\left(\frac{2\pi}{T}x + \varphi\right) \\ &= A \times \sin(2\pi f t + \varphi) \end{aligned}$$

The sound produced by the 440 Hz tuning work with a peak amplitude of 2×10^{-2} Pa with a $\pi \div 4$ rad phase is written as follows:

$$p(t) = 2 \times 10^{-2} \times \sin\left(2\pi \times 440 \times t + \frac{\pi}{4}\right).$$

If we wish to add an H_1 harmonic to this sound with an amplitude half of the fundamental of first harmonic H_0, we could simply apply the following addition:

$$p(t) = 2 \times 10^{-2} \times \sin\left(2\pi \times 440 \times t + \frac{\pi}{4}\right)$$
$$+10^{-2} \times \sin\left(2\pi \times 880 \times t + \frac{\pi}{4}\right)$$

2.2.8 Amplitude and Frequency Modulations

The amplitude and the frequency of a sound wave can vary with time. The instantaneous amplitude can increase and decrease following any pattern, periodic or not. Similarly, the instantaneous frequency can change over time, in a regular pattern or not. These variations are defined as amplitude and frequency modulations, abbreviated AM and FM. AM and FM can covary or follow independent patterns.

To apply an amplitude modulation to a wave, the maximum amplitude value A has to be replaced or multiplied with a function $a(t)$, the instantaneous amplitude, describing the evolution of the instantaneous amplitude a with respect to time t. For instance, a sound wave with an exponential decrease with a -0.4 factor can be written as (Fig. 2.13 top):

$$p(t) = A \times e^{-0.4 \times t} \times \sin\left(2\pi \times 440 \times t + \frac{\pi}{4}\right)$$

When the amplitude modulation is sinusoidal, that is, when A is replaced or multiplied by a cosine or sine function, the amplitude modulation is then defined by its rate or frequency f_{am}, its phase φ_{am}, and its depth m also called modulation rate that varies between 0 (no modulation) to 1 (or 100 if expressed in percentage) (Fig. 2.14):

$$p(t) = A \times \cos(1 + m \times 2\pi \times f_{am} \times t + \varphi_{am}) \times \sin\left(2\pi \times 440 \times t + \frac{\pi}{4}\right)$$

To generate a frequency modulation, a function $f(t)$ that sets the changes of the instantaneous frequency f with respect to time has to be added in the sine function. For instance, an exponential frequency increase is obtained with (Fig. 2.13 bottom):

$$p(t) = A \times \sin\left(e^t + 2\pi \times 440 \times t + \frac{\pi}{4}\right)$$

When the frequency modulation is sinusoidal, a cosine or sine function is added. The frequency modulation is then defined by its own frequency f_{fm}, its phase φ_{fm}, and its modulation index β (also named I). The modulation index β is the ratio of

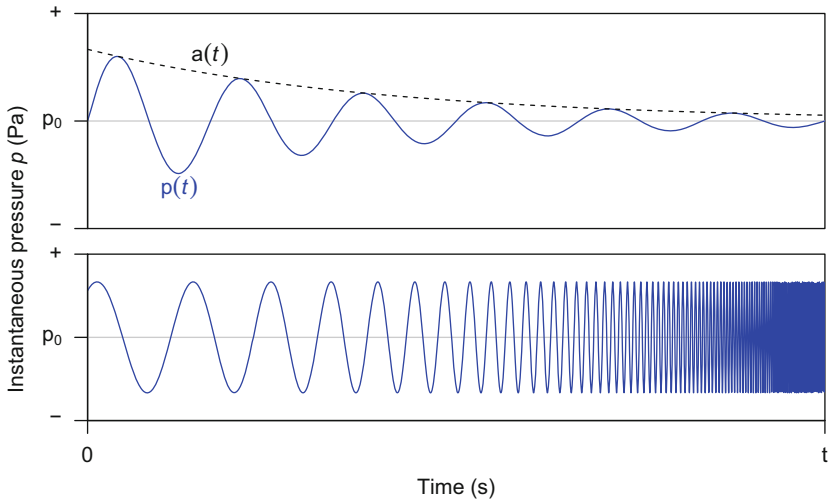


Fig. 2.13 Amplitude and frequency modulations (AM, FM). The instantaneous amplitude (blue plain line) is modulated according to an amplitude exponential decay $a(t)$ (black dashed line) (top) or according to a frequency exponential increase $f(t)$ (bottom)

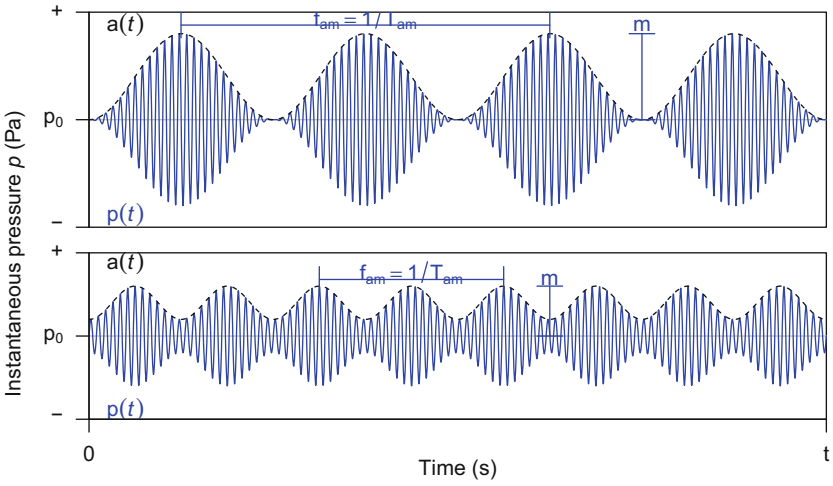


Fig. 2.14 Sinusoidal amplitude modulation. Two examples of instantaneous amplitude (blue plain line) modulated according to a sinusoidal amplitude modulation $a(t)$ (black dashed line). The frequency of the amplitude modulation f_{am} of the above example is half the one in the example below. The amplitude depth m is 1 (or 100%) in the example above and 0.5 (or 50%) in the example below

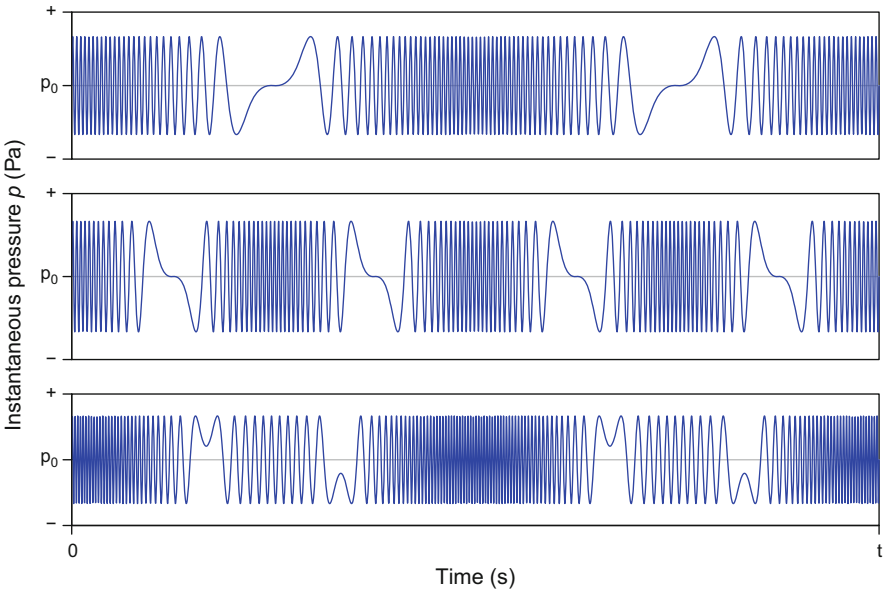


Fig. 2.15 Sinusoidal frequency modulation. Three examples of sinusoidal frequency modulations $f(t)$: a frequency modulation with a frequency of 2 and a modulation index of 50 (top), a frequency modulation of 4 with a similar modulation index of 50 (middle), and a frequency modulation of 2 with a modulation index of 100

the peak frequency deviation δf to the frequency of the FM (Fig. 2.15):

$$\beta = \frac{\delta f}{f_{fm}}$$

The equation of the sound becomes:

$$p(t) = A \times \sin\left(\beta \times \sin(2\pi \times f_{fm} \times t + \varphi_{fm}) + 2\pi \times 440 \times t + \frac{\pi}{4}\right)$$

Finally, both amplitude and frequency modulations can combine in a single wave, as in:

$$p(t) = A \times e^{-0.4 \times t} \times \sin\left(\beta \times \sin(2\pi \times f_{fm} \times t + \varphi_{fm}) + 2\pi \times 440 \times t + \frac{\pi}{4}\right)$$

2.3 Sound as a Time Series

A time series is a collection of data that have been observed at different points in time. For instance, monthly measurements of ambient carbon dioxide CO_2 concentration constitute a time series (Fig. 2.16). Looking at this famous time series suggests immediately a homology with sound. The CO_2 concentration could be replaced by air pressure, and the time series would be a (nice) sound. Indeed, when recording sound, we observe the pressure of the medium at regular time intervals just like measuring the CO_2 concentration monthly.

Sound is fundamentally a time series, and many time series could be converted into sound. A nice example of this property of sound is to be found in neurophysiology. The electrical activity of a nerve or a single neuron can be recorded with an adapted electrode and voltage amplifier. The action potential of the cell or the group of cells generates a time series with voltage as the measured data. This time series is often converted live during experimentation such that the experimenter can monitor the electrical activity by literally hearing the neuron or nerve. When working on animal hearing, you can even listen what the auditory nerves or neurons "hear." Time series are extremely common so that the statistics behind are important.

One of the main properties of time series is the autocorrelation of the data; the value measured at the time point $t + 1$ depends on the value observed at the previous time point t. Time series analysis uses a few tools shared with acoustics, in particular (auto)correlation, filtering, and spectral analysis that consist in identifying the periodicity (frequency) of the time series. However, the fundamental concepts of time series are rarely found in acoustics like the study of stochastic and stationary processes or the prediction of time series through statistics models. Getting into

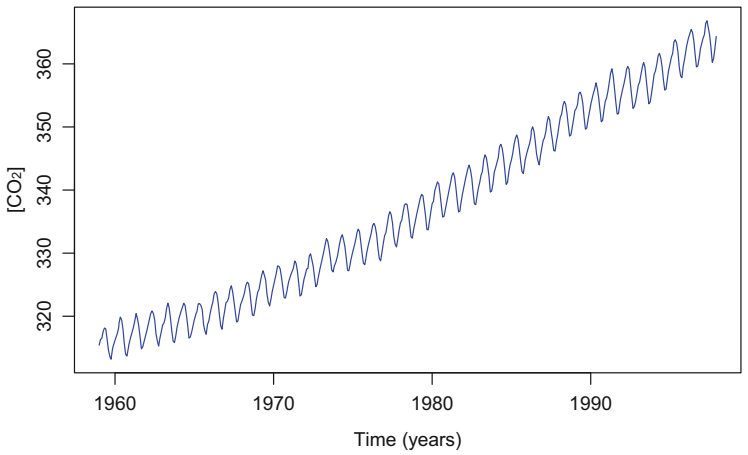


Fig. 2.16 Example of a time series. The atmospheric concentrations of CO_2 expressed in parts per million (ppm) from 1960 to 1997. This dataset could be transformed into a sound. Data from the package `datasets`

the details of time series may help to understand and, more importantly, to analyze the time series obtained when parametrizing a series of recordings achieved along time. It could be therefore useful to consult books dedicated to time series with R as Shumway and Stoffer (2006) and Cryer and Chan (2008).

2.4 Sound as a Digital Object

To analyze sound we need first to record it and to save it as a digital object. Sound pressure fluctuations have to be converted into binary items that could be saved, analyzed, and potentially modified. The conversion of the sound into a digital file is achieved through a recording chain that involves (1) a sensor—a microphone when recording in the air, a hydrophone when recording in water, a vibrometer when recording on a solid—and (2) a digital recorder with an amplifier and a digital storage unit. We will not detail the electronics behind the digital recording chain or digitization, but we will review the main principles of sound digital sampling and quantization that are quite important for avoiding errors when analyzing digital sound. We will also introduce the main audio file formats.

2.4.1 Sampling

Digital recording is a discrete process of data acquisition. The process of converting an analogue signal into serial binary data is called pulse code modulation (PCM). Sound is recorded through regular samples. These samples are taken at a specified rate, named the sampling frequency or sampling rate f_s given in Hz or kHz. The most common rate is 44,100 Hz (or 44.1 kHz), but lower rate can be used for low-frequency sound (e.g. 22.05 kHz), or higher rate can be used for high-frequency sound (up to 192 kHz or even higher).

Figure 2.17 shows 5 ms of a pure tone sound at 440 Hz sampled at two different sampling frequencies.

2.4.2 Quantization

Another important parameter of digitization is the process of quantization, or digitization depth, that consists in assigning a numerical value to each sample according to its amplitude (Fig. 2.18). These numerical values are attributed according to a bit scale. A quantization of 8 bit assigns amplitude values along a scale of $2^8 = 256$ states around 0 (zero). Most recording systems use a $2^{16} = 65{,}536$ bit scale.

Quantization can be seen as a rounding process. A high-bit quantization will produce values close to reality, i.e., values rounded to a high number of significant

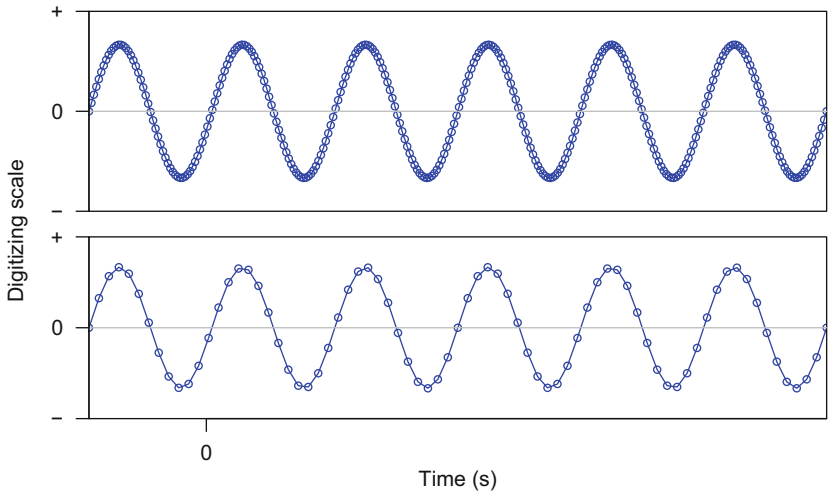


Fig. 2.17 Sampling. Digital sound is a discrete process along the time scale. The same wave is sampled at two different rates: the wave above is sampled four times more than the bottom wave. Each point is a sample; the line is original continuous sound

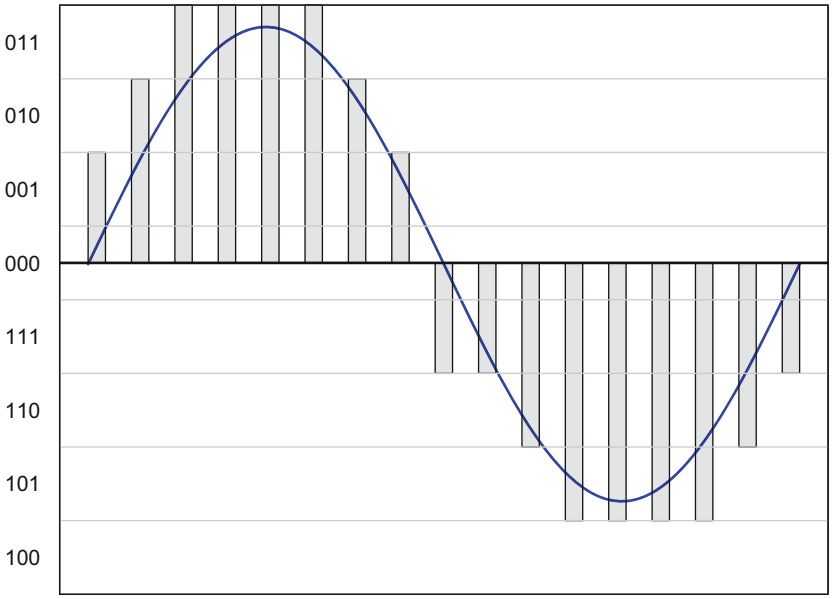


Fig. 2.18 Quantization. Digital sound is a discrete process along the amplitude scale: a 3 bit ($= 2^3 = 8$) quantization (gray bars) gives a rough representation of a continuous sine wave (blue line)

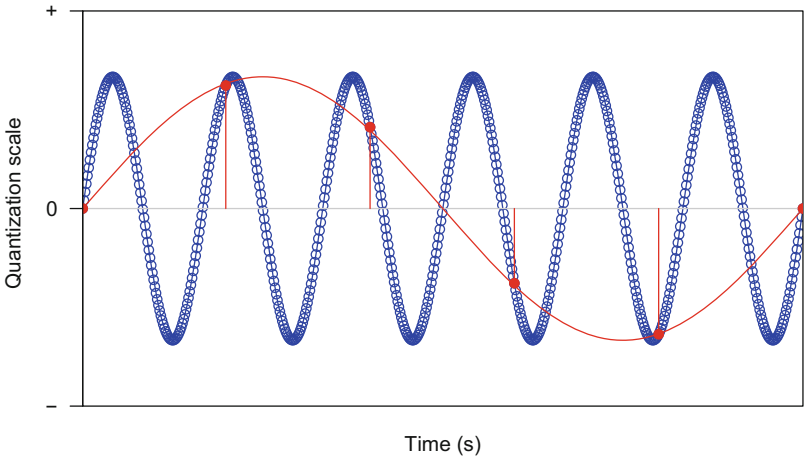


Fig. 2.19 Aliasing on a sine wave. In blue, the original sine wave was sampled at an appropriate rate representing well the cycle period or frequency. In red, the same sine wave sampled at a too low rate generating *aliasing* at a lower wrong frequency

digits, when a low-bit quantization will produce values far from reality, i.e., values rounded a low number of significant digits. Low quantization can impair the quality of the wave.

2.4.3 Issues in Sampling and Quantization

In order to represent properly the wave, it is advised to follow the Nyquist-Shannon sampling theorem which stipulates that at least two samples must be taken per audio cycle (Shannon 1949). This means that the sampling frequency f_s should be at least twice as high as the highest frequency of the wave also known as the Nyquist frequency f_N. Departing from this rule may induce frequency artifacts known as aliasing. If there are components in the wave at frequencies higher than the Nyquist frequency, then they appear at wrong frequencies. Figure 2.19 illustrates this issue with a sine wave sampled at an inadequate rate. The frequency of the original wave is corrupted into a lower frequency.

Figure 2.20 depicts another side effect of aliasing on a complex wave. The digitized wave can be quite different from the original wave, such that an inappropriate sampling might lead to biased results (see Sect. 6.1).

The two previous examples show how undersampling may affect the quality of a sound. However, oversampling during digitization may also generate important

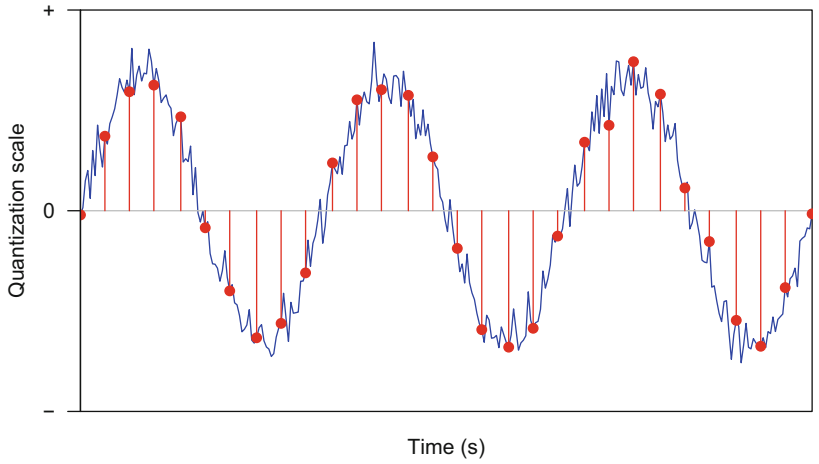


Fig. 2.20 Aliasing on a complex wave. The original blue wave is a complex wave including several frequency components. When sampled at an appropriate rate, the wave can be properly represented with all small amplitude changes. However, when sampled at a low rate, the main amplitude features are lost (red dots and red segments)

frequency artifacts. Oversampling indicates that sound was digitalized at a rate well above the Nyquist frequency, as sampling at 192 kHz when no frequency occur above 22 kHz. In the frequency domain, oversampling generates artificial copies of the frequency content of the sound. These frequency bands may be misinterpreted as high-frequency sound production. However, simple filtering processes can remove these unwanted frequencies and clean the oversampled sound.

Distortion can occur as well when the digitized wave is clipped with values outside the quantization scale (Fig. 2.21). Clipping can occur during recording due to a too high recording level and/or when digitizing at an incorrect level. Such bad sound acquisition conditions obviously induce errors when describing the amplitude and temporal features of the wave, but clipping also generates false harmonics and therefore can lead to wrong sound interpretation. As clipping cannot be removed, squared waves should be considered very carefully before undertaking any analysis.

2.4.4 File Format

There are multiple file formats to save sound. `R` can handle three categories of audio files:

`.wav` uncompressed format, the full information is stored in a heavy file. The format can be either PCM with data scaled along a 2^n scale with integer values or along an IEEE floating scale, i.e., along a $[-1, 1]$ scale with numeric values,

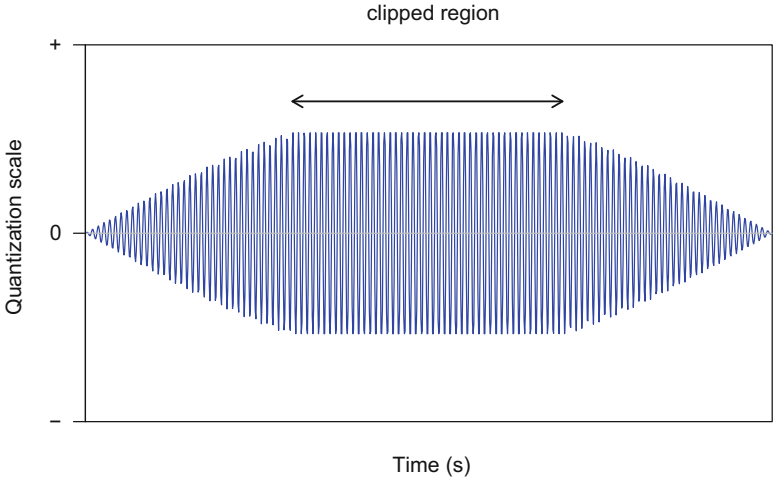


Fig. 2.21 Clipping. This wave was not properly acquired. The amplitude exceeds the limits of the quantization scale leading to a squared or flat waveform (arrow). Such waveform cannot be studied properly as amplitude, time, and frequency features are distorted

`.mp3` lossy compressed format, the information is reduced. Time, amplitude, and frequency parameters can be impaired. This format should be avoided for scientific studies which require full acoustic information,

`.flac` losslessly compressed format, the full information is stored in a reduced size file.

All these formats generate binary files, sound being encoded into a succession of 0 and 1. When importing these formats into `R`, the data can be transformed into a decimal format inducing an important increase in data size.

2.5 Sound as a Support of Information

So far, we described sound only in terms of shape but not in terms of content or meaning. Sound can be a **signal** embedding a certain amount of information, or sound can be **noise** that does not convey any information. Sound can be therefore the main material of a communication system between a transmitter and a receiver. These elements draw the diagram of communication as proposed by Shannon and Weaver (1949) shortly after the second world war (Fig. 2.22). Basically, the story is always the same: a transmitter emits a signal that encodes some information and that is transported by a medium or channel. The signal full of information reaches an aware receiver that decodes the information. This scheme is true for any communication system and is therefore used repeatedly for animal communication (Fig. 2.23). In this case, the emitter is a singing animal, like a bird tweeting or

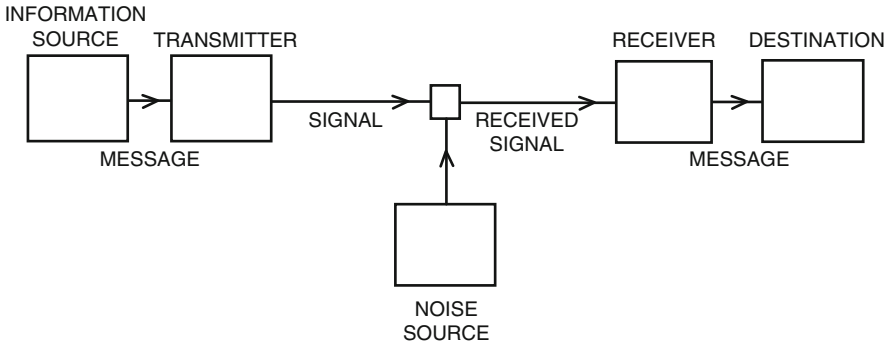


Fig. 2.22 Shannon diagram of a communication as published in Shannon (1949) and Shannon and Weaver (1949)

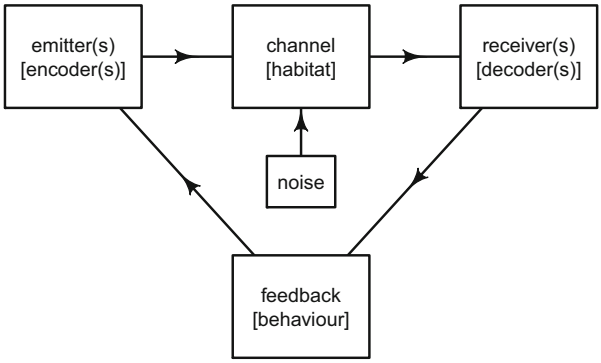


Fig. 2.23 Shannon diagram adapted to animal communication system. Drawn with the package `diagram` (Soetaert 2014)

an insect stridulating, the channel can be air, water, soil, or plant tissues, and the receiver is a hearing animal ready to react. Noise coming from the environment—the wind in the forest, the noise of boat in the ocean, the rain dropping a plant leaves—can impair the transfer of information and degrade or even cancel the information process between animals. When the connection is successful, the receiver may show a response or feedback to the emitter. This response can be obvious and fast as a movement toward or away from the emitter, or it can be conspicuous and delayed as a physiological state change (Fig. 2.23).

The nature and the amount of the information conveyed by a signal are highly variable. It depends on the nature of the dyad emitter-receiver and of the context of the communication act. Hence, information quantity, measured in bits along a logarithmic scale, can vary considerably whether you consider the introductory talk of a Nobel prize or the buzz of a flying mosquito.

The story schematized by Shannon sheds light on a third story character: noise. Noise can degrade the signal during its travel in the channel from the emitter to

the receiver reducing the information amount. Noise is often viewed as a random event producing a stochastic wave. However, if we refer to Shannon, noise is not necessarily random; it is a source of perturbation. This means that a pure tone can be noise if it scrambles the signal that aims at connecting an emitter and a receiver. The most satisfactory definition of noise could be *any unwanted sound*, as defended by Schafer (1977). The definition of noise should be a relative definition, in reference to the receiver hearing and decoding process, not in reference to the emitter encoding and producing system. Noise can be a signal for a receiver and noise for another one. The siren of an ambulance is usually perceived as a relief signal of an arriving aid for a victim but could be considered at the same time as noise by local residents.

The diagram of Shannon is extremely constructive as it builds a simple linear chain of events that are clearly identified and delimited (emission, propagation, noise, reception). This clarity explains why numerous studies in animal behavior refer, directly or indirectly, to the emitter-receiver paradigm. However, this scheme could also be viewed as a rather narrow concept as it suggests that communication simply works as a closed system between two individuals that share an encoding-decoding process. Several studies clearly revealed that animal communication is structured in a network rather than in a one-to-one communication relation (McGregor 2005). The emitter and the receiver rarely form an isolated pair, but rather they combine into a piece of a communication web spun by several individuals. These individuals can alternatively play the roles of emitter and receiver and thus generate signal and noise, depending on the receiver point of view, or better said point of listening.

Chapter 3
What Is R?

3.1 A Brief Introduction to an Ocean of Tools

Here are some important historical and technical facts about R for the reader who has never used R before.

R Is Not That New R was born in 1996 in New Zealand. The fathers of R, **R**oss Ihaka and **R**obert Gentleman, were both working at the University of Auckland. As described in their seminal paper (Ihaka and Gentleman 1996), Ihaka and Gentleman conceptualized a computer language for statistical data analysis influenced by two existing languages, namely, S and scheme. If R has the main appearance of S and is therefore often considered as a dialect of S, it has the implementation and the semantics of scheme. Ihaka and Gentleman aimed at coining a new language that "provides advantages in the areas of portability, computational efficiency, memory management, and scoping."

R Is a Language for Statistics and Related Disciplines R was originally considered as a language for statistical computing and graphics. However, R is much more than that as it now covers the needs of almost all scientific disciplines of mathematics, physic, computer, life, medical, and linguistic sciences. The use of R exponentially increased during the last 10 years, in both academic and commercial worlds with R examples found in almost all sorts of data analyses. The success of R mainly comes from its free access, from modularity, and, above all, from the coaction of users who permanently and all around the world participate to its development creating an ocean of tools.

R Is Free and Open-Source R is free to download and to install being declared under the terms of the Free Software Foundation's GNU General Public License in source code form. The main term of this license is "to guarantee your freedom to share and change free software—to make sure the software is free for all its users." R can be therefore installed on any computer, and most of its functionalities can

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_3

be inspected, modified, and redistributed. The source program is maintained by a core team, but any user can freely contribute to R by submitting accessory tools or packages. This way to contribute to R is experimenting a tremendous success with thousands of contributions available. These tools open the possibility to the user to customize R to his/her own needs. R can therefore work in very different ways in neighboring laboratories or offices. The dark side of R contribution process is that there is only a form check but not a content control. This means that a user contribution should run and comply with the last version of R but that the actions of the new functions are not peer-reviewed. This makes sailing in R ocean rather dangerous even if the risk of error is counterbalanced by user feedback following the main principle of collaborative projects. The plethora of packages also induces a redundancy and some inconsistency in R methods so that users can get lost in the middle of the ocean.

R Is an Object-Oriented Language Like many other programming languages (C++, Java, Javascript, Python, Perl), R paradigm is based on the concept of objects that have specific features or attributes. These objects are associated with methods, or functions, that can act on the objects. The concept of object is rather simple to understand as it more or less consists of data placed in a box with information (attributes) written on the box. The box can be combined or changed with functions. This way of programming has some constraints as the objects have to be clearly defined or typed, but this also ensures to properly apply methods to right objects preventing data corruption or unsuited analyses.

R Is a High-Level Interpreted Language Instructions are directly executed without the need of a program compilation. The instructions are given with written commands in a prompt console. This implies that R can be invoked from a command terminal without any graphical interface. This way to give instructions to a program might appear antiquated in these times of smartphones and tablets controlled with the tip of an index. However, writing commands is much more efficient than mouse clicking or screen touching. There is a single main reason for this efficiency: commands can be repeated. When using a graphical interface that drives the user from a menu to a sub-menu to another sub-sub-menu with a dozen of options, it is almost impossible to reproduce exactly the same procedure. There is no way to save all the instructions given through ticking and option selection. However, a command, and better a list of commands building up a script, can be saved in a few kilooctet file. This file can be tested several times, used at different times of a project, and shared with colleagues for collaborative work without any confusion. R scripts written several years ago still work and can be repeated without any issue ensuring an ascending compatibility. R ensures therefore a perfect reproducibility of data analysis, one of the mainstays—with refutability—of science. For those who might still be reluctant to written commands, just think that R language is the solution to translate a long series of fastidious mouse clicks into a simple series of clear commands. An R command can be seen as a shortcut of long series of visual operations. As an example, plotting an histogram of a dataset named `x` with a spreadsheet software takes about seven mouse clicks difficult to remember, whereas

it requires an easy stand-alone command (`hist(x)`) with R. This command will not change when the route to the histogram within the graphical interface of the spreadsheet will probably differ with software version.

R Is an Easy Language to Learn Learning R is not that difficult; it is like learning a foreign language that would include a dozen of words and a few syntax rules only. When it is necessary to stay abroad to properly learn a foreign language, learning R does not need any travel as there are R speakers all around the world communicating in hundreds of blogs, mail discussion lists, and local user groups. Help is everywhere.

R Is Not Perfect R is often introduced as a perfect tool. However, the main weakness of R is to be found in memory management as already mentioned by Ross and Ihaka at the very early stages of R (Ihaka and Gentleman 1996). All R objects are stored on the random-access memory (RAM) that can be quickly full when handling large datasets making R very slow. In a very interesting self-criticism exercise, Ihaka confirmed this memory problem and also pointed out that R interpreter was not fast enough and that vectorization (one of the main paradigm of R, see Sect. 3.3.7) can introduce computational overheads when scalar processes would be more efficient. It seems that these problems might be difficult to solve in a short time (Ihaka 2010).

3.2 How to Get R

R is composed of a core program or **base** system and additional accessory programs or contributed **packages** that can be installed on a wide range of operating systems including Windows, Mac, and Linux as soon as the user has the administration rights on his/her machine. All installation instructions of the base are available on R web site.[1] R base is available online at the Comprehensive R Archive Network or CRAN[2] with different geographical mirrors such that download time is optimized. The installation is straightforward so that there is no counterargument to install R on a computer!

R base comes with numerous functions and embedded packages, but specific analyses quickly requires the installation of additional packages. Packages are also stored at CRAN and can be downloaded and installed manually, but R offers the possibility to install the packages directly from the console with the following command:

```
install.packages("MASS")
```

[1] http://www.r-project.org/
[2] http://cran.r-project.org/

This command will download from CRAN the compressed folder or **package** MASS (Modern Applied Statistics with S) (Venables and Ripley 2002), uncompress, and install it on your computer. Once uncompressed the package becomes a **library** that can be made available locally with:

```
library(MASS)
```

Main R graphical user interface (GUI) is not very friendly. It might be useful to install other GUIs that will make the use of R more fancy. Several GUIs have been developed but RStudio[3] has a nice and complete design and is easy to handle. The open-source version of RStudio can be freely installed.

So the complete process to have a nice R version on your system is to (1) install R [necessary], (2) install a GUI-like RStudio [optional], and (3) install packages [optional].

3.3 Do You Speak R?

3.3.1 Where Am I?

When starting a session, R sets automatically a default working directory. A working directory is a folder where R will either read or write files if no other file localization instructions are given. The function (see Sect. 3.3.4) `getwd()` returns the complete path to the working directory:

```
getwd()
```

In most cases, this default directory is not very convenient so that a new default directory should be set. This can be done with the function `setwd()`. The path to give to `setwd()` will change according to the operating system:

```
setwd("complete-path/to-directory")
```

The function `dir()` returns the list of the files saved in the working directory.

[3] http://www.rstudio.com/

```
dir()
```

3.3.2 Objects

An object is a structure with a name that contains data, which is either imported into R or generated within an R session. For instance, the following commands generate and print (display) an object named num containing the numeric values 10, 11, 12, and 13:

```
num <- 10:13
num
[1] 10 11 12 13
```

This is one of the most simple objects, but there are numerous objects differing by their format and content. To avoid error by applying a function to a wrong object, it is crucial to know the identity of the object. The identity is defined by the class and the attributes of the object.

3.3.2.1 Classes

There are six major classes of objects:

`numeric vector` is a one-dimensional object consisting in a succession of items or cells containing numbers. The object num created just above is a vector:

```
num
[1] 10 11 12 13
```

`character vector` is a one-dimensional object consisting in a succession of items or cells containing characters. Here is a character vector with the first three letters of the alphabet:

```
char <- c("a", "b", "c")
char
[1] "a" "b" "c"
```

`factor` is a one-dimensional object consisting in a succession of items or cells with nominal or ordered categories. Here is a factor with two categories or levels:

```
fact <- factor(c("Male","Female", "Female"))
fact
[1] Male   Female Female
Levels: Female Male
```

`matrix` is a two-dimensional object with rows and columns. A matrix can contain numeric values only. Here is a matrix with 2 rows and 2 columns:

```
mat <- matrix(10:13, nrow=2)
mat
     [,1] [,2]
[1,]   10   12
[2,]   11   13
```

`array` is a (p, q, n) dimension object. An array can vary from a one-dimensional object (vector like) to a collection of n matrices of dimension (p, q). Here is an array consisting of 3 matrices with 2 rows and 2 columns each:

```
ary <- array(10:21, dim=c(2,2,3))
ary
, , 1

     [,1] [,2]
[1,]   10   12
[2,]   11   13

, , 2

     [,1] [,2]
[1,]   14   16
[2,]   15   17

, , 3
```

(continued)

```
     [,1] [,2]
[1,]   18   20
[2,]   19   21
```

`list` is an object consisting of an ordered collection of objects known as its components. A list can be viewed as a bag where any items can be placed in. Here is a list containing three items, two objects previously generated (`num` and `mat`) and an additional one containing the sentence `"Bye bye moon"`:

```
lst <- list(x=num, y=mat, z="Bye bye moon")
lst
$x
[1] 10 11 12 13

$y
     [,1] [,2]
[1,]   10   12
[2,]   11   13

$z
[1] "Bye bye moon"
```

`data.frame` is a special list in which all elements are vectors of equal length. It is basically a table with variables as columns and observations as rows. Here is a table with 3 columns and 4 lines:

```
df <- data.frame(x=10:13,
                 y=c("A", "B", "C", "D"),
                 z=c(TRUE, FALSE, FALSE, TRUE))
df
   x y     z
1 10 A  TRUE
2 11 B FALSE
3 12 C FALSE
4 13 D  TRUE
```

3.3.2.2 Attributes

Each object has one or more attributes, or metadata, that define data features:

names are labels of the different elements of an object. These names can be used to extract the elements. See Sect. 3.3.6.1,

Table 3.1 Type, mode, class and, dimensions of R objects

Object	Type	Mode	Class	Dimensions
vector `num`	integer	numeric	integer	NULL
vector `char`	character	character	character	NULL
factor `fact`	integer	numeric	factor	NULL
matrix `mat`	integer	numeric	matrix	(2, 2)
array `ary`	integer	numeric	array	(2, 2, 3)
list `lst`	list	list	list	NULL
data.frame `df`	list	list	data.frame	(4, 3)

Type, mode, class and dimensions of the R objects generated so far. See text for details

mode is the nature of the data stored in the object. It can be numeric (numbers: `12, 0, -4.1, 3.14`), complex (complex numbers: `3 + 2i`), logical (`TRUE/FALSE`), character (text: `"hello"`), or raw (bytes: `001011`),

type is very similar to `mode` except that a `numeric` is splitted into `integer` and `double`,

class for simple vectors is just the mode (numeric, complex, logical, character, raw); for other objects it can be `matrix`, `array`, `factor`, or `data.frame`,

dimension integers specify the respective extents of the object. They are `NULL` for one-dimensional objects.

It is not always easy to guess the properties of an object. There are a few functions that can help: `typeof()` to know the type, `names()` to get the names of the different components, `mode()` to know the mode, `class()` to identify the class, and `dim()` to obtain the dimensions (Table 3.1). The function `attributes()` provides the main attributes. There are no attributes for a `vector`:

```
attributes(num)
NULL
attributes(char)
NULL
```

The attributes of a `factor` provide the `class` and the different levels (or states) of the factor; here the gender states:

```
attributes(fact)
$levels
[1] "Female" "Male"

$class
[1] "factor"
```

The attributes of a `matrix` and of an `array` are the object dimensions:

```
attributes(mat)
$dim
[1] 2 2
```

```
attributes(ary)
$dim
[1] 2 2 3
```

The attributes of a `list` are the names of the `list` items:

```
attributes(lst)
$names
[1] "x" "y" "z"
```

The attributes of a `data.frame` are the names of the columns (corresponding to the names the items of a `list`), the names of the rows, and the class of the object:

```
attributes(df)
$names
[1] "x" "y" "z"

$row.names
[1] 1 2 3 4

$class
[1] "data.frame"
```

The function `str()` is a very helpful tool that details the structure of the object providing several information, for instance:

```
str(df)
'data.frame': 4 obs. of  3 variables:
 $ x: int  10 11 12 13
 $ y: Factor w/ 4 levels "A","B","C","D": 1 2 3 4
 $ z: logi  TRUE FALSE FALSE TRUE
```

This indicates that the object `df` is a `data.frame` with 4 rows (observations) and 3 columns (variables). The first column is named `x` and contains integers, the second column is named `y` and is a factor with four categories or levels, and the last column named `z` contains logical data.

3.3.3 Operators

The arrow `<-` is an R operator to place, or to assign, values into an object.

There are many other operators in R that can be classified in three main classes: indexing operators that are used to select a specific part or item of an object (see Sect. 3.3.6.1), arithmetic operators for common operations, and logical operators to compare values or assess conditions. R operators are summarized in Table 3.2.

Table 3.2 R operators

Operator	Description
Arithmetic	
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
Logical	
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x | y	x or y
x & y	x and y
Indexing	
[	1- or 2-dimension index
[[	List index
$	Component selection
@	Slot selection
::	Access variable in a name space

Main R operators to be used for arithmetic operations, logical conditions, and object indexing. See text for details

3.3.4 Functions

A function is an R instruction to be applied to an object. The behavior of the function may differ according to the attributes of the object. The function `mean()` computes and returns the arithmetic mean:

```
mean(num)
[1] 11.5
```

A function has a name (`mean`), is delimited with brackets (`()`), and has arguments. The arguments can be viewed as the options of a function. For instance, the function `mean()` has three arguments as detailed by the `mean` documentation:

`x` an R object, the data to be averaged.
`trim` an argument (option) to remove outliers in `x` that may bias the arithmetic mean. This argument has to be set with a numeric value between `0` and `0.5`
`na.rm` an argument (option) to indicate whether non-available or missing data (`NA`) should be removed. This argument has to be set with a logical value; it can be set to `TRUE` to remove the `NA` from `x`.

The name of a function can be very short as it is the case of one of the most used function named `c()`. This single letter function concatenates objects or values (see Sect. 3.3.6.3 for more concatenating functions). To update `num` by concatenating it with an outlier value of 50 and missing data, we run:

```
num <- c(num, 50, NA)
num
[1] 10 11 12 13 50 NA
```

Now, we can play with the arguments of `mean()` to trim the data to get rid of the value `50` and to exclude missing data:

```
mean(num)
[1] NA
mean(num, na.rm=TRUE)
[1] 19.2
mean(num, trim=0.5, na.rm=TRUE)
[1] 12
```

All function arguments and other details including examples are provided in a help page that can be obtained with a question mark immediately placed before the name of the function which is a shortcut for the function `help()`:

```
?mean
help(mean)
```

When encountering a new function, particularly in this book, it is highly recommended to consult the corresponding help page and to run the examples.

The function `args()` can also be handy as it returns the arguments and the default settings of a function. For instance, for the function `sd()`, which returns the standard deviation, we have:

```
args(sd)
function (x, na.rm = FALSE)
NULL
```

Several lines but several commands can be placed on a single line as soon as the commands are separated with a semicolon "`;`". This can be used to shorten a script:

```
num <- 10:13 ; mat <- matrix(10:13, nrow=2)
```

R has thousands of functions spread out in the base (Table 3.3) and packages. This ocean of tools has, nonetheless, limits, and users quickly need to write their own function for their specific needs. This is when R is beautiful, the magic time when a user turns into a designer. To write a function is extremely simple: choose a name, a list of arguments, and cook it! Let's start with an extremely simple function: compute the multiplicative inverse or reciprocal of a numeric value, that is, compute and return $1 \div x$. We name the new function `inverse`. The function, which has a single argument `x`, is written with a single line command:

```
inverse <- function(x) 1/x
```

Table 3.3 Fundamental R arithmetic and statistic functions

Function	Description
`min()`	Minimum
`max()`	Maximum
`sqrt()`	Square-root
`sum()`	Sum
`median()`	Median
`quantile()`	Distribution quantiles
`mean()`	Arithmetic mean
`var()`	Variance
`sd()`	Standard deviation
`log()`	Natural logarithm
`log10()`	Base 10 logarithm
`sin()`	Sinus
`cos()`	Cosinus
`tan()`	Tangent

We can use it immediately:

```
inverse(4)
[1] 0.25
```

It works! We can now try to design a function, a slightly more complex, that converts degrees Celsius in degrees Fahrenheit. The conversion follows:

$$°\mathrm{C} = \frac{(°\mathrm{F} - 32) \times 5}{9}$$

$$°\mathrm{F} = \left(°\mathrm{C} \times \frac{9}{5}\right) + 32$$

We can write a function named `f2c` taking two arguments. The first argument `x` is the temperature in °F, and the second argument is logical to optionally compute the reciprocal function, that is, to convert °C in °F. Because the function will include more than one line of code, we need to frame it with curly brackets { }. Here is the function:

```
f2c <- function(x, reciprocal=FALSE){
  if(reciprocal == FALSE) {res <- (x-32)*5/9}
  else {res <- (x*9/5)+32}
  return(res)
}
```

and a test:

```
f2c(70)
[1] 21.11111
f2c(21.11111, reciprocal=TRUE)
[1] 70
```

In this function we used a condition with the controls `if()` and `else` and the function `return()` to display or store the result in a new object. Actually any command can be included in a function living the opportunity to write shortcuts for long series of commands and to build up your own reproducible tools. It is now time to see these controls.

3.3.5 Controlling Flow

3.3.5.1 Conditioning

There are several R instructions to control the flow of the statements of a script. The controls `if()` and `else` are used to apply a condition. The syntax is:

```
if(condition) {evaluation 1} then {evaluation 2}
```

This means that if the condition is true, then the evaluation 1 is applied; otherwise the evaluation 2 is applied. Note the use of curly brackets { } to frame the evaluations. For instance:

```
if(is.vector(num)) {print(num)} else {cat("This is not a vector")}
[1] 10 11 12 13
```

Another way to use a `if/else` condition is to call the function `ifelse()` in which the first argument is the condition or test, the second argument `yes` is the evaluation 1, and the third argument `no` is the second evaluation. For instance, this can be used to replace values according to a threshold. In the following example, all values strictly above 2 are replaced by the word `"white,"` and the values under or equal to 2 are replaced by the word `"black"`:

```
num.color <- ifelse(num>11, yes="white", no="black")
num.color
[1] "black" "black" "white" "white"
```

switch() is another function that compacts the if/else control. This function takes as a first argument a numeric value or a character string that is evaluated and used to choose one of the further arguments. In the following example, the first argument is the numeric value k followed by 4 character strings (sound file formats), each character string being an argument. Setting $k = 3$ returns the third further argument, that is, the character string ".ogg":

```
k <- 3
switch(k, ".wav", ".mp3", ".ogg", ".flac")
[1] ".ogg"
```

The first argument can be a character string. The following lines allow to use between different functions to be applied on an object, here the object num:

```
estimator <- "mean"
switch(estimator,
       median=median(num), mean=mean(num), sd=sd(num))
[1] 11.5
```

3.3.5.2 Looping

A loop is a repetitive task applied to a series of indexed items, either objects or object components. Writing a program loop might sound as dangerous as maneuvering a plane loop for someone who is not used to script writing. But this is not that complex. R provides two types of loop maneuvers, a classical for semantic and an idiosyncratic function named apply() that is based on vectorization. The for loop is quite easy to understand but is in certain cases slower than the apply() solution.

There are three main recommendations to successfully build a for loop: (1) to prepare an empty object with the right class, the right attributes, the right length, and/or the right dimensions, this object will be used to store the results of each loop iteration, (2) carefully set the iteration index, (3) run a test with a single iteration, and (4) check the results.

The syntax of a for loop is as follows:

```
for (variable initialization-condition-variable update){
  code to execute if the condition is true
}
```

In the following example, we first create a vector res containing only missing values encoded as NA. The same number of items or cells than num is obtained by using the function rep() that repeat n times the same value:

```
res <- rep(NA, times=length(num))
res
[1] NA NA NA NA
```

Then we compute the logarithm of each `num` item, and we store the result in the corresponding `res` item. The variable initialization is `i=1`; the condition is that `i` belongs to `1:length(num)`, and the update is made at each iteration :

```
for(i in 1:length(num)) {res[i] <- log(num[i])}
```

We eventually check the result by printing the object `res`:

```
res
[1] 2.302585 2.397895 2.484907 2.564949
```

Another way to loop is to use the facility `repeat` that will execute repetitively instructions stated between curly brackets { }. This repetitive behavior can last indefinitely if no instructions are given to get out of the loop. The exit door is found by combining an `if` condition and a statement `break`, without any type of parentheses. The following instructions create an increment `i`, print this increment, and test afterward whether this increment is higher than 5. This results in a 5 time repeat loop printing value from 1 to 5:

```
i <- 0
repeat{
    print(i)
    i <- i+1
    if(i > 5) break
    }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

The same behavior can be obtained with the control flow function `while()`. In that case the condition is tested before the code execution:

```
i <- 0
while(i<6) {
    print(i)
    i <- i+1
}
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

`apply()` is a loop compacted in a single function.[4] The principle is to apply a function to the rows and/or to the columns of an array or a matrix. `apply` has three arguments: `X` is an array or a matrix object, `MARGIN` indicates whether the operation should apply on the rows (`MARGIN=1`) or on columns (`MARGIN=2`), and `FUN` is the function to apply. This function can be any function, either already implemented, designed previously, or even written directly in the `FUN` argument.

Here is a way to compute the sum of each row of the matrix `mat`, thus applying the function `sum` in a row loop:

```
apply(X=mat, MARGIN=1, FUN=sum)
[1] 22 24
```

The same can be achieved on the columns with:

```
apply(X=mat, MARGIN=2, FUN=sum)
[1] 21 25
```

The argument `FUN` can also accept a new function written within the frame of the function `apply`. Here we design a new function that computes the square of the sum on both rows and columns:

```
apply(X=mat, MARGIN=c(1,2), FUN=function(x) {sum(x)^2})
     [,1] [,2]
```

(continued)

[4]The function `apply` is the member of a family of functions, `sapply`, `lapply`, `vapply`, `mapply`, that slightly differs depending on the class of the input and output objects.

```
[1,]  100  144
[2,]  121  169
```

When consulting the help of `apply()`, we discover mysterious points of ellipsis after the argument `FUN`. These three dots mean that the arguments of the function declared in `FUN` can be embedded in `apply()`. In the following example, we compute the mean of the rows of the matrix `mat`, and we use the argument `na.rm` to remove `NA` if any:

```
apply(X=mat, MARGIN=1, FUN=mean, na.rm=TRUE)
[1] 11 12
```

The function `replicate()` can be used to repeat n times the action of a function. For instance, we repeat 5 times a random sample of 10 values taken in a numeric vector containing integers from 1 to 100. The sampling is operated thanks to the function `sample()`:

```
replicate(n=5, sample(1:10, size=10))
      [,1] [,2] [,3] [,4] [,5]
 [1,]    6    1    6   10    6
 [2,]    7    9    8    6   10
 [3,]    5    7   10    9    3
 [4,]    8    4    7    5    4
 [5,]    2    2    5    4    1
 [6,]   10    8    1    3    7
 [7,]    1    3    2    8    5
 [8,]    3   10    9    2    9
 [9,]    4    5    3    7    2
[10,]    9    6    4    1    8
```

3.3.6 Manipulating Objects

3.3.6.1 Indexing Operators

Part of `R` objects can be extracted through operators (Table 3.2) and specific functions. As defined above, a vector is made of a series of items or cells. Each cell has a specific position along the vector. For instance, in the object `num`, the value `10` has the position 1, and the value `11` has the position 2. We can select these

values individually by using their position or index within num. This is achieved by employing square brackets [] following the simple syntax:

```
objectname[index]
```

We can then extract a single or several cells of num combining this syntax with the function c() and by using the negative sign to obtain the complement selection:

```
num[2]       # item  2
[1] 11
num[2:4]     # items 2 to 4
[1] 11 12 13
num[-(2:4)] # item 1
[1] 10
num[c(1,3)] # items 1 and 3
[1] 10 12
```

This syntax applies to the matrix and data.frame objects by indicating the row and/or the column index following the syntax:

```
objectname[row index, column index]
```

Therefore, we can extract unique cells, columns, or rows as demonstrated here for a matrix:

```
mat[1,1]    # first cell
[1] 10
mat[1,]     # first row
[1] 10 12
mat[,1]     # first column
[1] 10 11
```

and a data.frame:

```
df[,1:2]
   x y
1 10 A
2 11 B
3 12 C
4 13 D
df[-(1:2),]
   x y     z
```

(continued)

```
3 12 C FALSE
4 13 D  TRUE
df[,c(1,3)]
   x     z
1 10  TRUE
2 11 FALSE
3 12 FALSE
4 13  TRUE
```

The `array` objects are defined with *n* dimensions that are each indexed leading to the syntax

```
objectname[index, index, ..., index]
```

The object `ary` has three dimensions; we therefore need to set three indices separated with two commas. To get the single first value, we need to provide three dimensions:

```
ary[1, 1, 1]
[1] 10
```

But if we wanted to get the first matrix, we leave the two first indices blank:

```
ary[, , 1]
     [,1] [,2]
[1,]   10   12
[2,]   11   13
```

The `list` objects are indexed by using double square brackets `[[]]` following:

```
objectname[[index]]
```

For instance, to get the second component of the list `lst`, we run:

```
lst[[3]]
[1] "Bye bye moon"
```

However, this third component was named z that we can use with a $ to extract it:

```
lst$z
[1] "Bye bye moon"
```

As a data.frame is a special case of a list with columns being components, we can apply the same syntax and extract data.frame columns by their name:

```
df$z
[1]  TRUE FALSE FALSE  TRUE
```

Eventually, there is special case with the particular s4 objects. This objects are a bit particular, their components are termed slots, and they are identified with an @ replacing the $. Here is the syntax:

```
objectname@component
```

To conclude, try to remember that (1) square brackets [], either simple or double, have to be used with objects, (2) normal brackets () with functions, (3) curly brackets { } with control flow and function writing, and (4) named components can be called with either a $ or a @ symbol.

3.3.6.2 Finding an Item Position

To find visually the position of an item or several indices in a long vector or large matrix is not possible. The solution is to ask the position to R with the function which() that follows the syntax:

```
which(objectname logical-operator value)
```

For instance, the following command returns that the value 11 is in second position in the vector num:

```
which(num==11)
[1] 2
```

If we had to ask to R where the females in the factor fact are, we should write:

```
which(fact=="Female")
[1] 2 3
```

We can apply which() to a matrix. Setting the argument arr.ind to TRUE returns the row and column numbers of the positions searched for, something like in a Battleship game. In the following, we look for the value 11 in mat:

```
which(mat==11)
[1] 2
which(mat==11, arr.ind=TRUE)
     row col
[1,]   2   1
```

Similarly for an array, we obtain:

```
which(ary==11)
[1] 2
which(ary==11, arr.ind=TRUE)
     dim1 dim2 dim3
[1,]    2    1    1
```

But this action is not possible in a heterogeneous list as the object lst; we need to run which() on each component of the list:

```
which(lst$x==11)
[1] 2
which(lst$y==11, arr.ind=TRUE)
     row col
[1,]   2   1
```

Eventually, it is often necessary, in particular in sound analysis, to localize the minimum and/or the maximum of an object. This can be obtained with the function which.min() and which.max(), respectively:

```
which.min(num)
[1] 1
which.max(num)
[1] 4
```

The results returned by `which()` can be used to obtain the values of the items searched for. If we want to know which values of `num` are superior or equal to `11`, we should run:

```
pos <- which(num >= 11)
num[pos]
[1] 11 12 13
```

or more simply:

```
num[num>=11]
[1] 11 12 13
```

3.3.6.3 Concatenating Objects

`R` includes a series of fundamental functions to concatenate or bind values and/or objects. We already met one of the most used function `c()` to combine values in a vector or to combine vectors. We can concatenate values:

```
c(1.618, 3.14)
[1] 1.618 3.140
c("hello", "moon")
[1] "hello" "moon"
```

values and vectors:

```
num2 <- c(num, 1.62, 3.14)
num2
[1] 10.00 11.00 12.00 13.00  1.62  3.14
```

or vectors:

```
c(num, num2)
 [1] 10.00 11.00 12.00 13.00 10.00 11.00 12.00 13.00  1.62
[10]  3.14
```

Rows and columns of `matrix` and `data.frame` can be combined with the functions `rbind()` and `cbind()` to bind rows and columns, respectively:

```
rbind(mat, 15:16)
     [,1] [,2]
[1,]   10   12
[2,]   11   13
[3,]   15   16
rbind(df, c(5, "D", FALSE))
   x y     z
1 10 A  TRUE
2 11 B FALSE
3 12 C FALSE
4 13 D  TRUE
5  5 D FALSE
```

```
cbind(mat, 15:16)
     [,1] [,2] [,3]
[1,]   10   12   15
[2,]   11   13   16
cbind(w=c(0,0,1,0), df)
  w  x y     z
1 0 10 A  TRUE
2 0 11 B FALSE
3 1 12 C FALSE
4 0 13 D  TRUE
```

The insertion of rows and columns is intuitively achieved by using indexing. So to insert a row:

```
rbind(mat[1,], 15:16, mat[2,])
     [,1] [,2]
[1,]   10   12
[2,]   15   16
[3,]   11   13
```

(continued)

```
rbind(df[1,], c(5, "D", FALSE), df[2:4,])
    x y     z
1  10 A  TRUE
2   5 D FALSE
21 11 B FALSE
3  12 C FALSE
4  13 D  TRUE
```

and to insert a column:

```
cbind(mat[,1], 15:16, mat[,2])
     [,1] [,2] [,3]
[1,]   10   15   12
[2,]   11   16   13
cbind(x=df[,1], w=c(0,0,1,0), df[,2:3])
   x w y     z
1 10 0 A  TRUE
2 11 0 B FALSE
3 12 1 C FALSE
4 13 0 D  TRUE
```

3.3.6.4 Reading and Saving Objects

Objects can be imported or exported through specific functions that are summarized in Table 3.4. Each function has specific arguments to tune the import or the export like the cell separator format, the decimal format, and the occurrence of row names. These arguments will not be detailed here, but it is essential to understand their use to avoid data corruption.

Table 3.4 Import and export of R data

File format	File extension	In	Out
ASCII file	No specific extension	`scan()`	`write()`
Text file	`.txt`, `.dat`	`read.table()`	`write.table()`
Tab-formatted file	`.csv` file	`read.csv()`	`write.csv()`
Binary file	No specific extension	`readBin()`	`writeBin()`
R compressed file format	`.Rdata`	`load()`	`save()`

Main functions to import and export data of different formats into and out of R

3.3.7 Vectorization and Recycling

Most of R operations are vectorized. This means that an operation does not only perform on the first item of an object but on all of them. Vectorization is a kind of implicit loop making programming much easier than with other languages.

The addition of a single value to a vector is achieved with:

```
num + 3
[1] 13 14 15 16
```

The multiplication of two vectors of same length is obtained as:

```
num * num
[1] 100 121 144 169
```

The natural logarithm of a matrix is simply returned with:

```
log(mat)
         [,1]     [,2]
[1,] 2.302585 2.484907
[2,] 2.397895 2.564949
```

Another nice side of vectorization is the replacement of items through indexing without a loop:

```
num[num > 11] <- "white"
num[num <= 11] <- "black"
num
[1] "black" "black" "white" "white"
```

We previously solved this problem (see Sect. 3.3.5.1) with an even more compact solution generating a new object:

```
num.color <- ifelse(num>11, yes="white", no="black")
num.color
[1] "white" "white" "white" "white"
```

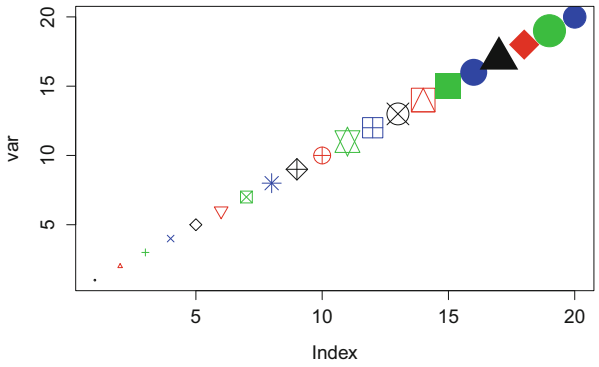


Fig. 3.1 Vectorization and recycling. This graphic uses data recycling (argument `color`) and vectorization (argument `cex`)

In addition, R can recycle the items of an object when vectorization operations are requested on objects of dissimilar length. This can be very useful in particular for graphics as in the following example the recycles color and point size (Fig. 3.1):

```
var <- 1:20
plot(var, col=1:4, pch=var, cex=var/4)
```

However recycling can be quite dangerous. For instance, it is possible to add two vectors of different length or two matrices of different dimensions as the missing items are replaced by the first items adjusting artificially the size of the objects. Here is an example of a dangerous recycling:

The addition of the two vectors of different length:

```
1:3 + 1:5
[1] 2 4 6 5 7
```

is equivalent to:

```
c(1,2,3,1,2) + 1:5
[1] 2 4 6 5 7
```

Vectorization and recycling are such common and intuitive that it is necessary to carefully check the length and dimensions of the objects in case of doubt. This

should allow to avoid important mistakes in particular in scripts with loops and conditions.

3.3.8 Handling Character Strings

Even if sound analysis and synthesis have nothing to deal with text and linguistics, it is often necessary to manipulate character chains for manipulating factors or plotting text on graphics. There are a few basic functions to write text properly with R. The concatenating function c() can generate vectors of several character strings:

```
c("My", "first", "spectrogram", "with R")
[1] "My"          "first"       "spectrogram" "with R"
```

However, this does not make a sentence, only a vector made of four items or cells. To obtain a sentence, we need to paste the four items with the function paste():

```
paste("My", "first", "spectrogram", "with R")
[1] "My first spectrogram with R"
```

By default the function paste() introduces a white space between the terms, but this can be changed with the argument sep:

```
paste("My", "first", "sound", "analysis", "with", "R", sep="-")
[1] "My-first-sound-analysis-with-R"
paste("My", "first", "sound", "analysis", "with", "R", sep="/")
[1] "My/first/sound/analysis/with/R"
paste("My", "first", "sound", "analysis", "with", "R", sep="")
[1] "MyfirstsoundanalysiswithR"
```

It may be necessary to extract or to split a character string. The function substr() extracts the characters between the position start and end when the function strsplit() splits the character string following a pattern specified in the argument split:

```
substr("sound.wav", start=1, stop=5)
[1] "sound"
```

(continued)

```
strsplit("sound_analysis_with_R", split="_")
[[1]]
[1] "sound"    "analysis" "with"     "R"
```

3.3.9 *Drawing a Graphic*

R is a splendid tool to produce high-quality, simple, or intricate graphics that exactly meets your need. There is almost no limitation to visualize your data converting again a software user into a designer. The graphical power of R is mainly due to the organization of graphical functions into four main groups of functions:

- **high-level** functions that produce a complete graphic,
- **parametrization** functions that control the general appearance of the graphic,
- **low-level** functions that complete or decorate a high-level function plot by adding text, arrows, rectangle, etc.,
- **plate organization** functions that organize several graphics into a single plate.

3.3.9.1 High-Level Plot Functions

R has several functions that draw a complete plot. The main high-level plot function is `plot()`. This function draws a classical X–Y plot as shown in Fig. 3.2 that maps

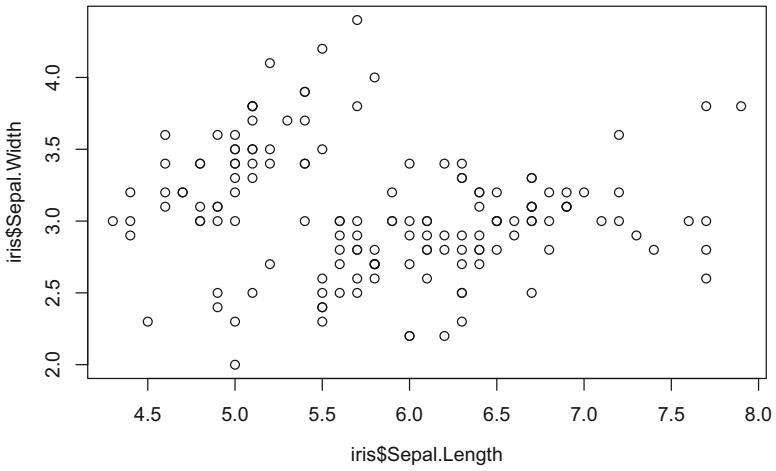


Fig. 3.2 Scatter plot. A simple X–Y scatter plot with the `Sepal.Length` and `Sepal.Width` variables of the dataset `iris`

the length and the width of the sepal of four species of iris included in the dataset `iris`. This is a classic dataset used that includes morphometric variables of three iris species, namely, *Iris setosa*, *Iris versicolor*, and *Iris virginica* (Fisher 1936):

```
data(iris)
plot(iris$Sepal.Length, iris$Sepal.Width)
```

The behavior of the `plot()` may change according to the `attributes` of the objects to be visualized. For instance, the function `hclust()` runs a hierarchical cluster analysis on a dissimilarity matrix that can be plot as a classification tree with `plot()`. The following lines, slightly modified from the help page of `hclust()`, run a cluster analysis and draw the decision tree on the dataset `USArrests`:

```
hc <- hclust(dist(USArrests))
plot(hc)
```

The function `plot()` can even accept a function as an input. The next command draws the sinus function between $-\pi$ and $+\pi$:

```
plot(sin, -pi, pi)
```

There are also specific high-level functions, somehow classical plot functions, to plot a histogram (`hist()`), a barplot (`barplot()`), a boxplot (`boxplot()`), a pie chart (`pie()`), and a dot chart (`dotchart()`). This short list of plot functions is not exhaustive as hundreds of graphical functions are implemented in packages.

Most of the high-level plot functions have the following fundamental arguments that can be used to change the main graphical parameters:

`main`	main title (above the graphic)
`sub`	subtitle (under the graphic)
`xlab`	label of the *x* axis
`ylab`	label of the *y* axis
`ylim`	limits of the *y* axis
`xlim`	limits of the *x* axis

3.3.9.2 Parametrization

The general appearance of an R plot can be completely modified with a long list of parameters that are available in the function `par()`:

`bg`	background color
`cex`, `cex.axis`, etc	character and symbol sizes
`col`, `col.axis`, etc	colour
`fg`	foreground color
`font`, `font.axis`, etc	font of the text
`las`	style (orientation) of axis labels
`lty`	line type
`lwd`	line width
`pch`	point symbol
`xaxt`, `yaxt`	axis type
`xlog`, `ylog`	logarithm scale
...	

These options can be either set with the function `par()` or with high-level plot functions that redirect to `par()`. However, the effects might slightly differ. For instance, setting character size to 2 with `par()` increases the size of all characters and symbols when setting it within `plot()` affects the size of the symbol only. The following instructions are not equivalent:

```
par(cex=2)
plot(Sepal.Length, Sepal.Width)
```

```
plot(Sepal.Length, Sepal.Width, cex=2)
```

So remember to check the help of `par()` function when you need to change a graphical parameter rather than searching in the plot function help.

3.3.9.3 Low-Level Plot Functions

It is often required to add items to an existing plot. This is achieved with low-level functions that can help in marking a graph. The functions can be classified in three main categories:

- functions to add text,
  - `text()` to add text within the plot region, i.e., within the region delimited by the axes,
  - `mtext()` to add text in the margins of the graphic,
  - `plotmath()` to write symbols and mathematical expressions,
  - `title()` to add main and subtitles,
- functions to add a geometric shape,
  - `points()` to add points,
  - `abline()` to add a $y = ax + b$ line,
  - `lines()` to add lines,
  - `segments()` to add segments,
  - `arrows()` to add arrows,
  - `rect()` to draw a rectangle,
  - `polygon()` to draw a polygon,
  - `box()` to add a frame around the plot,
- functions to add graphic items,
  - `axis()` to add an axis including ticks and labels on any side of the plot,
  - `legend()` to add a legend specifying the symbols, lines, or colors used to discriminate groups of data,
  - `grid()` to add a line grid within the frame.

All these parameters can be used successively in a script as in the following code that was used to produce Fig. 3.3:

```
data(iris)
par(las=1)
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch=20, cex=1.5, col=as.numeric(iris$Species),
     main="Iris sepal size",
     xlab="Sepal length", ylab="Sepal width")
grid(col="lightgrey")
axis(side=4)
legend("topright",
       legend=c("setosa", "versicolor", "virginica"),
       col=1:3, pch=19, bg="white", text.font=3)
points(mean(iris$Sepal.Length), mean(iris$Sepal.Width),
       pch="+", col=4, font=2, cex=2)
arrows(x0=6.25, y0=4, y1=4, x1=5.9, length=0.1)
text(x=6.75, y=4, labels="Important point")
box(lwd=3)
```

However, it is not possible to undo a change. A modification of a command implied to run another time all the script to get an updated graphic.

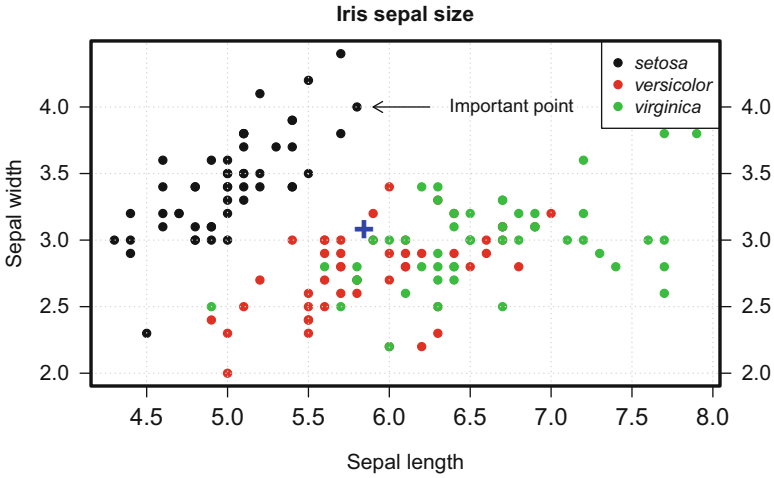


Fig. 3.3 Graphic tuning. A meaningless example of graphic changes using low-level plot functions

3.3.9.4 Organizing a Plate of Graphics

Plotting a single graphic is, in most case, not enough in particular when publishing in scientific journals that struggle against white space. Most of scientific illustrations are based on several graphics combined in a single plate. Combining graphics in drawing or illustrating software can be turned to a nightmare due to sources that may have different resolutions, sizes, color encodings, fonts, and file formats. Updating such a plate can also be very time consuming as sources can be not linked to the plate file and can be lost. `layout()` is a fancy function that can do all the job without quitting `R`. Thanks to this function you can prepare the layout of your plate and include graphics generated within `R` or out of `R` by importing drawings or pictures in usual graphics file formats like `.png` or `.jpg`. The principle of `layout()` is simple and brilliant. The first thing is to draw on a piece of paper the layout of your plate with each graphic numbered. This can look like the drawing of a 5-year-old child shown in Fig. 3.4.

You then only need to convert your work of art into a numeric matrix with the following simple rules: divide your drawing in a regular number of cells which dimensions should fit with the smallest graphic, number these cells with the number of each graphic so that several cells covered by a graphic have the same number, and give the number 0 to white cells. For the sketch shown in Fig. 3.4, this should give the following matrix that is filled by rows by setting the logical argument `byrow` to `TRUE`:

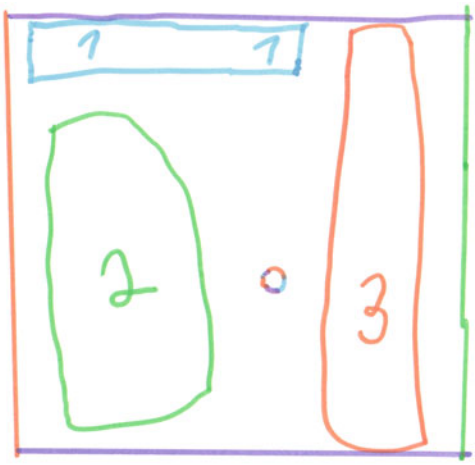


Fig. 3.4 Layout plate scheme by a 5-year-old hand. The first step of composing an R graphic plate is to take a pen and piece of paper and to draw it! Colors are not necessary...

```
drawing <- matrix(c(1,1,3,2,0,3), nrow=2, byrow=TRUE)
drawing
     [,1] [,2] [,3]
[1,]    1    1    3
[2,]    2    0    3
```

We now can feed the function `layout()` with this matrix. However, our sketch shows that the graphic 1 is about three times larger than the graphic 3 and that the height of the graphic 1 is roughly half the height of the graphic 2. This information has to be provided to `layout()` with the arguments `width` and `height`. Both argument work with relative values such that setting `height=c(1,2)` is equivalent to `height=c(0.5,1)`. The last step is to see the layout we prepared; this is achieved with the function `layout.show()` (Fig. 3.5).

```
layout(drawing, width=c(3,1), height=c(1,2))
par(cex=3)
layout.show(3)
```

All we have to do now is to fill in the sections of the plate with graphics. The graphics are added in the numbered order so that the flow of the script should follow plate numbering.

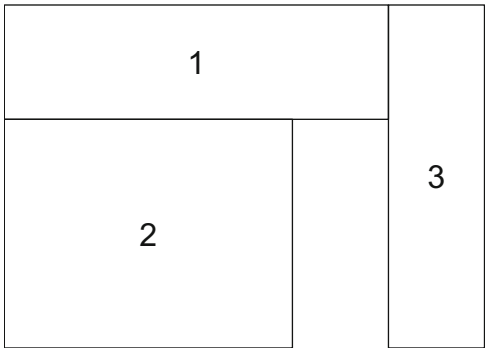


Fig. 3.5 Layout plate scheme with `layout()`. We first prepare the layout by generating an appropriate matrix. The size of the graphic numbers is increased with the function `par()`

3.3.9.5 `ggplot2` Alternative

Another family of R plots was introduced with the package `ggplot2` written by Wickham (2009). These graphics are of high quality and offer new possibilities to display data in a very efficient and elegant way. The success of `ggplot2` among the R community is tremendous such that there is no way to escape it. However `ggplot2` graphic production is based on a particular code grammar that cannot be detailed here and the packages dealing with sound (see Sect. 3.4.2) do not use `ggplot2` except the function `ggspectro()` of `seewave`. `ggplot2` will therefore not be considered here.

3.3.9.6 Saving Graphics

R can save graphics in a long list of graphic formats. For sound analysis and synthesis, it seems that only two file formats are necessary. The `.pdf` format produces high-quality vector files that is recommended in almost all cases, including writing reports, thesis, or scientific articles. This format is of high interest because graphics can be resized without resolution loss.

However, if a graphic contains a high number of points—as it is the case of spectrograms (see Chap. 11)—the size of the `.pdf` file may increase significantly reaching several Mo that may be difficult to open or to handle with word processors. The solution is to switch to a raster or bitmap file composed with pixels. Among the several raster formats available, the format `.png` ensures a good resolution with lossless compression and has the main advantage to be an open format.

Graphical user interfaces, as RStudio, may provide interactive options to save graphics, but these are only graphical wrappers of R functions, and the best solution is certainly to use directly these functions in scripts.

There are different ways to save plots in `.pdf` or `.png`, but the most convenient and efficient solution is (1) to open an external graphic file, (2) to write the graphical R instructions, and (3) to close the file. These three stages lead to the following pseudo-code:

```
open file
code to generate a graphic
close file
```

The R open file instruction is `pdf()` or `png()` and the instruction to close the file, or device, is `dev.off()`. If we wish to save a simple plot in a `.pdf` file named `my_first_plot.pdf`, we have to run the following commands:

```
pdf("my_first_plot.pdf")
plot(1)
dev.off()
```

or for `.png`:

```
png("my_first_plot.png")
plot(1)
dev.off()
```

The plot is saved in the R working directory. The complete path to a specific directory where graphics are saved can be specified in the file name argument. The path depends on the operating system in use. The code to produce the graphics can be as long as needed. This code can include analyses, not only plotting instructions.

The functions `pdf()` and `png()` have several arguments to control the size and the properties of the file. Usually there are few modifications to apply to the default values of the `pdf()` arguments, but it may be required to increase the size of the graphics device. This is mainly achieved with four arguments that have to be changed in conjunction: `width` and `height` of the device that are given in either pixels, inches, cm, or mm as specified in the argument `units` and `pointsize` that sets up the default point size of the plotted text. Increasing the width and height should be accompanied with an increase of point size, or puzzling graphics may pop up. For instance, the following instructions produces a large graphic with tiny labels:

```
png("plot.png", width=1200, height=1200)
plot(1)
dev.off()
```

It is necessary to increase the point size from the default value 12 to a new value 24 to obtain regular text size:

```
png("plot.png", width=1200, height=1200, pointsize=24)
plot(1)
dev.off()
```

The R graphical window display may slightly differ from the content of the graphics file. It is often safe to open the saved file to check the graphical output.

3.3.10 Scripting

An R user ends up very quickly with a long series of commands that constitute a script. Such script should be saved in `.r` (alternatively `.R`) file that can be opened by any code or text editor. Here follow some counsels to write repeatable and efficient scripts[5]:

- use an adapted R editor or GUI which offers at least syntax coloration, parenthesis matching, and auto-completion,
- include all the steps necessary to repeat the script without requiring the use of a mouse. This includes, for instance, code to load necessary packages, to change the working directory, to import data, to export graphics, and to save data,
- choose simple and short names for external files and internal objects. Avoid special characters and white space that might be complex to handle,
- comment your scripts by placing a hash symbol (#) before your comments so that you can understand what you have done a few days or years later,
- try to avoid to place several commands on the same script line,
- indent your code on the right with tabulations so that start and end of loops or long new functions can be easily determined,
- comment the script lines that seem to be useless but do not delete them; they might be useful later,
- replace when possible `for` loops by `apply()` and `apply`-like functions to speed up the computing process,
- avoid importing heavy files which may saturate the RAM,
- clean the RAM by removing unnecessary objects with `rm()`,
- call `print(i)` in loops to display the current iteration,
- save regularly your script on your local computer and on external backup devices,
- go out and run some kilometers in the forest if too many bugs invade your script.

[5] See Wickham (2014) for a complete description of efficient R programming.

Once properly written a script can be called with a single line of code by taking advantage of the function `source()`. This function just runs the code saved in an `.r` file. Its use is trivial:

```
source("myscript.r")
```

Another solution is to run a script without opening the graphical interface of R. This can be useful to gain memory access or to include an R application in a shell program. This is achieved by running in the Unix terminal (MaxOS, Linux):

```
R CMD BATCH myscript.r
```

Windows make things more complex as it is necessary to find the path to the executable. Here is an example with a 64 bit Windows system and R version 3.3.4; this command line should be adjusted to your configuration:

```
"C:\Program Files\R\R-3.3.4\bin\x64\R.exe" CMD BATCH
                                "myscript.r"
```

In each case, it might be necessary to specify the path to the `.r` file, like `"home/username/Desktop/myscript.r"` for Unix system or `"C:\analysis\myscript.r"` for a Windows system.

3.3.11 Calling External Software

Some functionalities might not be available in R, and it may be hence required to call external software (see Sect. 3.4.1). For instance, R might not be interactive enough for some users, as users wish to quickly explore through time navigation long sound files. In that case, external software can be launched from R with the command `system()`. In Unix system the function is easy to use; it only requires to include between quotes the command that would be written in the shell terminal. Audacity is a free and nice interactive and graphical audio editor that can be very useful. The following Unix command open with Audacity the file `theremin.wav` stored in a directory named `sample`:

```
system("audacity sample/theremin.wav")
```

On a PC running with 64 bit Windows system, the same command is considerably more complex to write. We need to find the full paths to the `.exe` program and to the `.wav` file. We also need to make use of the function `shQuote` so that the command is correctly quoted for Windows command line:

```
exename <- "C:/Program Files (x86)/Audacity/audacity.exe"
filename <- paste(getwd(), "/sample/theremin.wav", sep="")
system(paste(shQuote(exename), shQuote(filename, type="cmd")))
```

The path to the `.exe` file would probably need to be adapted to your Windows version and configuration. The `system()` function is probably even more useful when invoking command line-driven software as SoX (see Sect. 3.4.1). SoX includes a very long list of capabilities that might be worth to invoke from `R`. As a simple example, the following Unix instruction generates a new file named `theremin-slow.wav` slowed down in time by a factor of 2 without modifying the pitch of the sound[6]:

```
system("sox sample/theremin.wav sample/theremin-slow.wav
                                tempo 0.5")
```

3.4 R and Sound

3.4.1 To Use or Not to Use R for Sound Analysis?

Before going deep into the details of how `R` manages sound, it is legitimate to wonder whether `R` can behave as a good audio software when there is an army of other softwares. We can try to list in an objective way the pros and cons to use `R` mainly based on the properties of `R` as introduced above (see Sect. 3.1).

We vote for `R` to analyze and synthesize sound because:

- `R` is free and open,
- `R` ensures reproductibility of work,
- `R` is a collaborative project,
- `R` can be tuned to specific needs,
- `R` can run batch work,
- `R` is an integrative tool that can run upstream (data management, protocol sampling, etc.) and downstream (statistics, data visualization, etc.) analyses,
- `R` produces high-quality graphics.

We may denigrate `R` to analyze and synthesize sound because:

- `R` memory management is not optimized,
- `R` graphic device is slow,

[6]Note that the `seewave` package has a function `sox()` that can help in parsing SoX from `R`.

- R is not interactive.

To conclude, R seems to be a pertinent solution for sound analysis and synthesis but not for aural and visual inspection of sounds. This preliminary, but imperative, step of acoustic studies can be achieved with recording and editing software. There are actually free alternatives to R listed here by alphabetic order:

Audacity is a free, easy-to-use, multitrack audio editor and recorder for Windows, Mac OS X, Linux, and other operating systems. The interface is translated into many languages,
http://audacity.sourceforge.net

Praat is a command-driven software dedicated to the analysis of speech in phonetics. It includes several analysis functions that can be run on a wide range of operating systems, including various versions of Unix, Linux, Mac, and Microsoft Windows,
http://www.fon.hum.uva.nl/praat/

Raven is a software program for the acquisition, visualization, measurement, and analysis of sounds,
http://www.birds.cornell.edu/brp/raven/ravenoverview.html

Sonic Visualizer provides waveform and spectrogram audio visualizations for use with files of music audio data,
http://www.sonicvisualiser.org/

Syrinx is a Windows sound recording/editing/playback program including visual analysis, printing, and illustration of spectrographs, time/frequency measurements, and real-time automated sound event recording,
http://www.syrinxpc.com/

WaveSurfer is an open-source tool for sound visualization and manipulation. Typical applications are speech/sound analysis and sound annotation/transcription. WaveSurfer may be extended by plug-ins as well as embedded in other applications,
http://www.speech.kth.se/wavesurfer/

3.4.2 Main Packages

Here is the official description of the main R packages we may need to run sound analysis and synthesis. The main author/maintainer with his/her affiliation and the first date of publication are given between square brackets. Packages considered in this book are marked with a {*}; packages removed from CRAN are indicated with a {†}. The list, which can be converted as a network (Fig. 3.6), follows a historical order:

`sound`† basic functions for dealing with `.wav` files and sound samples [Matthias Heymann, Germany, 31 August 2002]

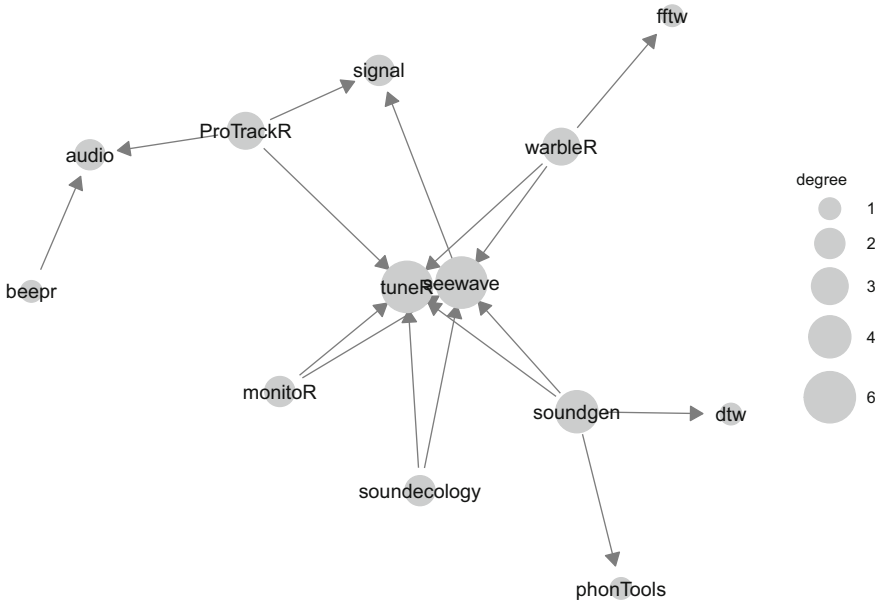


Fig. 3.6 Directed network of CRAN packages dedicated to sound. The network was constructed based on the main directed relationships between CRAN packages dedicated to sound. The size, or degree, of each node corresponds to the number of connections. This highlights the central position of `tuneR` and `seewave`. Built with the package `network` (Butts 2008) and drawn with the package `GGally` (Schloerke et al. 2017)

`tuneR*` collection of tools to analyze music, extract features like MFCCs, handle wave files, read mp3, transcription, ... Also contains functions ported from the rastamat Matlab package [Uwe Ligges, Technical University of Dordmund, Germany, 11 September 2004]

`seewave*` functions for analyzing, manipulating, displaying, editing, and synthesizing time waves (particularly sound). This package processes time analysis (oscillograms and envelopes), spectral content, resonance quality factor, entropy, cross correlation and autocorrelation, zero-crossing, dominant frequency, analytic signal, frequency coherence, 2D and 3D spectrograms, and many other analyses [Jérôme Sueur, Muséum national d'Histoire naturelle, France, 10 March 2006]

`signal*` a set of signal processing `R` functions originally written for Matlab/Octave. Includes filter generation utilities, filtering functions, resampling routines, and visualization of filter models. It also includes interpolation functions [Uwe Ligges, Technical University of Dordmund, Germany, 10 December 2006]

audio* interfaces to audio devices (mainly sample-based) from R to allow recording and playback of audio. Built-in devices include Windows MM, Mac OS X AudioUnits, and PortAudio (the last one is very experimental) [Simon Urbanek, AT&T Research Labs, USA, 29 September 2008].

playitbyr† a flexible toolkit for data sonification, with syntax modeled after the ggplot2 package. The functions allow the user to map data onto sonic parameters like pitch, tempo, and rhythm and output sound and sound files [Ethan Brown, 19 December 2011]

csound† provides basic access in an R session to Csound http://www.csounds.com, a powerful free and open-source software sound synthesizer. The package functionality is largely geared toward supporting the needs of the playitbyr package for sonification and is not particularly mature on its own. But it certainly can be used without knowing anything about playitbyr [Ethan Brown, 19 December 2011]

phonTools* contains tools for the organization, display, and analysis of the sorts of data frequently encountered in phonetics research and experimentation, including the easy creation of IPA vowel plots and the creation and manipulation of WAVE audio files [Santiago Barreda, University of California Davis, USA, 24 July 2012]

audiolyzR creates audio representations of common plots in R [Eric Stone, AT&T Research Labs, 17 February 2013]

soundecology* functions to calculate indices for soundscape ecology and other ecology research that uses audio recordings [Luis J. Villanueva-Rivera, USA, University of Purdue, USA, 10 November 2013]

monitoR* acoustic template detection and monitoring database interface [Sasha D. Hafner and Jon Katz, The University of Vermont, USA, 31 March 2014]

beepr Easily plays notification sounds on any platform [Rasmus Bååth, Lund University, Sweeden, 26 June 2014]

warbleR* a tool to streamline the analysis of animal acoustic signal structure. The package offers functions for downloading avian vocalizations from the open-access online repository xeno-canto, displaying the geographic extent of the recordings, manipulating sound files, detecting acoustic signals or importing detected signals from other software, assessing performance of methods that measure acoustic similarity, conducting cross-correlations, dynamic time warping, measuring acoustic parameters, and analyzing interactive vocal signals, among others. Most functions working iteratively allow parallelization to improve computational efficiency [Marcelo Araya-Salas and Grace Smith Vidaurre, Cornell University, USA, 24 July 2015]

ProTrackR "ProTracker" is a popular music tracker to sequence music on a Commodore Amiga machine. This package offers the opportunity to import, export, manipulate, and play "ProTracker" module files. Even though the file format could be considered archaic, it still remains popular to this date. This package intends to contribute to this popularity and therewith keeping the legacy of "ProTracker" and the Commodore Amiga alive [Pepijn de Vries, 26 September 2015]

soundgen tools for sound synthesis and acoustic analysis. Performs parametric synthesis of sounds with harmonic and noise components such as animal vocalizations or human voice. Also includes tools for spectral analysis, pitch tracking, audio segmentation, self-similarity matrices, and morphing [Andrey Anikin, 4 September 2017].

3.4.3 How to Install seewave

This book is mainly dedicated to the package seewave. seewave is linked to other R packages and requires some external tools (Fig. 3.7). seewave is principally linked to tuneR which is itself linked to signal. To increase the speed of Fourier transform computation (see Chap. 10), some seewave functions also refer to the package fftw written by Uwe Ligges. This package is a wrapper around the Fast Fourier Transform in the West (FFTW v.3.3), a C subroutine library for computing the discrete Fourier transform. In a similar effort to reduce computation time, a seewave function uses LIBSNDFILE, a widely used C library written by Erik de Castro Lopo for reading and writing audio files. These necessary dependencies imply that these packages and C libraries should be install

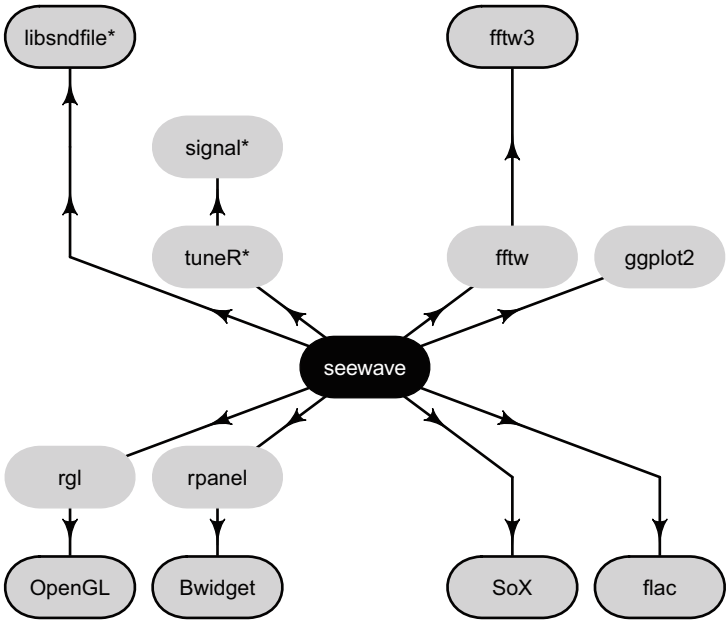


Fig. 3.7 Flowchart of seewave dependencies. R packages are in rounded boxes. External tools are in framed rounded boxes. Mandatory items are labeled with a star (*). Drawn with the package diagram (Soetaert 2014)

upstream before to try to install seewave. Both FFTW and LIBSNDFILE are usually installed by default on Windows systems. On Mac OS, the Mac package manager brew can be used to install the desired libraries.

```
brew install libsndfile
brew install fftw
```

On Ubuntu-like systems the external libraries can be installed with the following shell commands:

```
sudo apt-get install libfftw3-3 libfftw3-dev
sudo apt-get install libsndfile1 libsndfile1-dev
```

When everything is properly installed, the following instruction should finish the process:

```
install.packages(c("signal", "tuneR", "seewave"))
library(tuneR)
library(seewave)
```

However, there are three additional suggested packages. These packages are not necessary to install seewave, but they are required to run some specific functions. Among these three packages, the package rpanel, which produces graphical user interfaces, works with Tcl/Tk language and the external library BWidget. This library is installed by default on almost all operating systems, but it might be necessary to run the following command under Ubuntu-like systems:

```
sudo apt-get install bwidget
```

The package rgl is used to generate 3D dynamics graphics. rgl package is based on the external library OpenGL. The installation of rgl is straightforward on Windows, but there might be some issues with Linux so the package should be downloaded and installed directly from the terminal with the following shell command:

```
sudo apt-get install r-cran-rgl
```

We should then be ready to start with R and sound!

Chapter 4
Playing with Sound

We have seen in the previous chapter general facts about `R` objects and functions. Here we will see the peculiarities of `R` objects that contain sound and how to handle them.

4.1 Object Classes

Classes of objects that can contain sound can be grouped into three main groups:

1. Usual numeric classes: `vector`, `matrix`, and `data.frame`
2. Time series classes: `ts` and `mts`
3. Sound-specific classes: `audioSample`, `sound`, and `Wave`

4.1.1 `vector`, `matrix`, `data.frame` Classes

Any numeric vector can be considered as a sound as soon as the sampling frequency f_s is known. A 440 Hz tuning fork sine sound sampled at $f_s = 8000$ Hz during 1 s can be generated as a `vector` with the following code:

```
v.sound <- sin(2*pi*440*seq(0,1,length.out=8000))
```


J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_4

We can check the properties of the object:

```
is.vector(v.sound)       # class
[1] TRUE
mode(v.sound)            # mode
[1] "numeric"
length(v.sound)          # number of samples
[1] 8000
```

Similarly, any one-column `matrix`, or any column of a `matrix` or any column of a `data.frame`, can be considered as sound as soon as the sampling frequency f_s is known and could be used later in specific sound functions. For a one-column matrix, we would have:

```
m.sound <- as.matrix(v.sound)
is.matrix(m.sound)        # class
[1] TRUE
mode(m.sound)             # mode
[1] "numeric"
dim(m.sound)              # dimensions
[1] 8000    1
```

For one-column data frame, this would be:

```
df.sound <- data.frame(v.sound)
is.data.frame(df.sound)    # class
[1] TRUE
mode(df.sound)             # mode
[1] "list"
dim(df.sound)              # dimensions
[1] 8000    1
```

4.1.2 `ts` and `mts` Classes

Time series analysis is an important topic in statistics mainly because time series are the focus of econometry and finance. As detailed by the CRAN Task View dedicated to time series analysis,[1] there are more than 220 packages related to time

[1] https://cran.r-project.org/web/views/TimeSeries.html

series. These packages cover, among others, time and date formatting, forecasting, frequency analysis, decomposition and filtering, seasonality, models, and nonlinear time series. Most of these utilities are not relevant for sound analysis and synthesis, but, as mentioned in Sect. 2.3, sound being a time series, it is relevant to know how time series are treated by R.

The fundamental time series class is `ts` that can represent regularly spaced time series using numeric time stamps. The class `ts` is particularly well-suited for annual, monthly, and other calendar periods. If we come back to the monthly measurements of ambient carbon dioxide CO_2 concentration (see Sect. 2.3), the dataset `co2` is a time series including 468 observations taken every month from 1959 to 1998. This information can be retrieved with a set of functions as shown here:

```
data(co2)
is.ts(co2)        # class
[1] TRUE
length(co2)       # number of observations
[1] 468
start(co2)        # date of start
[1] 1959    1
end(co2)          # date of end
[1] 1997   12
deltat(co2)       # time resolution, here 1/12 of a year
[1] 0.08333333
frequency(co2)    # sampling frequency, here 12 times / year
[1] 12
```

These data can be visualized with the function `plot.ts()` or directly with `plot()` as shown in Fig. 2.16.

Therefore to generate the 440 Hz sound sampled at 8000 Hz, we can use the function `ts` and specify that time starts at 0 and ends at 1 and that the frequency of observation is 8000. In that case, the sampling frequency f_s is embedded in the `ts` object:

```
ts.sound <- ts(v.sound, start=0, end=1, frequency=8000)
is.ts(ts.sound)       # class
[1] TRUE
length(ts.sound)      # number of samples
[1] 8001
start(ts.sound)       # start of recording
[1] 0 1
end(ts.sound)         # end of recording
[1] 1 1
deltat(ts.sound)      # time resolution, here 125 ms
[1] 0.000125
frequency(ts.sound)   # sampling frequency, here 8000 Hz
[1] 8000
```

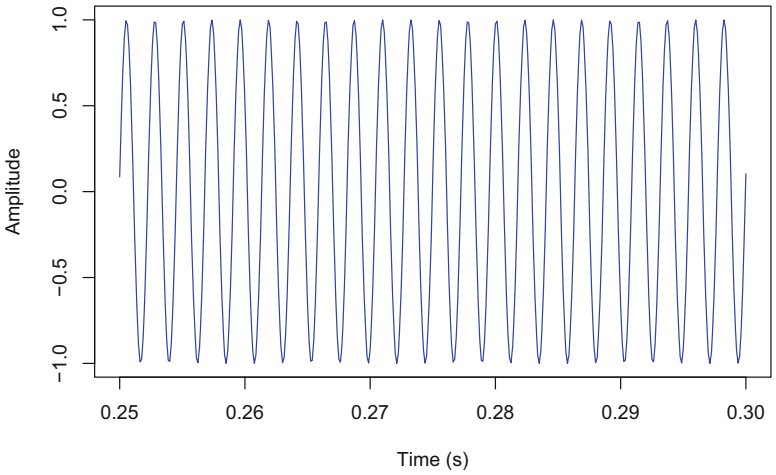


Fig. 4.1 Sound as a time series. This is a 0.05 s sound with a carrier frequency of 440 Hz and a sampling frequency of 8000 Hz. The plot was created with the function `plot()` applied to a `ts` object

Note that the length of `ts.sound` is not 8000 but 8001. Data of `ts` objects are actually recycled: the last value, `ts.sound[8001]`, is the same as the first value, `ts.sound[1]`.

Such `ts` object can be manipulated. For instance, a part of the sound can be extracted, or cut, with the function `window()`, and later plot to visualize a section of the time series (Fig. 4.1):

```
ts.sound.sel <- window(ts.sound, start=0.25, end=0.30)
plot(ts.sound.sel, xlab="Time (s)", ylab="Amplitude", col="blue")
```

Multiple time series, i.e., a time series made of several time series sampled together, can be stored in `ts` objects as several columns of a single matrix. Such objects are defined by two classes, `ts` and `mts`. Each column can describe a sound such that a two-column matrix could include the two channels of a stereo sound. Here we build a `mts` object with a pair of columns, each column containing the same `v.sound` vector:

```
mts.sound <- ts(cbind(v.sound, v.sound),
                start=0, end=1, frequency=8000)
is.mts(mts.sound)      # class
[1] TRUE
```

(continued)

```
dim(mts.sound)         # dimensions
[1] 8001    2
start(mts.sound)       # start of recording
[1] 0 1
end(mts.sound)         # end of recording
[1] 1 1
deltat(mts.sound)      # time resolution, here 125 ms
[1] 0.000125
frequency(mts.sound)   # sampling frequency, here 8000 Hz
[1] 8000
```

4.1.3 `audioSample` Class of the Package `audio`

The package `audio` has a specific class of object, the class `audioSample`, that facilitates the generation, import, and export of `.wav` sound samples. This sound-specific class is essentially a numeric vector for mono audio samples and a numeric matrix with two rows for stereo audio samples. The function `audioSample()` and `as.audioSample()` can be used respectively to generate an `audioSample` object and to coerce the class of an object into the `audioSample` class. The following command creates a new `audioSample` object based on the vector `v.sound` generated previously. The sampling frequency f_s is specified with the argument `rate` and the quantization with the argument `bits`:

```
sample.sound <- audioSample(v.sound, rate=8000, bits=16)
```

It is not a good idea to print an `audioSample` object as the console will return all the numeric values, or audio samples, of the object. It is therefore advised to use either the function `str()` that returns the structure of an object or the function `head()` that prints only the six first items of an object. The sampling frequency f_s can be accessed with `objectname$rate`. Quantization can be obtained with `objectname$bits`. The generic function `summary()` can also be called to get the main statistics of the sample values:

```
str(sample.sound)
Class 'audioSample'  atomic [1:8000] 0 0.339 0.637 0.861 0.982 ...
  ..- attr(*, "rate")= num 8000
  ..- attr(*, "bits")= int 16
head(sample.sound)
sample rate: 8000Hz, mono, 16-bits
[1] 0.0000000 0.3387786 0.6374906 0.8608080 0.9823196
[6] 0.9876545
sample.sound$rate
[1] 8000
sample.sound$bits
[1] 16
summary(sample.sound)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.000  -0.707   0.000   0.000   0.707   1.000
```

The duration in s of the file is not provided but can be easily obtained by dividing the number of samples per the sampling frequency f_s:

```
length(sample.sound)/sample.sound$rate
[1] 1
```

The duration can be reached in a more convenient way with the `seewave` function `duration()`:

```
library(seewave)
duration(sample.sound)
[1] 1
```

4.1.4 `sound` Class of the Package `phonTools`

The package `phonTools`, which is dedicated to phonetics, has a specific class named `sound`. An object of class `sound` can be generated with the function `makesound()` as in:

```
sound.sound <- makesound(v.sound, fs=8000)
```

where `fs` is the sampling frequency f_s. The object is a list with a print method associated:

```
sound.sound

      Sound Object

   Read from file:         v.sound.wav
   Sampling frequency:     8000  Hz
   Duration:               1000  ms
   Number of Samples:      8000
str(sound.sound)
List of 5
 $ filename  : chr "v.sound.wav"
 $ fs        : num 8000
 $ numSamples: int 8000
 $ duration  : num 1000
 $ sound     : Time-Series [1:8000] from 0 to 1: 0 0.339 0.637
 0.861 0.982 ...
 - attr(*, "class")= chr "sound"
```

The list is made of five items: `$filename` is a predefined name for a `.wav` file if the sound is exported or the name of the `.wav` file if the sound was imported into `R`, `$fs` the sampling frequency f_s, `$duration` the duration in ms, and `$sound.sound` the data. The duration can also be obtained with `duration()` of seewave:

```
duration(sound.sound)
[1] 1
```

4.1.5 *`Wave` Class of the Package `tuneR`*

The class `Wave` of the package `tuneR` can also handle sound samples by reading `.wav` files. `Wave` objects can be directly generated with the function `Wave()` specifying the sampling frequency and the number of bits (quantization):

```
wave.sound <- Wave(v.sound, samp.rate=8000, bit=16)
```

`Wave` is a `S4` class that includes six slots that can be listed with `str()`:

```
str(wave.sound)
Formal class 'Wave' [package "tuneR"] with 6 slots
  ..@ left     : num [1:8000] 0 0.339 0.637 0.861 0.982 ...
  ..@ right    : num(0)
  ..@ stereo   : logi FALSE
  ..@ samp.rate: num 8000
  ..@ bit      : num 16
  ..@ pcm      : logi TRUE
```

The data are stored in `@left` for mono audio samples and in `@left` and `@right` slots for stereo audio samples. The slot `@stereo` is a logical that indicates whether the audio sample is mono or stereo. The slot `@samp.rate` is the sampling frequency f_s, and `@bit` is the quantization ranging from 8 to 64 by power of 2. The last slot, named `@pcm`, is a logical that specifies the format of the audio sample, either PCM or IEEE. To get all of this information separately, it is necessary to use the `@` indexing of `S4` object class:

```
wave.sound@samp.rate # sampling frequency
[1] 8000
wave.sound@stereo    # stereo or mono
[1] FALSE
wave.sound@bit       # digitization depth
[1] 16
wave.sound@pcm       # PCM or IEEE format
[1] TRUE
length(wave.sound)   # number of samples
[1] 8000
```

There are print and summary methods for `Wave` objects making the access to the attributes easy:

```
summary(wave.sound)

Wave Object
Number of Samples:      8000
Duration (seconds):     1
Samplingrate (Hertz):   8000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

(continued)

```
Summary statistics for channel(s):

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.000  -0.707   0.000   0.000   0.707   1.000
```

As for `audioSample` objects, the duration in seconds of the file is not provided but can be obtained with:

```
length(wave.sound)/wave.sound@samp.rate
[1] 1
```

or with `duration()` of `seewave`:

```
duration(wave.sound)
[1] 1
```

Since `tuneR` version 1.0-0, the `Wave` class definition has been extended to the class `WaveMC` to take into account sounds generated or recorded on more than two channels. In `WaveMC`, the different channels are placed in a single slot, named `@.Data`. This slot is a numeric matrix where each column is representing one channel. In the following example, we generate a five-channel object:

```
data <- matrix(rep(v.sound,5), ncol=5)
head(data)
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[2,] 0.3387786 0.3387786 0.3387786 0.3387786 0.3387786
[3,] 0.6374906 0.6374906 0.6374906 0.6374906 0.6374906
[4,] 0.8608080 0.8608080 0.8608080 0.8608080 0.8608080
[5,] 0.9823196 0.9823196 0.9823196 0.9823196 0.9823196
[6,] 0.9876545 0.9876545 0.9876545 0.9876545 0.9876545
waveMC.sound <- WaveMC(data, samp.rate=8000, bit=16)
str(waveMC.sound)
Formal class 'WaveMC' [package "tuneR"] with 4 slots
  ..@ .Data    : num [1:8000, 1:5] 0 0.339 0.637 0.861 0.982 ...
  ..@ samp.rate: num 8000
  ..@ bit      : num 16
  ..@ pcm      : logi TRUE
```

The number of channels can be obtained with the function `nchannel()`:

```
nchannel(waveMC.sound)
[1] 5
```

Objects of class `Wave` generated with previous versions of `tuneR` can be updated with the function `updateWave()` to match the new definition. Eventually, the function `equalWave()` checks whether two `Wave` objects are compatibles for edition, i.e., whether the two objects are of the same class (`Wave` / `WaveMC`) and have the sample sampling frequency, the same quantization, and the same number of channels.

4.2 How to Read (Load) a Sound

4.2.1 `.wav` Files

External `.wav` files can be imported into an `R` session using either the function `load.wave()` of the package `audio`, the function `loadsound()` of the package `phonTools`, or the function `readWave()` of the package `tuneR`. For instance, a file named `tuning-fork.wav` stored in a subdirectory named `sample` can be loaded with the following command:

```
sound <- load.wave("sample/tuning-fork.wav") # audio solution
sound <- loadsound("sample/tuning-fork.wav") # phonTools solution
sound <- readWave("sample/tuning-fork.wav")  # tuneR solution
```

If all three functions do the same, which one then to prefer? The function `readWave()` has several options that makes it more fancy.

First, `readWave()` can read a section of a `.wav` file. This can be extremely useful when only a part of a long file should be treated. Such a selection can save a lot of memory. This selective loading is achieved by using the arguments `from` and `to` in conjunction with the argument `units` that specifies the sampling units, either `"samples"`, `"minutes"`, or `"hours"`. For instance, the following instruction read the file from the 1000th to the 2000th sample:

```
selection <- readWave("sample/tuning-fork.wav",
                      from=1000, to=2000,
                      units="samples")
```

Reading a selection from 0.25 to 0.75 s would be:

```
selection <- readWave("sample/tuning-fork.wav",
                      from=0.25, to=0.75,
                      units="seconds")
```

Second, the function `readWave()` can have access to the metadata of a `.wav` file without reading the data. This means that the file is not really loaded, occupying memory space, but that information on its sampling frequency, quantization, number of samples, or length can be obtained. This is achieved by turning the argument `header` to `TRUE`:

```
hdr <- readWave("sample/tuning-fork.wav", header=TRUE)
hdr
$sample.rate
[1] 44100

$channels
[1] 1

$bits
[1] 16

$samples
[1] 44100
```

Third, `readWave()` import the data as they are without trying to scale values between -1 and $+1$. This has two main advantages: (1) this saves memory as decimal number as those found between -1 and $+1$ take more memory than integers, and (2) the raw amplitude of the file can be estimated (see Chap. 7).

Fourth, `readWave()` can read multichannel files just by setting the argument `toWaveMC` to `TRUE`.

DIY 4.1 — How to read a single channel of a stereo file

How can we import only the left (respectively right) channel of a stereo `.wav` file with `tuneR`? The idea is to import as a numeric vector the left channel using the S4 `@left` slot of the `Wave` object, to read the header of the `.wav` stereo file, and to combine these data to create a new mono `Wave` object:

```
left <- readWave("sample/tuning-fork-stereo.wav")@left
hdr <- readWave("sample/tuning-fork-stereo.wav",
                header=TRUE)
left <- Wave(left, samp.rate=hdr$sample.rate, bit=hdr$bits)
left

Wave Object
Number of Samples:      33408
Duration (seconds):     1.04
Samplingrate (Hertz):   32000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Eventually, the function `checkwavs()` of the package `warbleR` offers the possibility to test whether one or several `.wav` files can be read a priori by `readWave()`. The function has no specific arguments and works directly with the files found in the working directory. It is therefore mandatory to first use `setwd()` to select the directory where the `.wav` files are stored, in our case, the directory `sample`:

```
library(warbleR)
setwd("sample") # change the working directory
checkwavs()     # check
setwd("../")    # change back the working directory
```

To conclude, it seems that `readWave()` has more options than `load.wave()` `loadsound()` and should be preferred in most cases. This implies that the `Wave` class should also be preferred in most cases.

4.2.2 `.mp3` Files

Accompanying the function `readWave()`, the function `readMP3()` of `tuneR` can decode and import `.mp3` files into a `Wave` object:

```
mp3 <- readMP3("sample/tuning-fork.mp3")
mp3

Wave Object
Number of Samples:      33408
Duration (seconds):     1.04
Samplingrate (Hertz):   32000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The package `monitoR` includes as well a function named `readMP3()` based on the eponymous function of `tuneR` but that has the advantage to have two arguments, `from` and `to`, that let the user specifying a time selection for the import. The function works only if the third-party software `mp3splt` is installed.[2] In the following example that fetched an `.mp3` file localized in the directory `sample`, we specify that we call the `monitoR` function with the `::` operator that specifies the use of `monitoR`:

```
mp3sel <- monitoR::readMP3("sample/tuning-fork.mp3",
                           from=0.5, to=1)
mp3sel

Wave Object
Number of Samples:      18432
Duration (seconds):     0.58
Samplingrate (Hertz):   32000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The returning object does not last 0.5 s as expected because the function cuts the `.mp3` file at frame boundaries and not at exact samples.

4.2.3 *From `.mp3` to `.wav` Files*

The function `mp32wav()` of the package `warbleR` uses `readMP3()` of `tuneR` to convert a batch of files stored in the working directory:

[2] http://mp3splt.sourceforge.net/

```
setwd("sample") # change the working directory
mp32wav()       # conversion of all .mp3 files into .wav files
setwd("../")    # change back the working directory
```

4.2.4 `.flac` Files

`R` cannot manage `.flac` files, but a wrapper of the external software `flac`[3] can be used to convert `flac` files into `.wav` files and vice versa. This opens the opportunity to save files in `flac` format and therefore to gain hard drive space:

```
# conversion 'tuning-fork.wav' into 'tuning-fork.flac'
wav2flac(file="sample/tuning-fork.wav")
# conversion 'tuning-fork.flac' into 'tuning-fork.wav'
wav2flac(file="sample/tuning-fork.flac", reverse=TRUE)
```

4.2.5 Local Files

Now that we know how to read the two most common sound file formats, we should be able to import a long list of files. Rather than to identify these files with a graphical file manager and to load them one by one, `R` provides functions to manage directories and files that can be very useful to import and work with a group of files. Imagine that we have a bundle of `.wav` files stored, here in a directory named `sample`, we would like to import successively. The first thing to do is to list the files of interest with the function `dir()`. This function has a very interesting argument `pattern` that can be used to filter the results. With this argument accepting regular expressions, we just need to run the following commands to select `.wav` and/or `.mp3` files:

```
dir("sample", pattern="wav$")         # .wav files
dir("sample", pattern="mp3$")         # .mp3 files
dir("sample", pattern="wav$|mp3$")    # .wav and .mp3 files
dir("sample", pattern="^synth.*wav$") # .wav starting with 'synth'
```

[3] http://flac.sourceforge.net/

The result of the `dir()` function is a character vector in which each item is a file name. We can save this vector in an object and use it into a loop to import successively the files. Here is an example that lists the `.wav` files of the `sample` directory which name starts with `"synth"`,[4] imports them temporarily, and stores their duration in a second vector named `duration`. We use `paste()` with the `sep` argument set to `"/"` to generate the right path to the working directory `sample`:

```
## files starting with 'synth' and ending with 'wav'
file.names <- dir("sample", pattern="^synth.*wav$")
## check file names
head(file.names)
[1] "synth-am-fm-1.wav" "synth-am-fm-2.wav"
[3] "synth-am-fm-3.wav" "synth-am-fm-4.wav"
[5] "synth-am-fm-5.wav" "synth-am-fm-6.wav"
## prepare a numeric vector to store the results
duration <- rep(NA, length(file.names))
## for loop around file.names
for (i in 1:length(file.names))
    {
        ## read the ith file
        tmp <- readWave(paste("sample", file.names[i], sep="/"))
        ## get the duration of the ith file
        duration[i] <- duration(tmp)
    }
# results rounded to 1 digit
round(duration, 1)
 [1]  1.0  1.0  1.0  1.0  1.0  3.0  3.0  1.0  3.0  1.0  1.0
[12]  1.0  1.0  1.0  1.0  1.0  1.0  1.0  6.0  3.9  1.0  1.0
[23]  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.4  1.4  1.4  1.4
[34]  7.9  1.1  0.7  0.1  1.0  0.1  0.1 56.0  0.2  0.2  0.2
[45]  5.8  1.0  1.0  2.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
[56]  1.0  1.0  1.0  1.0  2.0  1.0  1.0  1.0  2.8  1.0  1.0
[67]  1.0  1.0  1.0  1.0  1.5  2.1  0.5
```

4.2.6 Online Files

`download.file()` is a generic function to retrieve online files. This function opens the possibility to load the millions of audio files available on the internet, in particular those from the Xeno-Canto website[5] that provides more than 340,000 `.mp3` files of about 9700 species bird songs. Here is a short code to import a nice recording of the Tawny Owl *Strix aluco* deposited by Fernand Deroussen on Xeno-Canto website:

[4] These `.wav` files are those synthesized in Chap. 18.

[5] http://www.xeno-canto.org

```
url <- "http://www.xeno-canto.org/161948/download"
file <- "sample/161948.mp3"
download.file(url, destfile=file, quiet=TRUE)
owl <- readMP3(file)
```

We now print `owl` to see what it contains:

```
> owl

Wave Object
Number of Samples:      2549376
Duration (seconds):     57.81
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Stereo
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The package `warbleR` offers nice facilities to handle more directly Xeno-Canto files. The function `querxc()` sends a keyword request to Xeno-Canto database and then returns a `data.frame` containing all the metadata associated with files containing the keyword. This keyword should be either a species or a genus name. Here is a query with *Zonotrichia capensis*, the Latin name of the rufous-collared sparrow or tico-tico (Fig. 5.1):

```
qw <- querxc("Zonotrichia capensis", download=FALSE)
  |
  |                                                      |   0%
  |
  |++++++++++++++++++++++++                              |  50%
  |
  |++++++++++++++++++++++++++++++++++++++++++++++++++++++| 100%
dim(qw)
[1] 515  17
```

We can have a look at the metadata of the first recording found:

```
qw[1, ]
  Recording_ID       Genus Specific_epithet    Subspecies
1       349321 Zonotrichia         capensis costaricensis
            English_name    Recordist Country
```

(continued)

```
1 Rufous-collared Sparrow Ed Hutchings Ecuador
                                    Locality Latitude
1 Distrito Metropolitano de Quito, Pichincha   0.1038
  Longitude Vocalization_type
1  -78.6027  call, male, song
                                Audio_file
1 http://www.xeno-canto.org/349321/download
                                          License
1 http://creativecommons.org/licenses/by-nc-sa/4.0/
                                Url  Quality  Time
1 http://www.xeno-canto.org/349321 no score 12:00
        Date
1 2016-09-25
```

The `.mp3` files can be downloaded into the working directory by setting the argument `download` to `TRUE`. Here downloading hundreds of files might take a while. We could then wish to download only a selection of files, for instance, only the recordings from Brazil. We first build a new data.frame with the data we wish to select. The country of recording is specified in the column `"Country"`:

```
colnames(qw)
 [1] "Recording_ID"       "Genus"
 [3] "Specific_epithet"   "Subspecies"
 [5] "English_name"       "Recordist"
 [7] "Country"            "Locality"
 [9] "Latitude"           "Longitude"
[11] "Vocalization_type" "Audio_file"
[13] "License"            "Url"
[15] "Quality"            "Time"
[17] "Date"
```

so that Brazil can be selected with conditional indexing:

```
qw.brazil <- qw[qw$Country=="Brazil",]
dim(qw.brazil)
[1] 109  17
```

The Brazilian `.mp3` files can then be downloaded using the argument `X` of `querxc`

```
querxc(X=qw.brazil, download=TRUE)
```

A fancy way to visualize the localization of these files is to use the function `xcmaps()` that uses the function `map()` of the package `maps` (Fig. 4.2):

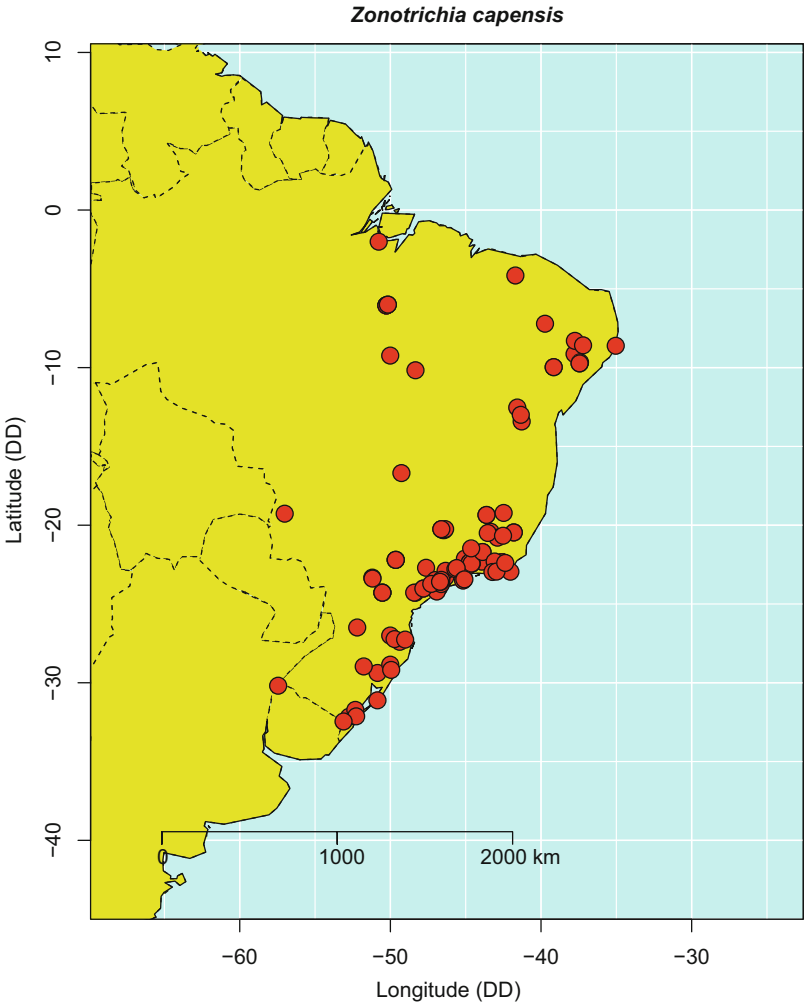


Fig. 4.2 Geographical map of Xeno-Canto recordings. The function `xcmaps()` of `warbleR` can return a map of a species recordings, here for the rufous-collared sparrow, or tico-tico, *Zonotrichia capensis*, recorded in Brazil

```
xcmaps(qw.brazil, img=FALSE)
```

The process works as well for a genus search: `xcmaps()` produces then a map for each species found in the genus. The maps can be saved externally in `.jpeg` files with `img=TRUE`.

```
qw.genus <- querxc("Zonotrichia", download = FALSE)
xcmaps(qw, img=TRUE)
```

The above instructions produce five image files each containing the map of one *Zonotrichia* species. The files are named according to species names, here: `Map of Zonotrichia albicollis recordings.jpeg`, `Map of Zonotrichia atricapilla recordings.jpeg`, `Map of Zonotrichia capensis recordings.jpeg`, `Map of Zonotrichia leucophrys recordings.jpeg`, and `Map of Zonotrichia querula recordings.jpeg`.

4.2.7 *Song Meter© Files*

The company Wildlife Acoustics© has developed autonomous digital recorders, the Song Meters SM2, SM3, and SM4, that are widely used in bioacoustics and ecoacoustics.[6] These devices generate audio files with specific names that include useful information related to the date and main features of the recording. This information is sometimes not straightforward to read as files look like they are written in a cryptic code, such as `"CNRS_0+1_20130824_153000.wav"`.

The function `songmeter()` of `seewave` can decompose these file names and return the information into a readable `data.frame` that details the prefix used to identify the recording unit, the microphones plugged in (`character`), the year (`numeric`), the month (`numeric`), the day (`numeric`), the hour (`numeric`), the minute (`numeric`), the second (`numeric`), the POSIX time of recording (`POSIX` format), and the occurrence of a geolocalization system:

[6] http://www.wildlifeacoustics.com/

We first create artificial file names:

```
file1 <- "MNHN_20141225_234500.wav"      # SM2 or SM4
file2 <- "CNRS_0+1_20130824_153000.wav"  # SM3 without geoloc.
file3 <- "PARIS_-0-_20150410$195550.wav" # SM3 with geoloc.
files <- c(file1, file2, file3)
```

that we submit to `songmeter()`:

```
songmeter(files)
    model prefix    mic year month day hour min sec
1 SM2/SM4   MNHN   <NA> 2014    12  25   23  45   0
2     SM3   CNRS stereo 2013     8  24   15  30   0
3     SM3  PARIS  monoL 2015     4  10   19  55  50
                 time   geo
1 2014-12-25 23:45:00    NA
2 2013-08-24 15:30:00 FALSE
3 2015-04-10 19:55:50  TRUE
```

4.3 How to Listen to a Sound

R is not a music player; it does not include a sound file reader. The trick to listen to a sound from R is to call an external player or audio driver. This external tool depends on the operating system. For Windows system, Windows Media Player[7] might be the most appropriate tool. For Mac OS the most common player is AudioUnits.[8] For Linux systems, the best solution is certainly to install SoX[9] that embeds a player named play. A free tool to all operating systems could be the well-known player VLC.[10] VLC has the advantage to be executable either with or without a graphical interface (through the terminal command `cvlc`); however it seems that VLC cannot handle sound with exotic sampling rates. As an example, a sound sampled at 2000 Hz could not be listened.

There are different ways to declare the default audio player and to play sound depending on the use of `audio` or `tuneR` packages. We will consider each package solution separately with functions summarized in Table 4.1.

[7] http://windows.microsoft.com/en-us/windows/windows-media

[8] https://developer.apple.com/library/content/documentation/MusicAudio/Conceptual/ AudioUnit-ProgrammingGuide/Introduction/Introduction.html

[9] http://sox.sourceforge.net/

[10] http://www.videolan.org/vlc/

Table 4.1 Equivalence between `audio`, `phonTools`, and `tuneR` functions dedicated to sound import and export

Description	`audio`	`phonTools`	`tuneR`
List of currently loaded and available audio drivers/players	`audio.drivers()`	–	–
Load a modular audio driver/player	`load.audio.driver()`	–	–
Name of the currently active audio driver/player	`current.audio.driver()`	–	`getWavPlayer()`
Set the default audio driver/player	`set.audio.driver()`	–	`setWavPlayer()`
Record a sound	`record()`	–	–
Read/load a `.wav` file	`load.wave()`	`loadsound()`	`readWave()`
Read/load a `.mp3` file	–	–	`readMP3()`
Play a `.wav` file	`play()`	`playsound()`	`play()`
Save into a `.wav` file	`save.wave()`	`writesound()`	`writeWave()`

4.3.1 With the Package `audio`

The package `audio` comes with predefined external players, named audio drivers. These are Windows Media Player for Windows, AudioUnits for Mac OS, and PortAudio for Unix. There are four functions to manage the audio drivers:

`audio.drivers()` lists **all** available audio drivers
`current.audio.driver()` returns **the** active audio driver
`set.audio.driver(name)` selects an audio driver to make it active
`load.audio.driver(path)` attempts to load a modular audio driver to make it active

With Windows 7, the two first commands return the following results:

```
> audio.drivers()
name                       description current
1  wmm Windows MultiMedia audio driver    TRUE
> current.audio.driver()
[1] "wmm"
```

With Mac OS X (10.7.5), we obtain:

```
> audio.drivers()
name description current 1 macosx AudioUnits (Mac OS X) driver
TRUE
> current.audio.driver()
[1] "macosx"
```

With Ubuntu 14.04 LTS, it is first necessary to install PortAudio with the following shell command (in the terminal) before to install `audio`. If this was not the case, it is necessary to install again the package for a right compilation and interaction with PortAudio[11]:

```
sudo apt-get install portaudio19-dev
```

Then we obtain:

```
audio.drivers()
       name      description current
1 portaudio PortAudio driver    TRUE
current.audio.driver()
[1] "portaudio"
```

Once the audio driver is loaded, the function `play()` can be used for playback action of `audioSample` objects. The following command broadcasts the `sample.sound` object we generated previously. The sound will be played in the background, which means without popping up the audio driver graphical interface:

```
play(sample.sound)
```

As soon as `play()` is called, the default audio driver opens and starts a session or sound instance. This sound instance can be controlled with a few functions that would correspond to the basic buttons of the player if it were open with a graphical interface:

`pause()` stops the playback
`rewind()` rewinds audio recording to start position
`resume()` resumes previously paused playback

[11]Note that the audio driver used with Ubuntu may change rapidly questioning the use of these `audio` functions with Ubuntu.

These functions can be used to program a playback session as in the following example where the playback is stopped, resumed, and restarted at its beginning:

```
# generates a 60 s sound with a carrier frequency at 440 Hz
long.sample <- sin(2*pi*440*seq(0, 60, length.out=8000*60))
long.sample <- audioSample(long.sample, 8000)
# starts playback and save the audio instance
# in an object named 'a'
a <- play(long.sample)
# pauses the playback
pause(a)
# resumes the playback where it was stopped
resume(a)
# pauses the playback again
pause(a)
# rewinds and resumes the playback at start position (restart)
rewind(a)
resume(a)
```

The package `audio` also includes a function `wait()` that asks the console to wait for either a certain amount of time in seconds or for a specific event like the playback of an object. The combination of these four functions opens the possibility to manage quite precisely a playback session as usually requested in playback experiments involving the broadcast of different stimuli following a predefined timing protocol.

The following exercise consists in playing back a control signal (object `control`), stopping the playback for a duration equal to duration of the control, and then playing back the test signal (object `test`). We first create artificial control and test objects:

```
control <- sin(2*pi*440*seq(0,1,length.out=8000))
control <- audioSample(control, 8000)
test <- sin(2*pi*880*seq(0,1,length.out=8000))
test <- audioSample(test, 8000)
```

We then include `play()` and `pause()` actions in a `repeat` loop (see Sect. 3.3.5.2) with five iterations. To run properly, this code should be executed at once, either by sending all the functions together to the console or by sourcing with `source()` an `.r` file containing the code (see Sect. 3.3.10):

```
i <- 1
repeat{
  wait(play(control))     # plays the control signal
  wait(duration(control)) # pause as long as control duration
  wait(play(test))        # plays the test signal
  wait(duration(test))    # pause as long as control duration
  i <- i+1                # iteration increment
  if(i>5) break           # breaks the loop after 5 iterations
  }
```

In some dynamic playback experiments, the user may have to interact with the playback process. We can imagine, for instance, that the user plays a first test (object `test`). Then if the animal reacts, the test is stopped; if the animal does not show any reaction, the test is repeated. The idea is therefore to ask the user if the test should be played back another time or stopped. We first write a new function, named `read.answer()`, which displays a question message in the console with the function `cat()` and stores the user response in a vector with the function `scan()`:

```
read.answer <- function(){
  cat("\n","Do you wish to play the sound again?",
      "\n", "Enter your choice 'y' [yes] or 'n' [no])",
      "\n")
  letter <- scan(what="character", n=1)
}
```

We then play the sound with `play()`, and we enter into a `repeat` loop. The loop uses the function `read.answer()` to check the user decision. If the user decides to play back another time, we use another time `play()`; otherwise we leave the loop with `break`:

```
## first play back with audio play() function
a <- play(test)
## repeat the action until the answer is no
repeat{
        answer <- read.answer()
        if(answer=="y") a <- play(test)
        if(answer=="n") break
}
```

Similarly to the previous case, this code should be placed in a `.r` file and executed with the function `source()` (see Sect. 3.3.10).

4.3.2 With the Package `phonTools`

The function `playsound()` of the package `phonTools` uses VLC as a background player. The path to VLC executable should be specified in the argument `path`. Here is the solution for Unix systems:

```
playsound(tico@left, fs=tico@samp.rate, path="/usr/bin/vlc")
```

4.3.3 With the Package `tuneR`

With `tuneR`, the setting of the default audio player is achieved with the function `setWavPlayer()` by providing the path to the executable file of the audio player chosen.

With a Windows system, the selection of Window Media Player could look like the following code, but note that the path will differ according to Windows version and that the path should be framed with simple quotes (`' '`) and double quotes (`""`):

```
setWavPlayer(
    ' "C:/Program Files/Windows Media Player/wmplayer.exe" '
)
```

The following command selects the application play on a Mac OS X:

```
setWavPlayer('/applications/play')
```

With a Linux (Ubuntu) system, setting the player play of SoX is rather simple:

```
setWavPlayer("play")
```

The function `getWavPlayer()` returns the default player. Think that the functions `setWavPlayer()` and `getWavPlayer()` work like default working directory functions `setwd()` and `getwd()`, respectively (see Sect. 3.3.1).

`tuneR` has a single function named `play()` to play back `Wave` objects. The following code plays the sound `wave.sound`. We have to multiply the object by

an arbitrary value of 32,000 to increase the amplitude that is originally between −1 and +1. Without this change, the sound would be inaudible:

```
play(32000*wave.sound)
```

If both packages are loaded in a single R session, a conflict will occur between both play() functions. To avoid such synonym issue, R has an operator to specify the package to use (Table 3.2). The command tuneR::play() will play the sound x using the function play(x) of tuneR, whereas the command audio::play(x) will use play() of audio. We could then use both play() with the following lines:

```
audio::play(sample.sound)
tuneR::play(wave.sound)
```

4.3.4 With the Package seewave

The play() functions of audio and tuneR have one main restriction: they work only with a single class object, audioSample or Wave class, respectively. However, we saw that a sound could also be written as a vector, a matrix, a data.frame, a ts, a mts, or a sound class object. The function listen() of seewave takes the best of tuneR::play() and accepts different object classes as input. The sampling frequency f_s should be specified in the argument f if the input object does not include the sampling frequency:

```
f <- 8000
listen(v.sound, f=f)
listen(m.sound, f=f)
listen(df.sound, f=f)
listen(ts.sound)
listen(sample.sound)
listen(sound.sound)
listen(wave.sound)
listen(waveMC.sound)
```

In addition, `listen()` has two arguments, `from` and `to`, to specify the time start and time end in seconds of the playback:

```
listen(wave.sound, from=0.25)
listen(wave.sound, from=0.25, to=0.75)
listen(wave.sound, to=0.75)
```

It is also possible to change the speed of playback by modifying the sampling frequency. Increasing artificially the sampling frequency implies that the player will read more samples per second than normally; the playback speed is faster. Reversely, decreasing the sampling frequency means that less samples will be read per second; the playback speed is slower. Note that the pitch of the sound will be modified accordingly, higher at a higher speed, lower at a lower speed:

```
f <- 8000                 # sampling frequency
listen(wave.sound, f=f)   # normal speed and pitch
listen(wave.sound, f=f/2) # twice slower and lower in frequency
listen(wave.sound, f=f*2) # twice faster and higher in frequency
```

This frequency change can be used to slow down a high-frequency sound, as those produced by some insects and bats, or reversely to fasten some low-frequency signals, as those recorded in seismology.

Eventually, `seewave` has a function `playlist()` that can handle a list of sound files stored in a directory. The order of files playback can be shuffled, and the list can be played back in a specific number of loops. The function is based on `play()` of `tuneR` and is optimized to work with play player of SoX. File music files stored in the directory `"sample"` could be listened in a randomized order and twice with (be aware that this can take a while...):

```
playlist("sample/", sample = TRUE, loop=2)
```

This playlist facility can be used when programming playback experiments, for instance, playback used to test the frequency sensitivity of a hearing system.

4.4 How to Record a Sound

The main advantage of the package `audio` is that sound can be directly acquired within an `R` session. This is achieved by first preparing a vector full of `NA` values. The number of `NA`s should exactly fit with the numbers of desired samples; this

means that the length of the vector should result from the multiplication of the time of recording in s per the sampling frequency f_s in Hz. Then the call of the function `record()` starts the recording session by replacing the `NA` values of the vector with sound samples. For instance, to get a mono sound of 5 s sampled at 22,050 Hz, we have to write:

```
d <- 5
f <- 22050
rec <- rep(NA_real_, d*f)
record(where=rec, rate=f, channels=1)
play(rec)
```

This function will work only if you have declared an audio driver (see Sect. 4.3). A recording session can be controlled with `pause()`, `rewind()`, `resume()`, and `wait()`. This function works properly with Windows and Mac Os but is highly experimental with Linux systems.

4.5 How to Write (Save) a Sound

Now that we have imported and listened to sounds, it might be necessary to save them out of `R` for another use. As for importing a `.wav` file, each package has a solution to export `R` objects as `.wav` files. The function of `audio` is `save.wave()`, the one of `phonTools` is `writesound()`, and the one of `tuneR` is `writeWave()`.

The three functions work the same way, the object to be saved is the first argument, and the file name with optionally a path is the second argument. The following instruction saves an `audioSample` object in `sample-test.wav` file in directory named `sample`:

```
save.wave(what=sample.sound, where="sample/sample-test.wav")
```

This will be translated with `phonTools` with:

```
writesound(samples=sound.sound, filename="sample/sample-test.wav")
```

The same can be achieved with `tuneR` but requires, in this case, an additional step. The object `wave.sound` is a `Wave` object with values between -1 and $+1$. It is necessary to convert these values on an appropriate bit scale, i.e., to

obtain values varying within a 2^n range. This conversion is ensured by the function `normalize()` of `tuneR`; the argument `unit` waits for the value n as a character, here `"16"` for a 16 bit quantization:

```
writeWave(object=normalize(wave.sound, unit="16"),
          filename="sample/wave-test.wav")
```

As neither `save.wave()` nor `writeWave()` does not return anything in the console, it might be worth to check whether the new files were created indeed. This is easily done with the function `dir()` with an appropriate regular expression given to the `pattern` argument:

```
dir("sample", pattern="*test*")
```

The files can be removed with the function `file.remove()` using the results of the previous `dir()` and pasting the character chain `"sample"` to complete the path:

```
file.remove(paste("sample",
                  dir("sample", pattern="*test*"), sep="/"))
```

`seewave` has a function named `savew()` that takes the best of `writeWave()` but has additional facilities. `savewav()` accepts all object classes used for sound and can generate a file name based on object name (the `file name` argument is not mandatory), and the sample values can be rescaled in any range. The `wave.sound` object can be therefore saved quickly with:

```
savewav(wave.sound)
```

The sampling frequency can be changed so to have a saved sound either slower or faster:

```
savewav(wave=wave.sound, f=44100) # twice faster
```

The values of the sound can also be rescaled with the argument `rescale`. In the following command, the sound is rescaled between -1500 and $+1500$:

```
savewav(wave=wave.sound, f=44100, rescale=c(-1500,1500))
```

This argument can be used to change the "volume" of the saved `.wav` file.

4.6 Tuning R

We have seen already a few instructions that we will need to call each time that we open an R session. Rather than repeating these actions every day, it is possible to declare them in the setting file of R. This file is a hidden file with no name, only the extension `.Rprofile`. The localization of this file depends on the OS and should be found with a file search. Once identified, the file can be edited with a text editor, including R GUIs, and all required instructions can be written. For instance, I have in my current `.RProfile`:

```
# welcome message
cat("Hello, it's time to have a strong coffee
     and to work hard on the book!\n")
# preferred packages
library(tuneR)
library(seewave)
# set 'play' of SoX as default player
setWavPlayer("play")
# set default working directory
# here a folder named 'R' on a computer named 'jerome'
# operated with a Linux system
setwd("~/jerome/R/")
```

To play with sound is not satisfactory, it is now time to get information from them. This is the matter of the next chapter that details the visualization and analysis processes.

Chapter 5
Display of the Wave

A challenge in sound description is to turn sound into an image. The most common way to visualize a sound is to show the variations of the instantaneous amplitude a against time t. We already used this display when introducing sound in Chap. 2 where almost all waves were depicted in a (a, t) frame. This 2D graphic is known as an **oscillogram**, the main visual output of an oscilloscope. The oscillogram is therefore a way to see the **waveform**. It is also possible to track the amplitude variations by plotting the **envelope** of the sound, that is, the shape of the sound along time. We will detail hereafter both options mainly exploring the song of a tropical bird, the rufous-collared sparrow *Zonotrichia capensis* (Fig. 5.1). The song was recorded by Thierry Aubin in Brazil where the bird is also known as tico-tico. The song, which is a nice pure and modulated sound made of four distinct notes, is included in an object named `tico` coming with `seewave`. It is available with the function `data`:

```
data(tico)
```

We can explore quickly what `tico` contains with:

```
tico

Wave Object
Number of Samples:      39578
Duration (seconds):     1.79
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
```

(continued)

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_5

Fig. 5.1 The rufous-collared sparrow *Zonotrichia capensis* also named tico-tico in Portuguese. Reproduced with the kind permission of Ladislav Nagy

```
Bit (8/16/24/32/64):     16
summary(tico@left)
      Min.    1st Qu.     Median       Mean    3rd Qu.
-18596.000   -583.750      0.000      0.495    585.750
      Max.
 19125.000
```

So `tico` is a mono PCM `Sample` object that was sampled at 22,050 Hz with a 16 bit depth during 1.79 s. The values of the recorded samples vary between −18,596 and +19,125.

5.1 Oscillogram

5.1.1 Simple Oscillogram

In `seewave`, the function to draw an oscillogram is `oscillo()`. The most elementary use of `oscillo()` is (Fig. 5.2):

```
oscillo(tico)
```

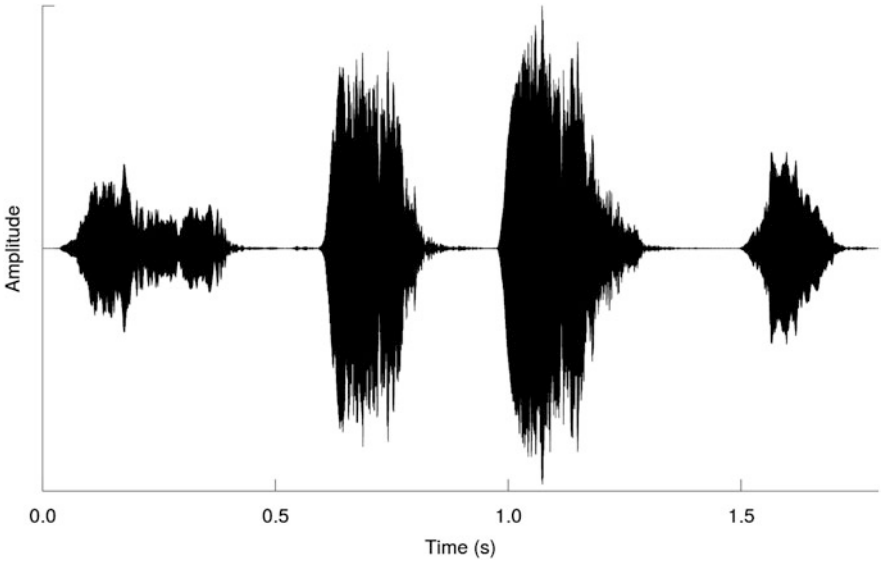


Fig. 5.2 A simple oscillogram. The waveform of the `tico` sound obtained with `oscillo(tico)`

`oscillo()` is a high-level plot function that comes with a long list of arguments that can be grouped according to the following topics:

input data `wave`, `f`, `from`, `to`
output data `identify`,
interactivity `zoom`, `scroll`,
general parameters `k`, `j`, `byrow`, `type`, `cex`, `bty`,
wave parameters `coly0`, `colwave`,
title parameters `title`, `coltitle`, `cextitle`, `fonttitle`,
axis parameters `labels`, `cexlab`, `fontlab`, `collab`, `colline`, `colaxis`, `cexaxis`, `fontaxis`, `tcl`, `xaxt`, `yaxt`.

Remember that you can display the arguments of a function using `args()`:

```
args(oscillo)
function (wave, f, from = NULL, to = NULL, scroll = NULL, zoom = FALSE,
    k = 1, j = 1, cex = NULL, labels = TRUE, tlab = "Time (s)",
    alab = "Amplitude", byrow = TRUE, identify = FALSE, nidentify = NULL,
    plot = TRUE, colwave = "black", coltitle = "black", cextitle = 1.2,
    fonttitle = 2, collab = "black", cexlab = 1, fontlab = 1,
    colline = "black", colaxis = "black", cexaxis = 1, fontaxis = 1,
    coly0 = "lightgrey", tcl = 0.5, title = FALSE, xaxt = "s",
    yaxt = "n", type = "l", bty = "l")
NULL
```

Like all `seewave` functions dealing directly with a wave, the first argument `wave` is an `R` object that includes the data, or samples. This object can be of various formats from a numeric `vector` to an `audio` or `Sample` object (see Sect. 4.1). The second argument, `f`, is the sampling frequency in Hz. This argument is mandatory only for objects that do not embed the sampling frequency, i.e., for `vector`, `matrix`, or `data.frame` objects. In the previous example, the argument `f` is not provided as `tico` includes the sampling frequency `f` in the `S4` slot `@samp.rate`. If `tico` were a `vector`, we should have written:

```
oscillo(tico, f=22050)
```

The oscillogram is a very simple plot; it only consists in plotting a X–Y plot with a line joining the successive sample values (see DIY box 5.1).

DIY 5.1 — How to draw your own oscillogram

A simple oscillogram can be achieved with base functions. The following code produces exactly the same output than Fig. 5.2 by simply combining the high-level `plot()` function and the low-level function `axis()`:

```
wave <- tico@left             # samples
time <- seq(0, 1.5, by=0.5)   # time
plot(wave,
     type="l",                # line type plot
     xaxs="i", yaxs="i",      # format of the axis
     xaxt="n", yaxt="n",      # no axis drawn
     xlab="Time (s)",         # x axis label
     ylab="Amplitude")        # y axis label
axis(side=1,                  # time axis built by hand
     at=time*tico@samp.rate+1,
     labels=time)
```

5.1.2 Axes

One of the most puzzling feature of `seewave` oscillogram is that the *y*-axis has neither scale nor unit. As we saw in Chap. 2, the amplitude of a sound can refer to different quantities, displacement, velocity, acceleration, pressure, or even voltage, and each quantity can be measured on a different scale (m, mm s^{-1}, cm s^{-2}, Pa, mV, etc.). In addition, referring to a scale requires that the sound was recorded and digitalized with a fully calibrated acquisition system, something that is rarely met.

All these constraints explain why the *y*-axis is left blank and why the scale should be considered without unit. However, if data were calibrated, it is still possible to change the *y*-axis. It is quite easy to display the *y*-axis with the low-level graphic function `axis()`:

```
oscillo(tico)
axis(side=2)
```

Now, imagine that we know that a sample value of 19,000 equals to 1.5 Pa, we can convert the data in Pa, change the axis label with the argument `alab` of `oscillo()`, and redraw the *y*-axis with `axis()` (Fig. 5.3):

```
tico.calibrated <- tico@left*1.5/19000
oscillo(tico.calibrated, f=tico@samp.rate, alab="Amplitude (Pa)")
axis(side=2, las=2)
```

It is also possible to remove the axes and to add a time scale bar with the functions `arrows()` and `text()` (Fig. 5.4):

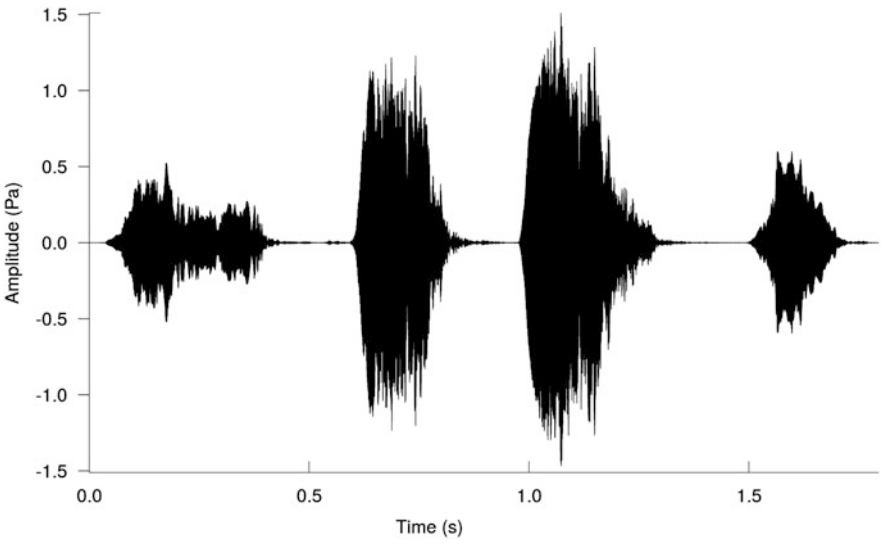


Fig. 5.3 Oscillogram with a calibrated amplitude. The default blank *y*-axis is tuned to display absolute values, here along a Pascal scale

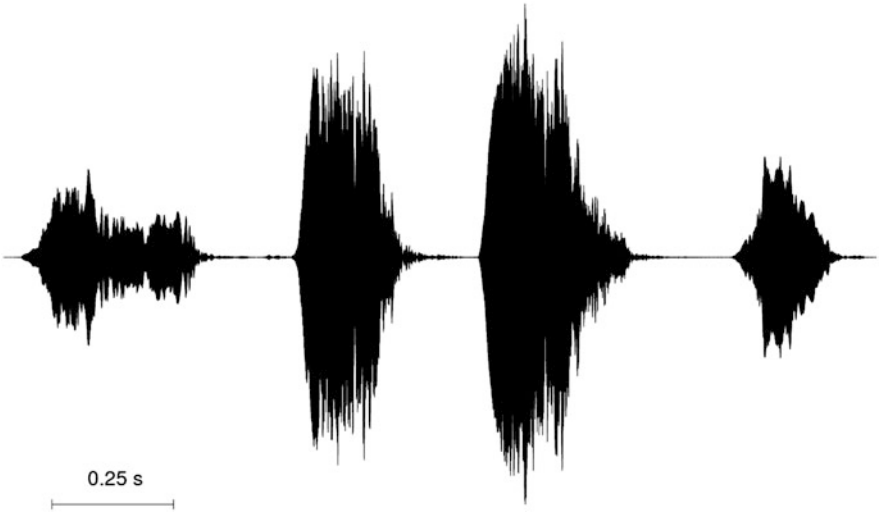


Fig. 5.4 Oscillogram axes. The axes were removed, and a time scale bar was added

```
oscillo(tico,
        colline="white",          # white lines
        colaxis="white",          # white axes
        xaxt="n", labels=FALSE)  # no axis labels
y <- min(tico@left)               # minimum of tico sample values
arrows(x0=0.1, y0=y,              # scale bar
       x1=0.35, y1=y,
       length=0.1, angle=90, code=3)
text(x=0.23, y=y+2000,            # legend of the scale bar
     labels="0.25 s")
```

5.1.3 Colors

`oscillo()` contains several parameters to change the color of the different items of the plot as exemplified in Fig. 5.5 obtained with the following instructions:

```
cex <- 1.25                  # main character size
col1 <- "brown4"             # first color
col2 <- "darkgreen"          # second color
oscillo(tico, title = TRUE, # add a title
        colwave = col1,      # color of the wave
        collab = col2,       # color of the axis labels
```

(continued)

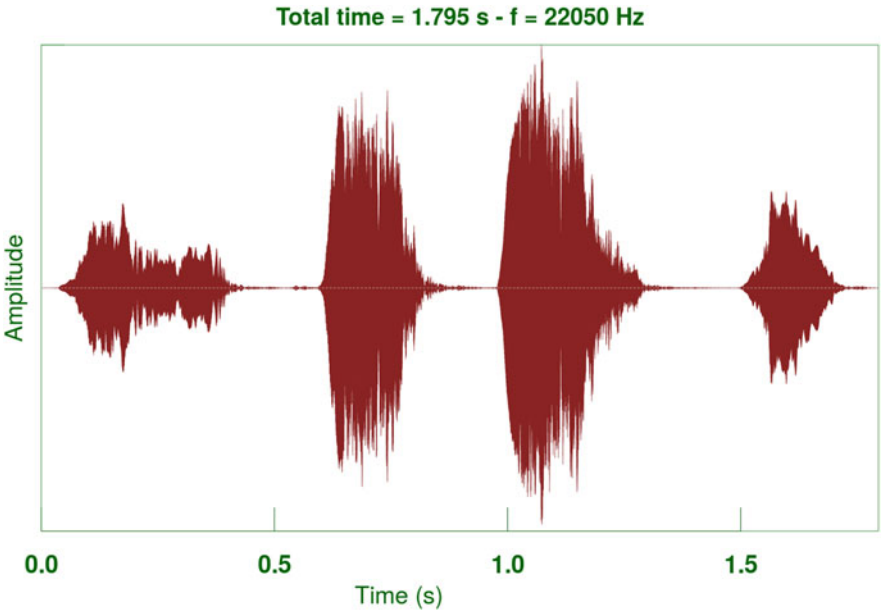


Fig. 5.5 Oscillogram colors. The colors of most graphical items can be changed to tune the oscillogram plot

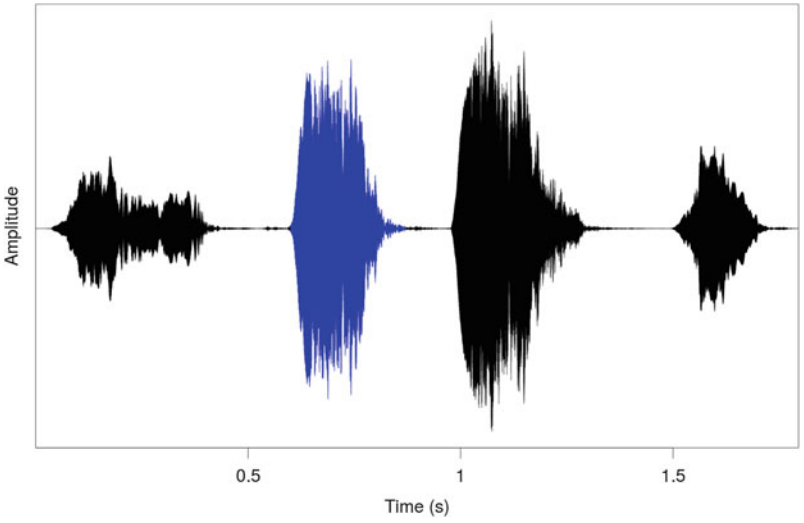
```
colline = col2,      # color of the axis lines
colaxis = col2,      # color of the axis annotations
cexlab = cex,        # size of the axis labels
cexaxis = cex,       # size of the axis annotations
fontaxis=2,          # font (bold) of the axis annotations
tcl = 1,             # length of axis tick marks
coltitle = col2,     # color of the title
bty = "o"            # framed box around the oscillogram
)
```

The function `oscillo()` does not provide a way to change the color of the wave against time, i.e., plotting the wave between t_1 and t_2 in one color and the wave outside t_1 and t_2 in another color. However, we saw that it was easy to draw an oscillogram with the function `plot()` (see DIY box 5.1). It is then possible, with a few tricks, to highlight a part of a wave with a different color as explained in the DIY box 5.2.

DIY 5.2 — How to highlight a part of an oscillogram with a different color

Here is the way to change the color of the wave against time. The example does not call `oscillo()` but the high-level graphical function `plot()` and the low-level functions `lines()` and `axis()`.

```
f <- tico@samp.rate              # sampling frequency
s <- tico@left                   # audio data
l <- length(s)                   # number of samples
t <- 1:l                         # time axis preparation
labels <- seq(0, 1.5, length=4) # time axis labels
from <- 0.6*22050                # highlight start (time*f)
to <- 0.87*22050                 # highlight end (time*f)
sel <- round(from):round(to)     # signal highlighted
outsel <- s[-sel]                # signal not highlighted
plot(x=t, y=s, type="n",         # blank plot
     xaxt="n", xaxs="i",         # x axis settings
     xlab="Time (s)",            # x axis label
     yaxt="n",                   # y axis settings
     ylab="Amplitude")           # y axis label
lines(x=t[-sel], y=outsel)       # signal unselected in black
lines(x=sel, y=s[sel], col=4)    # signal selected in blue
axis(side=1, at=labels*f,        # add time axis
     labels=labels)
```



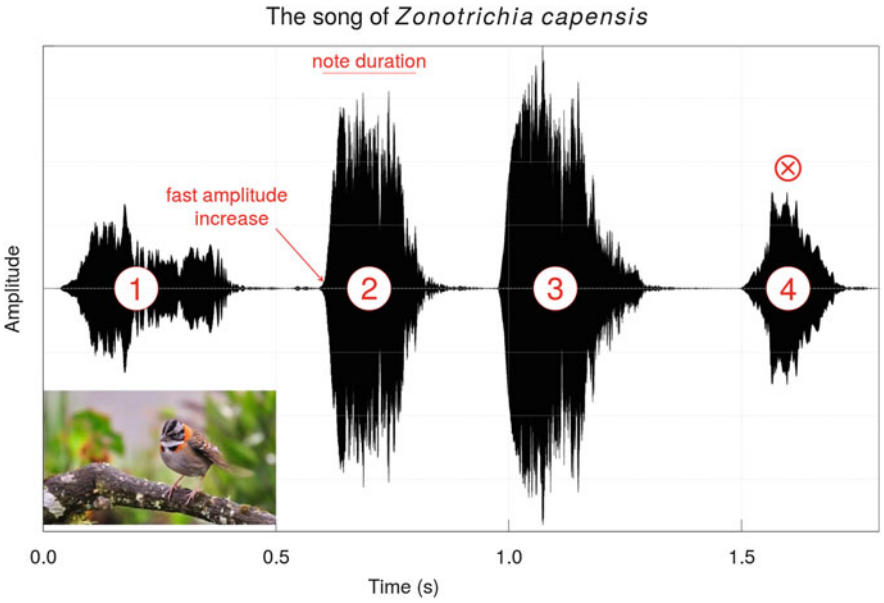


Fig. 5.6 Oscillogram decoration. Example of necessary and useless annotations on an oscillogram

5.1.4 Decoration and Annotation

An oscillogram needs sometimes to be annotated with text and/or geometric shapes. This can be achieved with low-level plot functions as for any `R` plot. Figure 5.6 shows the addition of labels, symbols, lines, a grid, and an inset with a picture.

Plotting a `.png` picture requires the installation and the call of the package `png` that provides an easy and simple way to read, write, and display bitmap `.png` images:

```
install.packages(png)
library(png)
```

The bitmap image, named `Zonotrichia_capensis_LNagy.png` and stored in a directory named `image`, is read with the function `readPNG()`:

```
img <- readPNG("image/Zonotrichia_capensis_LNagy.png")
```

Note that `.jpg` files can be similarly read with the function `readJPEG()` of the package `jpeg`.

The title of the oscillogram is prepared with the following character vector that uses the functions `expression()` and `italic()` to italicize the species name:

```
title <- expression(paste(
    "The song of ",
    italic(Zonotrichia), " ", italic(capensis),
    sep=" "))
```

The final plot is obtained with the following code that uses the function `rasterImage()` to display the picture (Fig. 5.6):

```
oscillo(tico, title=title)              # oscillogram with title
col <- "red"                            # color for annotations
pos <- c(0.2, 0.7, 1.1, 1.6)            # text position
points(x=pos, y=rep(0,4),               # white and red circles
       pch=21, cex=5,
       bg="white", col=col)
text(x=pos, y=0, cex=2,                 # numbers inside circles
     labels=as.character(1:4), col=col)
arrows(x0=0.5, y0=4700, x1=0.6, y1=600, # arrow
       length=0.1, col=col, lwd=2)
text(x=0.4, y=6500,                     # text for the arrow
     labels="fast amplitude \n increase",
     col=col)
text(x=1.6, y=9500,                     # symbol above
     expression(symbol("\304")),        # the last syllable
     cex=2, col=col)
segments(x0=0.6, y0=17000, x1=0.8,      # segment above
         col=col)                       # the second syllable
text(x=0.7, y=18000,                    # text accompanying
     labels="note duration", col=col)   # the segment
grid()                                  # axis grid
rasterImage(img,                        # image
            xleft=0, ybottom=min(tico@left),
            xright=0.5, ytop=-8000)
box()                                   # frame
```

We have seen how to change the color of a wave section (see Sect. 5.1.3), but highlighting a wave section—another kind of decoration—can also be achieved by drawing a rectangle behind the selection as proved by the following code (Fig. 5.7):

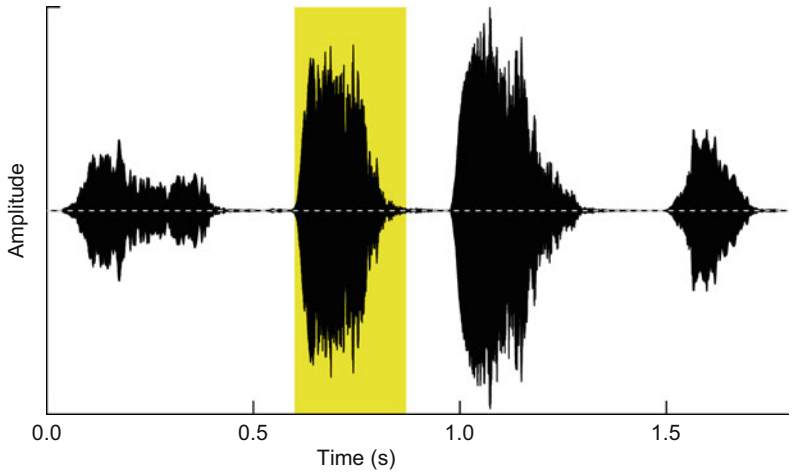


Fig. 5.7 Oscillogram highlight with a rectangle. The yellow background was added, thanks to the function `polygon()`

```
s <- tico@left                          # audio data
y <- max(abs(s))                        # rectangle upper limit
oscillo(tico, type="n")                 # blank plot
polygon(x=c(0.6,0.87,0.87,0.6),         # rectangle time limits
        y=c(-y, -y, y, y),              # rectangle amplitude limits
        col="yellow2", border="NA")     # decoration parameters
par(new=TRUE)                           # new plot layer
oscillo(tico)                           # oscillo over the rectangle
```

5.1.5 Zoom In

We already went through the main input data arguments, except `from` and `to`. These two arguments can be set in s to display an oscillogram section. A succession of zooms arranged in a four-row plate can be built with the following code (Fig. 5.8):

```
layout(matrix(1:4, nrow=4))  # plate layout with 4 lines
par(mar=c(4.5,4,2,2))        # internal margins slightly changed
cex <- 0.75                  # axis label size
oscillo(tico, cexlab=cex)    # complete wave
oscillo(tico, from=0.5, to=0.9, cexlab=cex)   # zoom 1
oscillo(tico, from=0.65, to=0.75, cexlab=cex) # zoom 2
oscillo(tico, from=0.68, to=0.70, cexlab=cex) # zoom 3
```

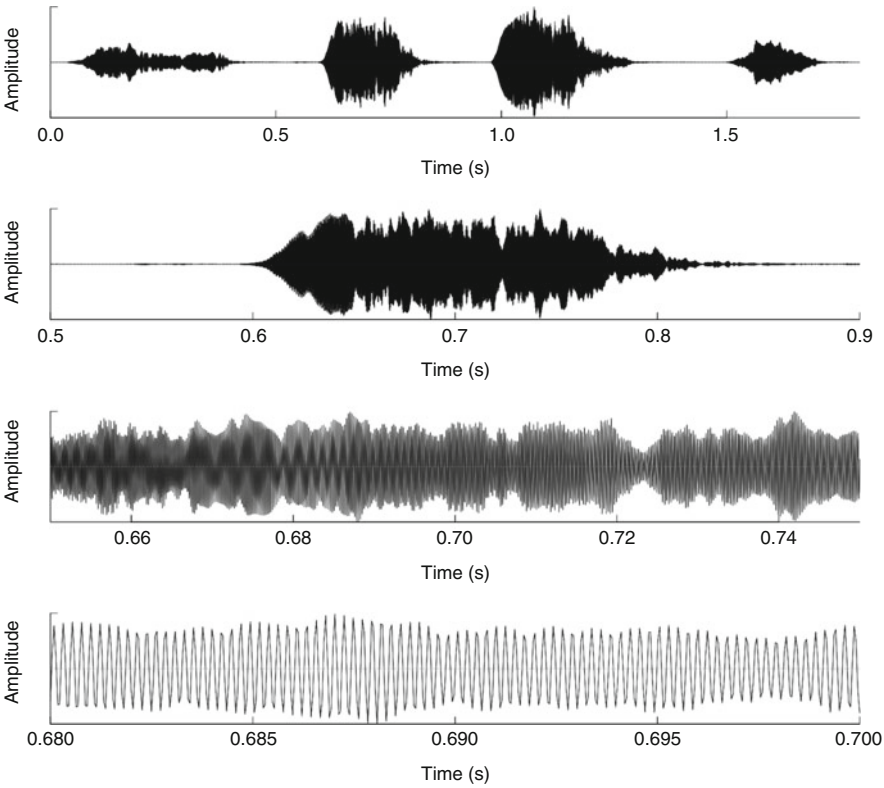


Fig. 5.8 Oscillogram time zoom in. The plate was built with four calls to the function `oscillo()` using different values for the arguments `from` and `to`

Note that another solution to produce such a plate of oscillograms could be to write a `for` loop along the rows of a matrix named `zoom` that embeds the `from` and `to` values. We first prepare the `zoom` matrix:

```
zoom <- matrix(c(NA, NA, 0.5, 0.9, 0.65, 0.75, 0.68, 0.70),
               byrow=TRUE,
               ncol=2)
zoom
     [,1] [,2]
[1,]   NA   NA
[2,] 0.50 0.90
[3,] 0.65 0.75
[4,] 0.68 0.70
n <- nrow(zoom)
```

and we then produce the plate of oscillograms with a loop:

```
layout(matrix(1:n, nrow=n))
par(mar=c(4.5,4,2,2))
cex <- 0.75
for(i in 1:n) {
    oscillo(tico,
            from=zoom[i,1], to=zoom[i,2],
            cexlab=cex)
}
```

The code can appear a bit more complex, but it is more universal as it can work for any `zoom` matrix.

5.1.6 A Bit of Interactivity

The `from` and `to` values can be fetched graphically by setting the argument `zoom` of `oscillo()` to `TRUE`. This interactive zoom works in a three-step process: (1) the complete wave is displayed, (2) the user chooses with the mouse two positions on the complete wave, and (3) these positions are used to display the wave selected. This zooming action works only once: it is not possible to zoom repeatedly.

The argument `scroll` is another way to interact a bit with the oscillogram display. This argument divides the complete wave in n sections or windows of equal length. A pop-up tool, generated with the package `rpanel`, allows to jump from one section to another one moving along the signal. The function `dynoscillo()` does the same with some more options regarding the section length and display. However, as pointed out in Sect. 3.4.1, the interactive exploration of sound, in particular long sound, is not optimized with `R`, and other tools, as Audacity, might be preferred.

5.1.7 Multiple Oscillogram

`seewave` offers a way to display a long wave by splitting it in several horizontal and/or vertical cells. The arguments `k` and `j` of `oscillo()` specify the number of line(s) and column(s) that defines these cells. The argument `byrow` sets whether the cells should be filled in by rows or by columns. The following code splits the signal into four lines (Fig. 5.9):

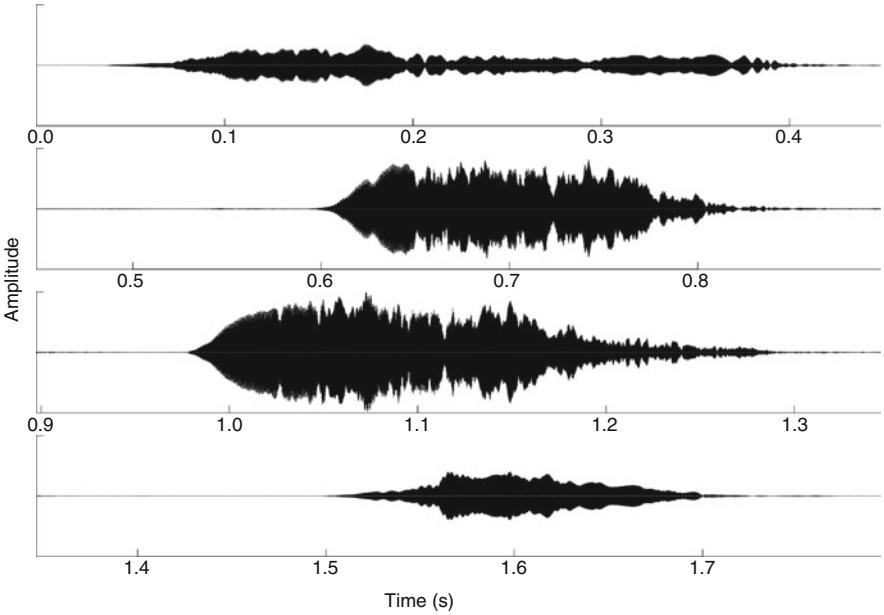


Fig. 5.9 Multi-line oscillogram. Using the argument `k`, the oscillogram is split in four sections of equal duration over four lines. The argument `j` can also be used to divide the oscillogram in columns

```
oscillo(tico, k=4)
```

A 2 × 2 matrix of oscillograms can be obtained with the following instruction:

```
oscillo(tico, k=2, j=2)
```

We just saw how to organize different oscillograms into a single plate, but it is also possible to overlay several oscillograms, thanks to the argument `new` of the graphical parametrization function `par()`. The following code demonstrates the plot of a `tico` sound that would have been recorded in noise and saved in a `Wave` object named `tico.noise` overplotted with the original `tico` sound (Fig. 5.10):

```
oscillo(tico.noise)          # plot the noisy recording in black
par(new=T)                   # open a new graphical layer
oscillo(tico,colwave="red") # visualize the clean recording in red
```

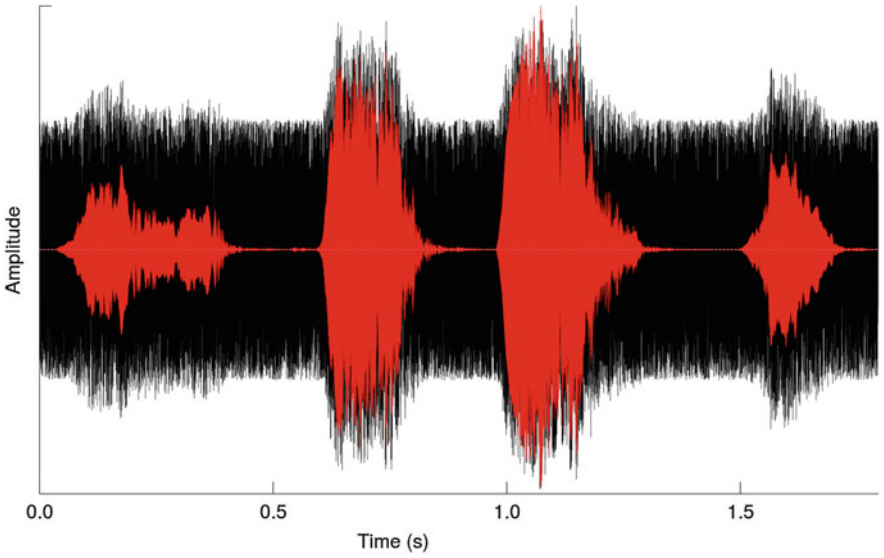


Fig. 5.10 Overplotting oscillograms. This figure demonstrates the overplot of two oscillograms, a noisy and a clean version of the dataset `tico`

`oscillo()` can manage mono files only, but the function `oscilloST()` can display stereo objects or two mono objects with almost the same list of graphical arguments.

5.2 Amplitude Envelope

5.2.1 Principle

The amplitude envelope is a time function that shows the amplitude modulations of the signal along time. The envelope is therefore the profile of sound energy over time. There are two different types of amplitude envelope.

The **absolute amplitude envelope** is the absolute value of the wave $|s(t)|$. All negative values of the wave are turned into positive values. This envelope is fast and easy to compute and to display (see DIY box 5.3).

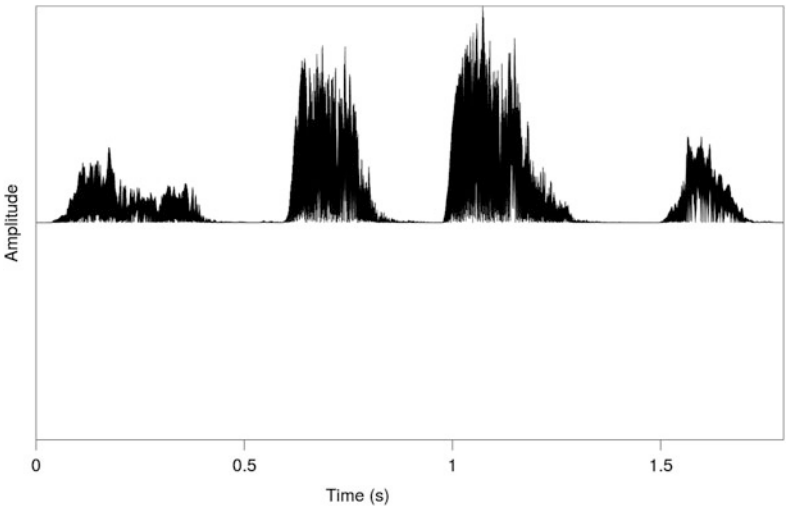
DIY 5.3 — How to compute and draw the absolute amplitude envelope

The absolute envelope of `tico` can be computed with a single line of code. This is simply the absolute value of the sample values:

```
envelope <- abs(tico@left)
```

It is then quite easy to plot the envelope as already done when plotting manually the oscillogram in the DIY box 5.1:

```
time <- seq(0, 1.5, by=0.5)    # time
max <- max(envelope)           # envelope maximum
plot(envelope,
     type="l",                 # line type plot
     xaxs="i", yaxs="i",       # format of the axis
     xaxt="n", yaxt="n",       # no axis drawn
     xlab="Time (s)",          # x axis label
     ylab="Amplitude",         # y axis label
     ylim=c(-max, max))        # y axis limits
axis(side=1,                   # time axis built by hand
     at=time*tico@samp.rate+1,
     labels=time)
```



However, the absolute envelope does not perfectly track the amplitude modulations as demonstrated on a signal with a triangular shape in Fig. 5.11.

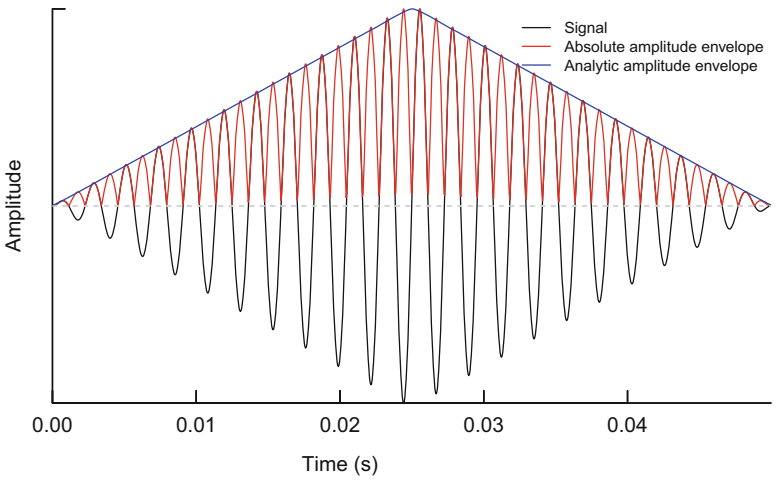


Fig. 5.11 Absolute and analytic (or Hilbert) amplitude envelope. The figure shows a 0.05 s signal with a triangular shape sampled at 22,050 Hz. Both absolute and analytic (or Hilbert) envelopes are overplotted to show their different behavior in the following amplitude modulations

The second type of amplitude envelope is the **analytic amplitude envelope** deriving from the analytic signal. It is behind the scope of this book to get into the complex details of the analytic signal, but following Mbu Nyamsi et al. (1994), we can try to understand where the analytic envelop comes from. The analytic signal, noted $\xi(t)$, is a complex-value signal which real part is the signal $s(t)$ and the imaginary part is the Hilbert transform $H(t)$. It is written as:

$$\xi(t) = s(t) + i\,H(t)$$

where $i^2 = -1$.

This is the rectangular version of the analytic signal. However, complex numbers can also be expressed in a trigonometric from, i.e., referring to the angle φ of $\xi(t)$ in the complex plane. The trigonometric form of $\xi(t)$ is:

$$\xi(t) = a(t)(\cos(\varphi(t)) + i\,\sin(\varphi(t)))$$

Knowing that a trigonometric form can be expressed with the base of the natural logarithm e (Euler's formula) following:

$$e^{ix} = \cos(x) + i\,\sin(x)$$

the analytic signal can be written in a more compact expression:

$$\xi(t) = a(t)e^{i\varphi(t)}$$

where $a(t)$ is the instantaneous amplitude and the angle $\varphi(t)$ is the instantaneous phase.

Then, the square modulus of the analytic signal can be obtained using these two expressions of the analytic signal. First we have:

$$\begin{aligned} |\xi(t)|^2 &= (s(t) + iH(t)) \times (s(t) - iH(t)) \\ &= s^2(t) + H^2(t) \end{aligned}$$

and, second, we have:

$$\begin{aligned} |\xi(t)|^2 &= a(t)e^{i\varphi(t)} \times a(t)e^{-i\varphi(t)} \\ &= a^2(t) \end{aligned}$$

Combining these two equations, we obtain the relation:

$$a^2(t) = s^2(t) + H^2(t)$$

and thus:

$$a(t) = \sqrt{s^2(t) + H^2(t)}$$

The amplitude envelope can therefore be obtained by computing the modulus of the analytic signal. As shown in Fig. 5.11, the analytic or Hilbert envelope returns the true amplitude profile. However, the analytic amplitude takes time to be computed and can be difficult to obtain for long waves.

5.2.2 *In Practice with* `seewave`

How can we get these envelopes with `R`? The function `env()` of `seewave` computes and plots either the absolute or the analytic (Hilbert) envelope by setting the argument `envt` to either `"abs"` or `"hil"`. The default value is turned to `"hil"` (Fig. 5.12):

```
env(tico)
```

Computing the Hilbert transform may take some time for long files; turning the argument `fftw` to `TRUE` can divide by 2 the computing time. This argument requires loading the package `fftw` (see Sect. 3.4.3).

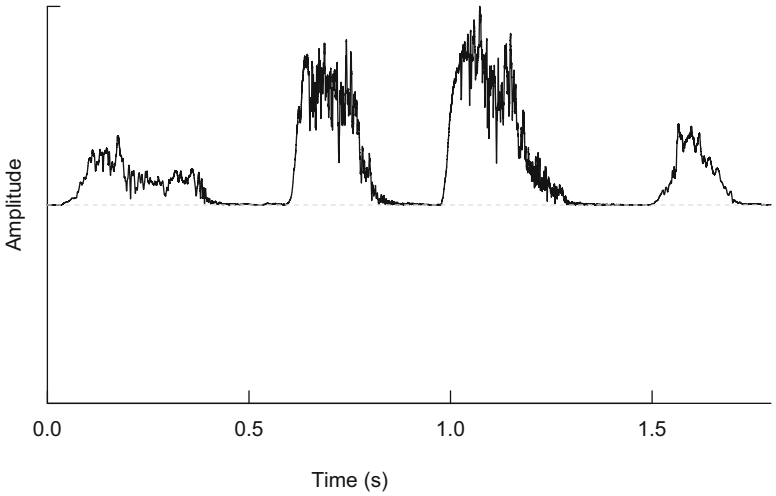


Fig. 5.12 Analytic envelope of `tico`. The envelope was obtained with the simple command `env(tico)`

The absolute amplitude envelope is obtained by using the argument `envt`:

```
env(tico, envt="abs")
```

The "..." at the end of `env()` indicates that all the arguments of `oscillo()` can be invoked from `env()`. This means that all the graphical arguments of `oscillo()` detailed above can be used. For instance, the following code zooms in time on the second syllable, changes the envelope color and adds a default title (Fig. 5.13):

```
env(tico, from=0.6, to=0.87, colwave="blue", title=TRUE)
```

5.2.3 Smoothing

Sometimes fast and/or unstable amplitude modulation waves return noisy amplitude envelopes that are difficult to display and to treat. In these cases, the amplitude envelope needs to be smoothed, or filtered, to remove non-informative data. Before to develop the three ways to smooth an envelope, the concept of the sliding window will be detailed as we will refer to it several times in the next chapters.

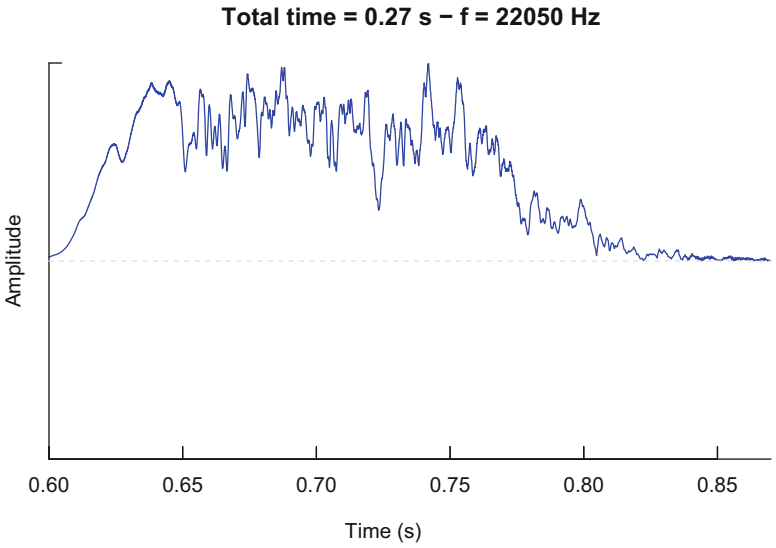


Fig. 5.13 Tuning of an amplitude envelope. The envelope of `tico` was zoomed in on the second syllable, the color of the envelope was changed, and a title was added

5.2.3.1 Sliding Window Process

The idea of the sliding window is rather simple: the input signal $s(t)$ is segmented in subsets that are delimited by a time window $w(t)$. This window has a fixed size or length characterized by a number of samples n or by a duration in s. The window is first positioned at the start of the signal from sample 1 to sample n. Then the window slides forward and frames the signal from sample $n + 1$ to sample $2n$ (Fig. 5.14, top panel). The sliding motion is repeated such as the window travels up to the end of the input signal. At each sliding step, any operation can be processed and saved. The output signal is a new time series smaller in length than the input signal. The sliding window process can be therefore viewed as data transformation and undersampling at the same time.

Undersampling means that the sampling rate is changed or in other words that the time resolution is lowered. If we start with an input signal digitalized at 22,050 Hz during 1 s and that we apply a sliding window including 512 samples, then the output signal will be made of $\lfloor 22{,}050 \div 512 \rfloor = 43$ values. The output signal still represents 1 s of observation but with a time resolution of $\Delta t = 1 \div 43 = 0.0232$ s. Such a rough time resolution might not be acceptable. A direct way to increase the time resolution is to shorten the window length but averaging few samples might not have important smoothing effect. An indirect solution consists in increasing the number of windows used along the signal. This can be achieved by overlapping successive windows. For example, if successive windows overlap by 50%, then the number of windows and so the time resolution doubles. Examples of overlapping windows are shown in Fig. 5.14 (middle and bottom panels) and a few examples of

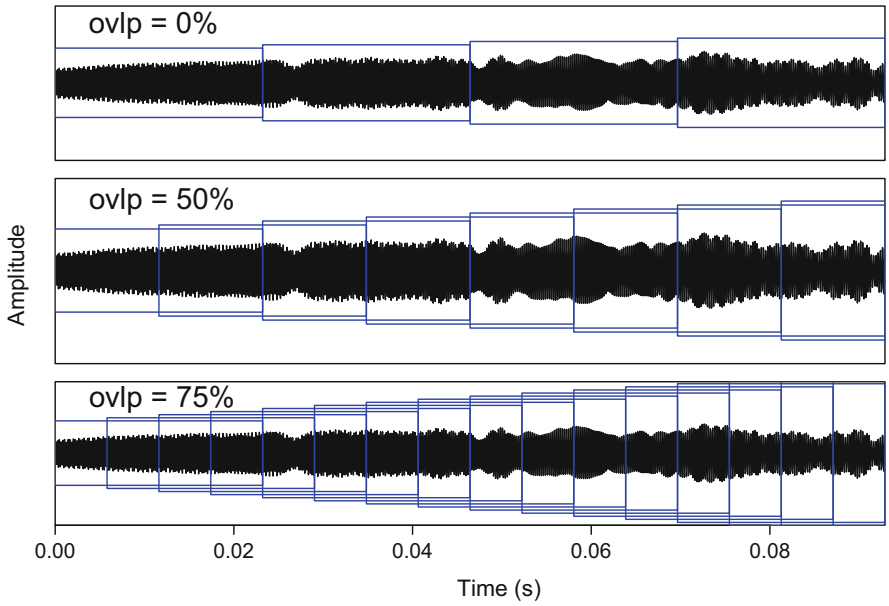


Fig. 5.14 Sliding window. Graphical representation of a window sliding along the time axis. The sound is sampled at 22,050 Hz; the window length is made of 512 samples which is equivalent to 0.0232 s. The overlap is 0% (top), 50% (middle), and 75% (bottom). The height of the window was artificially increased for a sake of clarity

time resolution depending on sampling rate, window length and overlap are given in Table 5.1.

5.2.3.2 Moving Average

The first possibility to clean a noisy amplitude envelope is to apply a moving or rolling average. The moving average principle relies on the sliding window concept detailed above. The values of samples framed by the window $w(t)$ are averaged. This leads to an averaged value per window slide and to an output time series made of successive averages. In `R`, the moving average option is set up with the argument `msmooth` of the function `env()`. This argument requires a numeric vector of length 2 with the first element specifying the window length in number of samples and the second element the window overlap in percentage (%). Setting `msmooth` to `c(512, 50)` slides a window of `512` samples jumping every `256` samples (50% of 512):

```
env(tico, msmooth=c(512,50))
```

Table 5.1 Time resolution of a sliding window

f	wl	ovlp	Δt
22,050	512	0	0.0232
22,050	512	50	0.0116
22,050	512	75	0.0058
22,050	512	95	0.0012
22,050	1024	0	0.0464
22,050	1024	50	0.0232
22,050	1024	75	0.0116
22,050	1024	95	0.0023
44,100	512	0	0.0116
44,100	512	50	0.0058
44,100	512	75	0.0029
44,100	512	95	0.0006
44,100	1024	0	0.0232
44,100	1024	50	0.0116
44,100	1024	75	0.0058
44,100	1024	95	0.0012

The table shows different time resolutions (Δt in s) in respect to sampling frequency (`f` in Hz), sliding window length (`wl` in number of samples), and sliding window overlap (`ovlp` in %)

The `tico` dataset was sampled at 22,050 Hz and includes 39,578 samples. If we apply a 128 sample window with an overlap of 75%, we obtain an envelope made of $(39{,}578 \div 128) \times 4 = 1233$ windows with a time resolution of $128 \div 22{,}050 = 0.0058$ s. Figure 5.15 shows the results of different window lengths and overlaps on `tico` analytic amplitude envelope. Smoothing can have a strong impact on the amplitude profile and should therefore be used with caution.

If absolute and analytic amplitude envelopes show quite important differences (Fig. 5.11), they tend to look very similar when applying a sliding average as shown in Fig. 5.16.

5.2.3.3 Moving Sum

Another way to smooth an envelope is to compute the sum of neighbor values. This method relies on a sliding window as well, but the window is slided sample by sample so that the smoothed output wave has the same number of samples than the input wave. The only parameter to provide is the length in number of samples of the sliding window. This is achieved with the argument `ssmooth` of the function `env()`. For instance, the following code computes the sum of envelope samples 1 to 512 (window 1), then 2 to 513 (window 2), and so on until the end of the wave:

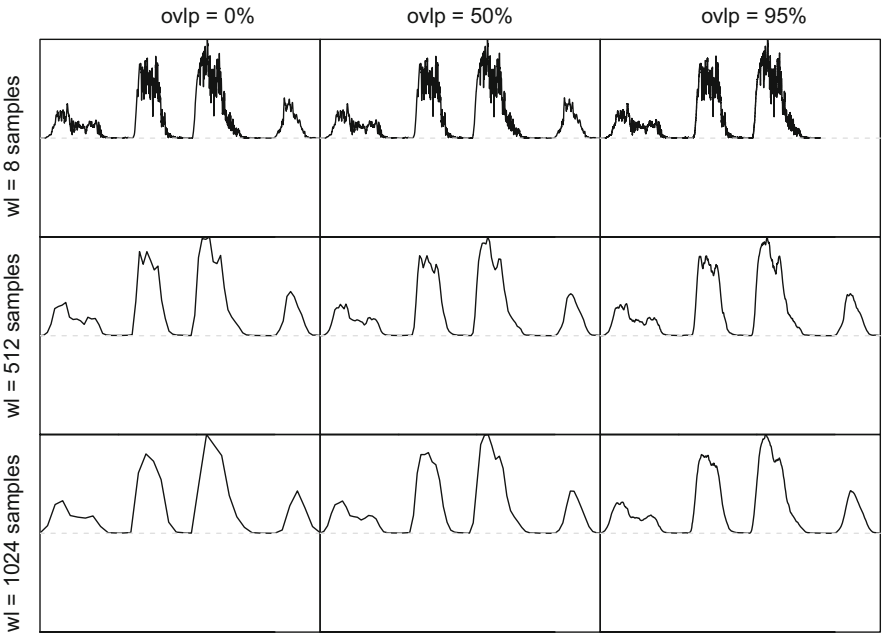


Fig. 5.15 Amplitude envelope smoothing. Example of the `tico` amplitude analytic envelope smoothed with different sliding window lengths and overlaps

```
env(tico, ssmooth=512)
```

Figure 5.17 shows the smoothed envelope of `tico` with different window lengths.

5.2.3.4 Moving Kernel

Smoothing kernel is a smoother that uses a weight function to average the observations, i.e., the sample values. There are different weights or kernel functions $K(x)$ available in `R`, namely, "daniell," "dirichlet," "fejer," and "modified.daniell." These functions are parametrized with a single parameter m called either the bandwidth, the spread, or the smoothing parameter. Increasing the value of m increases the smoothing. In `R`, kernel smoothing is available in the function `kernel()` with the first argument being the kernel function and the second argument the parameter m, so that a Daniell kernel function with $m = 8$ is created with:

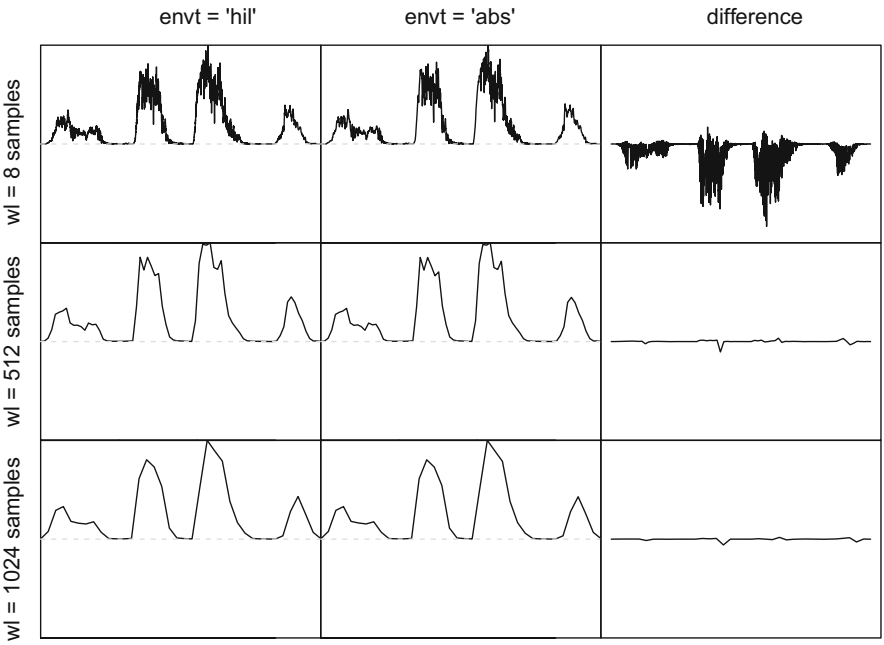


Fig. 5.16 Amplitude envelope types and smoothing with a sliding average. The plate shows the shape of the `tico` envelope either as an absolute amplitude envelope (`envt='abs'`) or as an analytic envelope (`envt='hil'`) for different average sliding window lengths. The difference by subtraction between the two envelopes is also shown

```
K <-  kernel("daniell", 8)
```

This function is reused with the argument `ksmooth` of the function `env()` to apply a kernel smoothing to the envelope:

```
env(tico, ksmooth=kernel("daniell",8))
```

Figure 5.18 shows the smoothed envelope of `tico` with the same Daniell kernel but with different smoothing parameters m. It is important to note that smoothing with a kernel function changes the length of the object. For instance, the smoothed envelope of Fig. 5.18 contains 39,562, 38,554, and 37,530 samples for $m = 8$, $m = 512$, and $m = 1024$, respectively, when the original signal has 39,578 samples. This induces a subsampling process that needs to be controlled for further analyses. The use of the kernel smoothing can be accelerated with the use of the argument `fftw`.

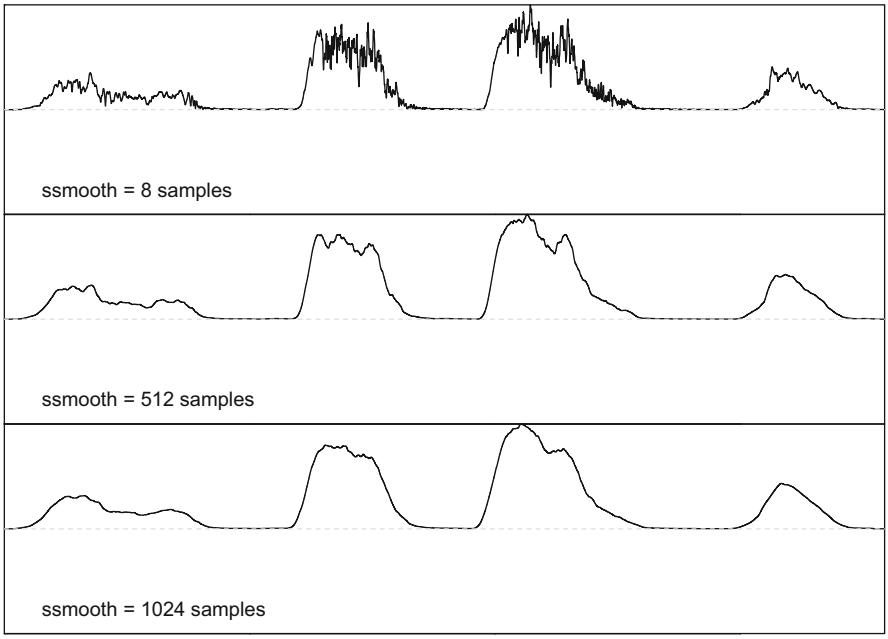


Fig. 5.17 Amplitude envelope smoothing by moving sum. The envelope is smoothed by computing the sum of neighbor values within a window containing 8, 512, or 1024 samples

5.2.3.5 Which Smoothing Method Should I Use?

To summarize there are three options to smooth an envelope: average (`msmooth`), sum (`ssmooth`), or kernel (`ksmooth`). The results do not differ so much, but the three possibilities mainly differ in the time they take to return the result (process time) and in the length of the vector returned (sampling). The fastest option is `msmooth`, followed by `ksmooth` and then `ssmooth`. The factor of process time varies with the length (duration) of the input wave. For instance, the process time ratio between `ssmooth` and `msmooth` was, respectively, 2.4 and 5.2 for a 18 s and a 180 s sound, and the ratio between `ssmooth` and `ksmooth` was 2.4 and 2.16 for the same objects. Regarding sampling, `ssmooth` has the great advantage not to change the number of samples when `ksmooth` slightly downsample and `msmooth` downsample quite drastically the wave. Therefore a trade-off arises between process and sampling. If the analysis requires to keep strictly the original number of samples, then `ssmooth` has to be used. If not, then the `msmooth` option might be preferred for its speed, in particular on short (<60 s) sounds.

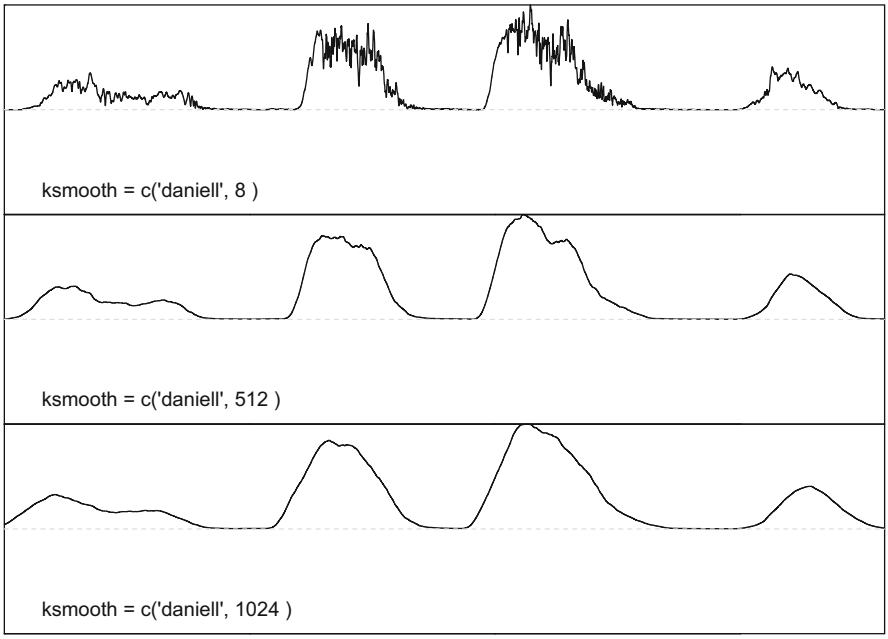


Fig. 5.18 Amplitude envelope smoothing with a kernel function. The envelope is smoothed by applying a kernel function parametrized with a smoothing parameter m

5.2.4 *In Practice with* `phonTools`

The package `phonTools` has a function named `powertrack()` which computes and displays another profile of the amplitude changes. The function operates a moving average on the square of the signal $s(t)$ weighted by a window function $w(t)$. This operation is quite similar to applying a moving average on the absolute amplitude envelope. The values returned by `powertrack()`, named "power," are converted into dB with a maximum value of 0 dB. The length and the overlap between the successive windows are set in ms with the arguments `windowlength` and `timestep`, respectively. We can apply `powertrack()` on `tico` with a window of 10 ms and an overlap of 5 ms = 50% (Fig. 5.19). We need to call directly the S4 slots of `tico` as `powertrack()` does not handle `Wave` objects:

```
powertrack(tico@left, timestep=5, windowlength=10,
           fs=tico@samp.rate)
```

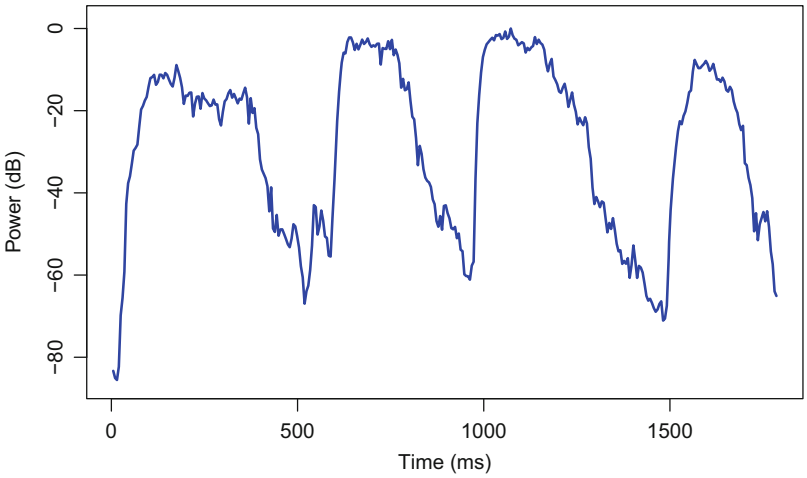


Fig. 5.19 Envelope following `powertrack()` function. The envelope of `tico` was obtained with the function `powertrack()` of `phonTools`. The envelope is obtained through a smoothing average on the square of the sound

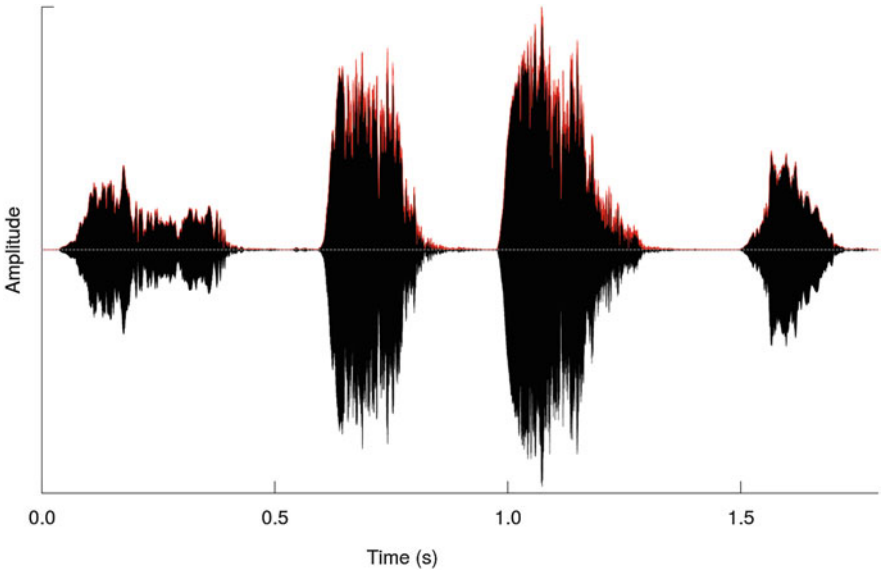


Fig. 5.20 Oscillogram and envelope. The analytic amplitude (or Hilbert) envelope is plotted in red over the oscillogram

5.3 Combining Oscillogram and Envelope

The oscillogram and the amplitude envelope are two important representations of sound that can be overlaid as shown in Fig. 5.20. This superposition is obtained by adding a layer to an `oscillo()` plot with `par(new=TRUE)`:

```
oscillo(tico)
par(new=TRUE)
env(tico, colwave=2)
```

Chapter 6
Edition

Sounds we recorded are not always exactly as we wished them to be. It may be necessary to edit them to get the best of them. In this chapter we will see how to change the sampling frequency, to control the channels and to manipulate sound according to time. Sound modifications involving amplitude and frequency analyses are detailed in Chaps. 14 and 15.

6.1 Resampling

We have seen in Chap. 2 that digital sound is sampled at a specific rate or sampling frequency f_s. The higher the sampling frequency is, the higher the time resolution is. However, a high f_s increases the amount of data to store so that sound management and analysis can be difficult. A high f_s also implies that the resolution of a frequency decomposition can be reduced (see Sect. 10.1.2.1). A solution to relax these constraints is to **downsample** the sound, that is to decrease f_s. Both `tuneR` and `seewave` provide solutions for downsampling. The function `downsample()` of `tuneR` is straightforward to use; it only requires the sound to be downsampled (input) and the sampling frequency of the downsampled sound (output). The following code downsamples `tico` from 22,050 to 11,025 Hz in a new object named `tico.downsampled`:

```
tico.downsampled <- downsample(tico, samp.rate=11025)
tico.downsampled@samp.rate
[1] 11025
```


J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_6

Downsampling can also be processed with the function `resamp()` of `seewave`. The advantage of this function is that it can accept not only `Wave` objects but all other sound classes. The first argument of `resamp()` is similarly the input sound, the second argument `f` is the optional sampling frequency of the input sound, and the third argument `g` is the sampling frequency of the output sound. The function `resamp()` has an extra argument, `output`, that can be found in several other `seewave` functions (see Chap. 18). This argument allows the user to control the class of the returned value. The argument `Output` is a character vector that can take the following values: `"matrix"` (default), `"wave"`, `"audioSample"`, or `"ts."` Here is the code to downsample `tico` from 22,050 to 11,025 Hz and save the output into a matrix named `tico.downsampled.m`:

```
tico.downsampled.m <- resamp(tico, g=11025)
```

In this case the sampling frequency of the output information is lost. To keep it as an attribute, it is necessary to change the `output` to a class embedding the sampling frequency as `Wave` does:

```
tico.downsampled.wave <- resamp(tico, g=11025, output="Wave")
tico.downsampled.wave@samp.rate
[1] 11025
```

Downsampling can be a risky process leading to a well-known phenomenon in acoustics, **aliasing**, as already detailed in Sect. 2.4.3. A usual mistake is to downsample a sound such that the new Nyquist frequency f_N—half the new sampling frequency—is too close from the highest frequency of the sound. For instance, imagine that a 5000 Hz sound was recorded with a sampling frequency of $f_s = 22{,}050$ Hz. Everything is fine at this stage: the 5000 Hz frequency band is far away from the Nyquist frequency $f_N = f_s \div 2 = 11{,}025$ Hz. However, if we downsample the sound to a new sampling frequency of 11,025 Hz, the 5000 Hz frequency band gets then very close the new Nyquist frequency $f_N = 11{,}025 \div 2 = 5512{,}5$ Hz. This induces time and frequency artifacts as shown in Fig. 6.1. A solution to avoid such issue is to apply a band-pass filter before to undersample as explained in Sect. 14.6.3.3.

It may also be necessary to **oversample**, that is, to increase the sampling frequency f_s. This can happen when several sounds have been sampled at different rates and that oversampling is required to have a similar sampling frequency for all sounds. Oversampling can be processed with `resamp()` by setting `g > f`. The

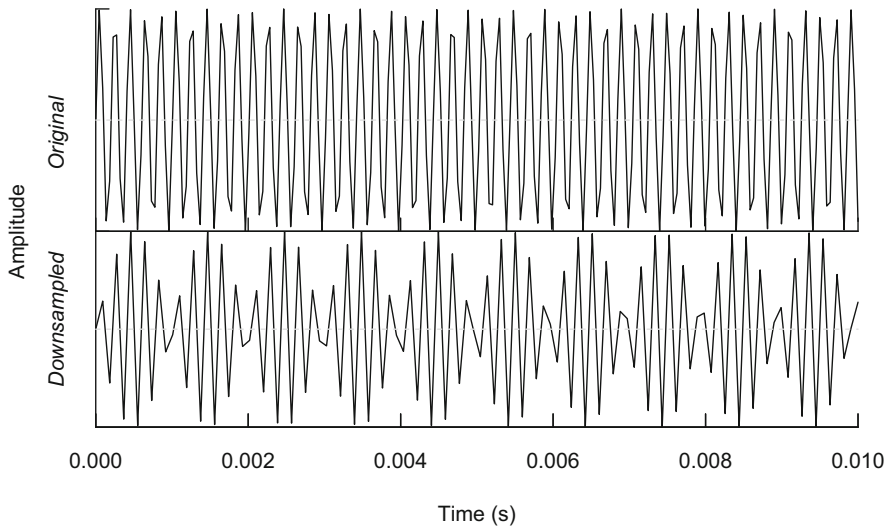


Fig. 6.1 Aliasing and downsampling. The original file (top) is a 5000 Hz pure tone sampled at 22,050 Hz. The same sound downsampled at 11,025 Hz clearly shows time and frequency artifacts (bottom)

following example is an oversampling of `tico` by a factor of 2:

```
tico.oversampled <- resamp(tico, g=tico@samp.rate*2,
                           output="Wave")
tico.oversampled@samp.rate
[1] 44100
```

Oversampling means that samples not existing in the input sound are artificially created in the output sound. The generation of new samples is achieved through a linear or constant interpolation process between existing samples. A linear process might not always be adapted to a sound wave. Oversampling should therefore be used with caution.

The sampling frequency f_s can be also changed to slow down or speed up the sound by using the function `Wave` of `tuneR`:

```
tico.fast <- Wave(left=tico@left, bit=tico@bit,
                  samp.rate=tico@samp.rate*2)
tico.slow <- Wave(left=tico@left, bit=tico@bit,
                  samp.rate=tico@samp.rate/2)
```

The result can be listened and saved in an external `.wav` file with:

```
listen(tico.slow) ; savewav(tico.slow)
listen(tico.fast) ; savewav(tico.fast)
```

6.2 Channels Managing

Sound can be recorded in mono (a single channel usually recorded on the left channel of the recorder device), in stereo (two channels, usually left and right channels), or in multichannel format that is on more than two channels. The number of channels of a `Wave` object can be accessed with the function `nchannel()` of `tuneR`. The order and the name of channels of multichannel objects are predefined by the position of the speakers. This information is stored in the dataset `MCnames` coming with `tuneR`:

```
MCnames
   id                  label name
1   1             Front Left   FL
2   2            Front Right   FR
3   3           Front Center   FC
4   4         Low Frequency   LF
5   5             Back Left   BL
6   6            Back Right   BR
7   7  Front Left of Center  FLC
8   8 Front Right of Center  FRC
9   9           Back Center   BC
10 10             Side Left   SL
11 11            Side Right   SR
12 12            Top Center   TC
13 13        Top Front Left  TFL
14 14      Top Front Center  TFC
15 15       Top Front Right  TFR
16 16         Top Back Left  TBL
17 17       Top Back Center  TBC
18 18        Top Back Right  TBR
```

As detailed in Chap. 4, the number of channels is stored in the `audioSample` and `Wave` objects. There are no specific channel functions in `audio`, but `tuneR` includes three functions for managing mono and stereo sounds. The function `stereo()` is used to generate a stereo file. We can then convert `tico`, which

is a mono sound, into a stereo sound with:

```
tico.stereo <- stereo(left=tico, right=tico)
```

and check if `tico.stereo` is indeed a stereo object:

```
tico.stereo@stereo
[1] TRUE
```

and even plot it with `oscilloST()` of `seewave` (Fig. 6.2):

```
oscilloST(tico.stereo)
```

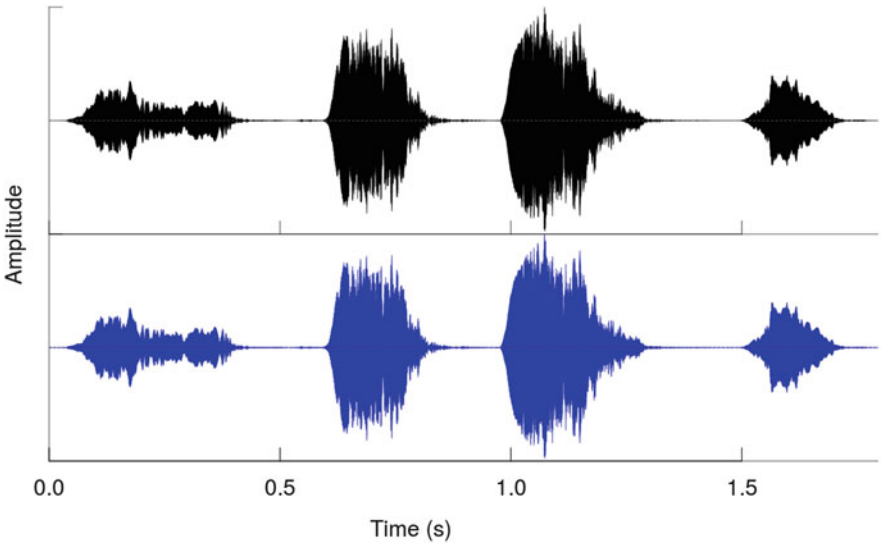


Fig. 6.2 Oscillogram of a stereo `Wave` object. The object `tico` was converted into a stereo `Wave` object with `stereo()` and plotted as an oscillogram with the function `oscilloST()`. The left channel is on the top and the right channel is at the bottom of the plot

The function `mono()` can convert a stereo sound into a mono sound, either by extracting a single channel or by averaging both channels into a single one:

```
tico.left <- mono(tico.stereo, "left")   # left channel
tico.right <- mono(tico.stereo, "right") # right channel
tico.both <- mono(tico.stereo, "both")   # left+right channels
```

All the objects returned by `mono()` are `Wave` objects. Remember that the data or samples of the channels can be accessed directly with the `S4` syntax `@` (see DIY box 6.1).

The function `channel()` can also manage channels. In particular, this function can mirror the left and right channel, that is, replace the left channel by the right channel and vice versa:

```
tico.mirror <- channel(tico.stereo, which="mirror")
```

The relative amplitudes of left and right channels can be balanced changing the stereo effect with `panorama()`, another function of `tuneR`. The function works with only two arguments: the input sound (either a stereo `Wave` object or a `WaveMC` object which only the two first channels will be considered) and a value, `pan`, varying between -1 and $+1$ that controls the mixture of the channels and therefore the virtual distance between the two loudspeakers. If `pan=1`, nothing is changed; if `pan=0`, the channels have the same amplitude and therefore are both centered. If `pan=-1`, the channels are mirrored. If `-1<pan<0`, the amplitude of the left channel is decreased so that the stereo effect seems to come from the right. Reversely, if `0<pan<1`, the amplitude of the left channel is increased so that the stereo effect is on the left. The following use of `panorama()` balances positively the stereo version of `tico` on the left:

```
pan <- panorama(tico.stereo, pan=0.5)
```

Mono and stereo sounds can be created from `WaveMC` objects through simple matrix indexing. The returned objects will be of class `WaveMC`. For instance, if we convert `tico` into `WaveMC` objects with three channels with the function `WaveMC()`:

```
tico.mc <- WaveMC(data=cbind(tico@left, tico@left, tico@left))
nchannel(tico.mc)
[1] 3
```

we then can extract any channel into a single channel `WaveMC` object, in other words a mono sound:

```
tico.mc.mono <- tico.mc[,2]
nchannel(tico.mc.mono)
[1] 1
```

or any pairs of channels into a `WaveMC` object with two channels, that is, a stereo sound:

```
tico.mc.stereo <- tico.mc[,c(2,3)]
nchannel(tico.mc.stereo)
[1] 2
```

DIY 6.1 — How to apply mono conversion and to mix channels

It is possible to have access manually to the two channels of a stereo sound by extracting the slots of the `Wave` objects. The following code extracts and converts the left and right channels into `Wave` mono objects as does `mono()`:

```
tico.left <- Wave(tico.stereo@left,
                  samp.rate=tico.stereo@samp.rate,
                  bit=tico.stereo@bit)
tico.right <- Wave(tico.stereo@right,
                   samp.rate=tico.stereo@samp.rate,
                   bit=tico.stereo@bit)
```

We have also seen that `mono()` can average the left and right channels to produce a new mixed mono object. This can be achieved by associating the channel slots into a single `matrix` with `rbind()` and then by computing the column means with `colMeans()`:

```
tico.both <- colMeans(rbind(tico.stereo@left,
                            tico.stereo@right))
tico.both <- Wave(tico.both,
                  samp.rate=tico.stereo@samp.rate,
                  bit=tico.stereo@bit)
```

This opens the possibility to mix the channels with different weights. If we want to mix the left and right channels with twice more weight to the left channel than to the right channel, we just multiply the left channel by a factor of 2:

(continued)

DIY 6.1 (continued)

```
tico.mix <- colMeans(rbind(2*tico.stereo@left,
                           tico.stereo@right))
tico.mix <- Wave(tico.mix,
                 samp.rate=tico.stereo@samp.rate,
                 bit=tico.stereo@bit)
```

6.3 Manipulating Sound Sections

Deleting, moving, and repeating sentence sections are basic operations of digital writing. It is also often necessary to apply such actions with sound to change the order of sound events, to remove unwanted sounds, and to copy or to repeat a section of interest. Both `tuneR` and `seewave` provide solutions to manipulate sound sections.

6.3.1 Extract

The extraction, or cut, of a sound section consists in selecting a section of an input sound and storing it in a new object without changing the input sound. `extractWave()` of `tuneR` and `cutw()` of `seewave` process such a job in a rather similar way. Both functions require an input object, a start position (argument `from` in both functions), and an end position of the section to extract (argument `to` in both functions). The main difference lays in that `cutw()` can accept different object classes as input and can return the output section in different object classes. `extractWave()` has also an argument `xunit`, that `cutw` does not have, to specify the unit of time. This can be either `sample` for the sample position (not value) or `time` for seconds. To extract the second syllable of `tico` which is localized between 0.6 and 0.87 s, we can call either `extractWave()` or `cutw()` to reach the same result:

```
syllable2 <- extractWave(tico, from=0.6, to=0.87, xunit="time")
```

is equivalent to:

```
syllable2 <- cutw(tico, from=0.6, to=0.87, output="Wave")
syllable2

Wave Object
Number of Samples:      5955
Duration (seconds):     0.27
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

It is not mandatory to provide values for both `from` and `to` arguments. With `cutw()` if only one argument is provided, then the section extracted will either start at the beginning of the sound or stop at its end. The following code extracts the first and second syllables of `tico` that are included between 0 and 0.87 s.

```
sylabbles1and2 <- cutw(tico, to=0.87, output="Wave")
```

and the following code extracts the third and fourth syllables of `tico` that occur between 0.87 s and the end of the sound.

```
sylabbles3and4 <- cutw(tico, from=0.87, output="Wave")
```

If `from` and/or `to` of `extractWave()` are omitted, then the oscillogram of the input sound pops up in a new graphical device, and the console waits for clicking on it to choose the missing value(s). Actually, the argument `interact` of `extractWave()` and `choose` of `cutw()` allows to choose visually the limits of the section to be extracted. `cutw()` has an extra logical argument, `plot`, that plots the oscillograms of the input and output sounds.

The functions `extractWave()` and `cutw()` can be included in a loop so that different sections of a single input sound can be extracted sequentially. Such sound splitting requires some tricks that are detailed in the DIY box 6.2.

DIY 6.2 — How to split a sound into several sound bouts

The idea is to split `tico` into four `Wave` objects each containing one of the four syllables of `tico`. The `from` and `to` positions are first stored in a matrix named `positions` so that the first row of the matrix corresponds to the time start and end of the first syllable, the second row to the time start and end of the second syllable, and so on. Here the positions

(continued)

DIY 6.2 (continued)

were identified visually with the oscillogram, but solutions for time measurements are suggested in Chap. 8:

```
positions <- matrix(data=c(0, 0.6, 0.6, 0.87,
                           0.87, 1.42, 1.42, 1.79),
                    byrow=TRUE, ncol=2)
colnames(positions) <- c("from", "to")
positions
     from   to
[1,] 0.00 0.60
[2,] 0.60 0.87
[3,] 0.87 1.42
[4,] 1.42 1.79
```

The loop consists in using successively each row of the matrix `positions` to create a new object named `syllable1`, `syllable2`, and so on. There is however a trick to do so. It is not possible to write code on the left of the assign arrow (`<-`). For instance, using the function `paste()` to generate an object name does not work:

```
paste("object", 1, sep="") <- 1:10
```

The arrow `<-` can be replaced by the function `assign()` that takes two main arguments. The first argument `x` is the code that would be written on the left of the arrow, and the second argument `value` corresponds to the code on the right of the arrow. The above issue can be fixed with:

```
assign(paste("object", 1, sep=""), 1:10)
object1
 [1]  1  2  3  4  5  6  7  8  9 10
```

So the solution for splitting `tico` into four `Wave` objects is a `for` loop using `assign()`:

(continued)

DIY 6.2 (continued)

```
for(i in 1:nrow(positions))
    {
        assign(paste("syllable", i, sep = ""),
               cutw(tico, from = positions[i,1],
                          to = positions[i,2],
                          output = "Wave"))
    }
```

We can check the first object:

```
syllable1

Wave Object
Number of Samples:      13230
Duration (seconds):     0.6
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

6.3.2 Delete

Deleting is a kind of extraction process as a section of a sound is extracted and removed from the input sound. This can be achieved with the function `deletew()` of `seewave`. Removing the second syllable of `tico` is rather easy as the syntax is the same as for `cutw()`:

```
tico.nosyllable2 <- deletew(tico, from=0.6, to=0.87, output="Wave")
```

The duration of the output object is $1.79 - (0.87 - 0.6) = 1.52$ s:

```
duration(tico.nosyllable2)
[1] 1.524853
```

6.3.3 Paste

Both `tuneR` and `seewave` provide solutions to paste or concatenate sound objects. The `bind()` function of `tuneR` can concatenate `Wave` objects in a very simple way as the generic function `c()` does. The only requirement is that the objects should have the same properties, that is, the same sampling frequency (f_s), the same digitization depth (bit), and the same number of channels, something that can be checked with the function `checkwavs()` of the package `warbleR` (see Sect. 4.2.1). We can, for instance, build a new sound using the objects we have generated above by concatenating `tico`, the second syllable of `tico` which is stored in `syllable2` and `tico` without the second syllable which is saved in the object `tico.nosyllable2`. All these objects are `Wave` objects and have the same sampling features so that binding same together does not return any error:

```
b <- bind(tico, syllable2, tico.nosyllable2)
b

Wave Object
Number of Samples:      79156
Duration (seconds):     3.59
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Binding two sounds may generate clicks at the junction of the waves. These clicks are mainly due to a break in the phase of the waves. Figure 6.3 shows two waves out of phase, `wave1` and `wave2`, that generate a click when they are concatenated.

The function `prepComb()` of `tuneR` offers a solution to try to avoid such clicks. The function removes some amounts of the waves to be pasted, either at their start or end, or at both extremities. To remove the click at the junction of the `wave1` and `wave2`, it is necessary to apply `prepComb()` at the end of `wave1` and at the start of `wave2`, respectively:

```
wave1 <- prepComb(wave1, where="end")
wave2 <- prepComb(wave2, where="start")
wave <- bind(wave1, wave2)
```

Figure 6.4 clearly proves that the click disappeared.

However, if this procedure is very efficient, it also reduces the length of the concatenated objects. Here `wave1` and `wave2` were made of 300 samples each

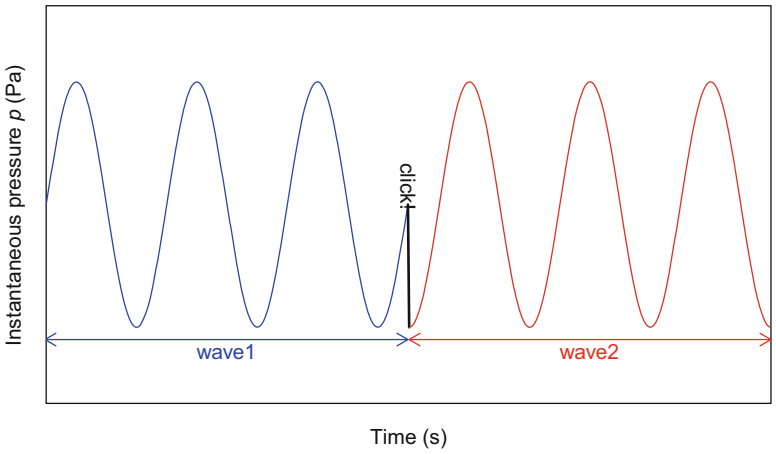


Fig. 6.3 Clicks when concatenating (pasting) waves. The concatenation of two waves with different phases might generate unwanted clicks. There is a $3\pi \div 2$ rad or 270° shift between the two waves

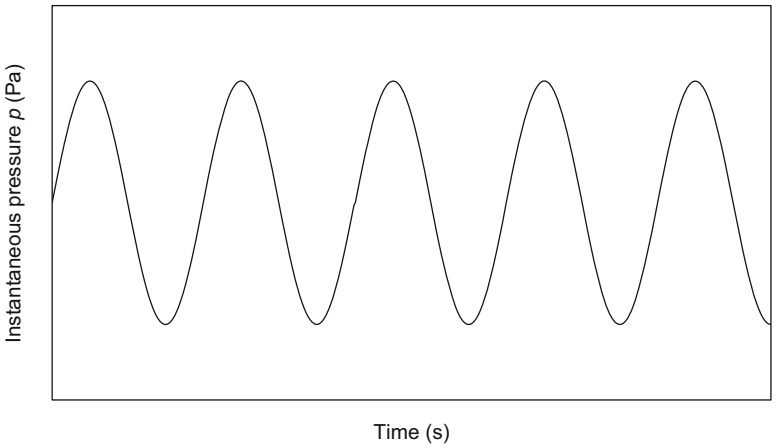


Fig. 6.4 Click removing by `prepComb()`. The click at the junction between `wave1` and `wave2` was removed thanks to the function `prepComb()` of the package `tuneR`

so that the concatenated objects should be made of 600 samples, but the use of the `prepComb()` truncated the length of the original objects to 200 and 275 so that the final object has a reduced length of 475:

```
length(wave1)
[1] 200
```

(continued)

```
length(wave2)
[1] 275
length(wave)
[1] 475
```

This may have important consequences as neither the sampling frequency nor the duration of the bound object is changed. This may induce artifacts in terms of frequency, here a negative frequency shift.

`seewave` also includes a function that concatenates or pastes sound. This function, `pastew()`, is more limited than `bind()` as it accepts only two objects to be pasted so that we need two lines of codes to do the same as `bind()`:

```
b <- pastew(syllable2, tico, output="Wave")
b <- pastew(tico.nosyllable2, b,output="Wave")
b

Wave Object
Number of Samples:      79156
Duration (seconds):     3.59
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Note that the default orders of the objects is reversed compared to `bind()` as the first argument of `pastew()` is the object to be pasted and the second argument is the object to be completed (input). However, the function `pastew()` offers the possibility to paste the second object anywhere in the first object with the argument `at`. By default `at` is set up to `"end"`. The default value can be changed to `"start"` so that the order of pasting is similar to `bind()`. The value of `at` can also be `"middle"` to paste the second object in the middle of the input object. The same argument can also accept a numeric value in s specifying where the object should be pasted. The result can be plotted with the argument `plot`. In the following example, the second syllable is pasted at the position 0.6 s so that the second syllable is repeated once and the result is displayed in the graphics device with `plot=TRUE` (Fig. 6.5):

```
pastew(syllable2, tico, at=0.6, output="Wave", plot=TRUE)
```

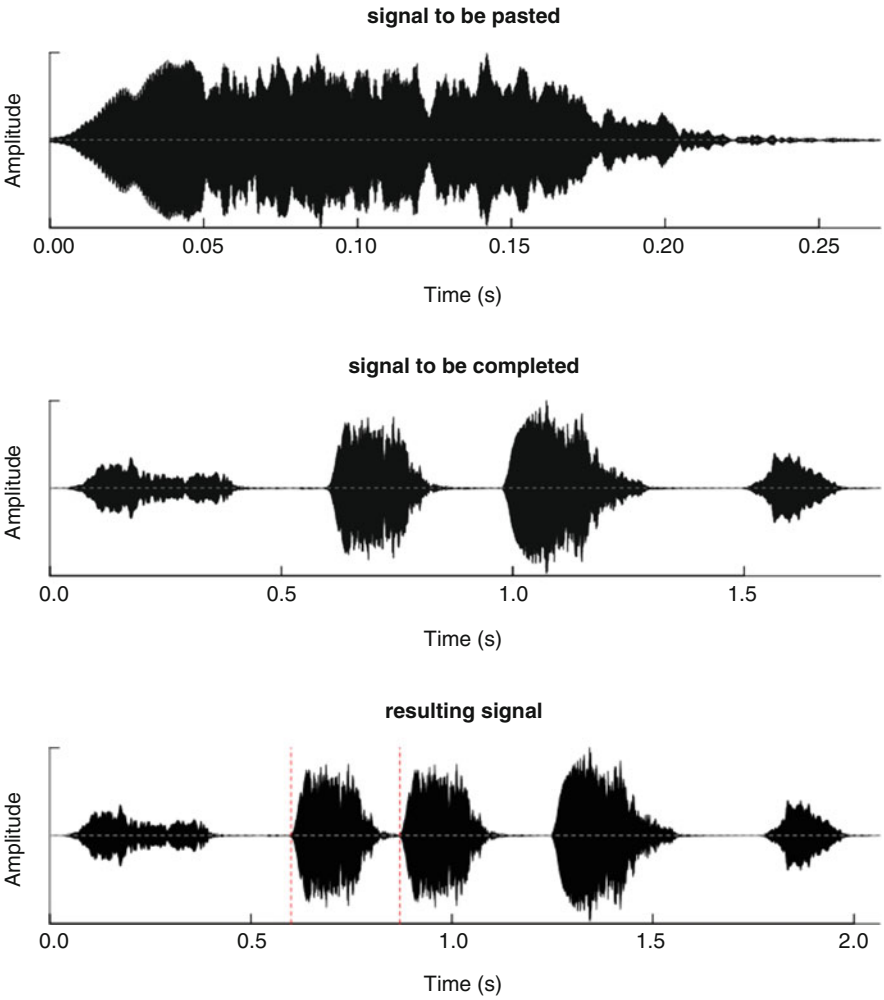


Fig. 6.5 Pasting sounds with `pastew()`. The second syllable is pasted (inserted) into `tico` at 0.6 sand the result is plotted

This results in inserting a sound into another one. The position of insertion can also be chosen interactively by turning the argument `choose` to `TRUE`:

```
pastew(syllable2, tico, choose=TRUE, output="Wave")
```

The management of clicks due to phase shifts can be managed with two arguments of `pastew()`. When set up to `TRUE`, the argument `join` removes the

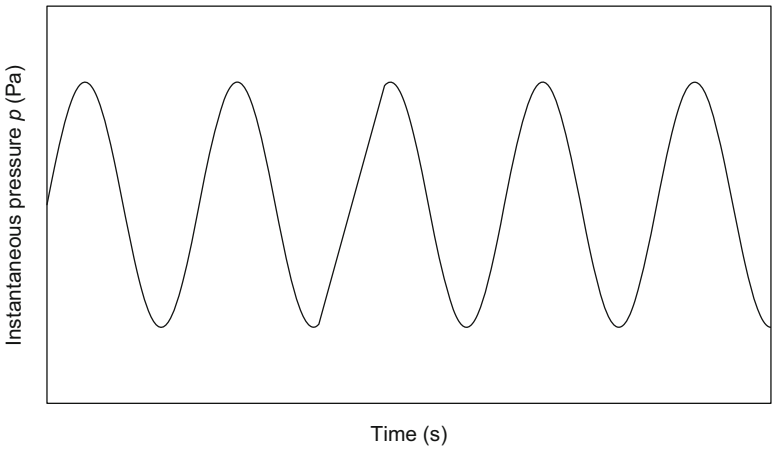


Fig. 6.6 Click removing by `pastew()`. The click at the junction between `wave1` and `wave2` was removed thanks to the function argument `tjunction` of `pastew()` of the package `seewave`

first sample of the object to be pasted. This can limit the occurrence of some clicks but this will not work in all cases. The argument `tjunction` does a bit more as it replaces original values at the junction of the two objects by new values generated through a linear interpolation. This procedure applies to a certain amount of time that is specified with a numeric value in s. For instance, setting `tjunction` to 0.001 will apply the linear interpolation on a 1 ms section. The result might not be always as good as the one obtained with `prepComb()`, but it has the great advantage to preserve the number of samples and therefore the main properties of the output object (Fig. 6.6):

```
wave <- pastew(wave2, wave1, tjunction=0.001, output="Wave")
length(wave)
[1] 600
```

Clicks can also be avoided by applying a fade effect as explained in Sect. 6.5.3.

6.3.4 Repeat

The function `repw()` of `seewave` is an iterative copy-and-paste action of the input object by itself. The main argument is `times` that sets up the number of times the object has to be repeated. The following code repeats `tico` three times:

```
tico3 <- repw(tico, times=3, output="Wave")
tico3

Wave Object
Number of Samples:      118734
Duration (seconds):     5.38
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

6.3.5 *Reverse*

The function `revw()` of `seewave` can "reverse" the sound so that the last sample of the original sound becomes the first sample of the modified sound:

```
tico4 <- revw(tico, output="Wave")
```

Note that both amplitude and frequency modulations are reversed with this action. See Sect. 15.5 to reverse independently the amplitude and frequency modulations.

6.4 Removing and Inserting Silence Sections

Sections of low amplitude or silence are sometimes unnecessary as they might contain no relevant information. The functions `noSilence()` of `tuneR` and `zapsilw()` of `seewave` both intend to remove quiet sound sections.

`noSilence()` cuts off silence that can occur at the start and/or end of a sound; in other words `noSilence()` is a trim function. It requires to set up an amplitude level under which samples will be removed. It is therefore necessary to get information on the distribution of the samples values. This can be obtained by drawing a histogram of the absolute value of the instantaneous amplitude a, i.e., a histogram of the absolute amplitude envelope (Fig. 6.7):

```
tico.h <- hist(abs(tico@left))  # histogram plotted and saved
first.cell <- tico.h$mids[1]    # center of histogram cell 1
abline(v=first.cell, col="red") # vertical line at cell 1 center
```

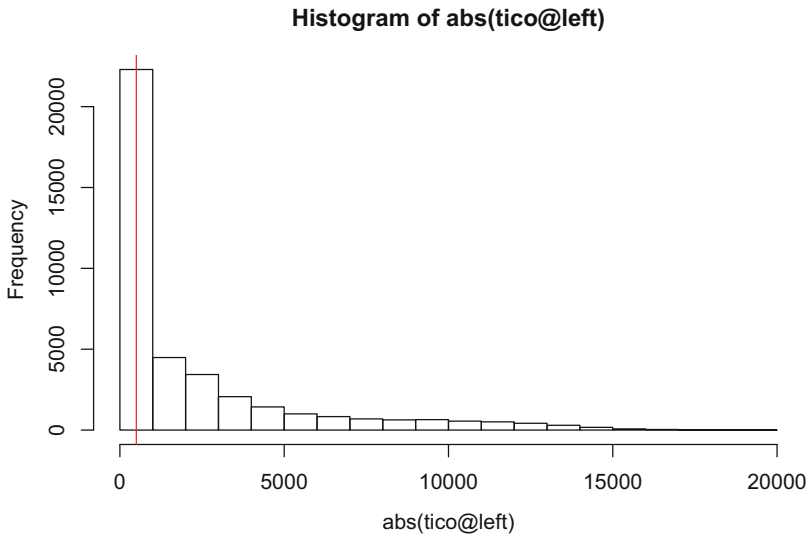


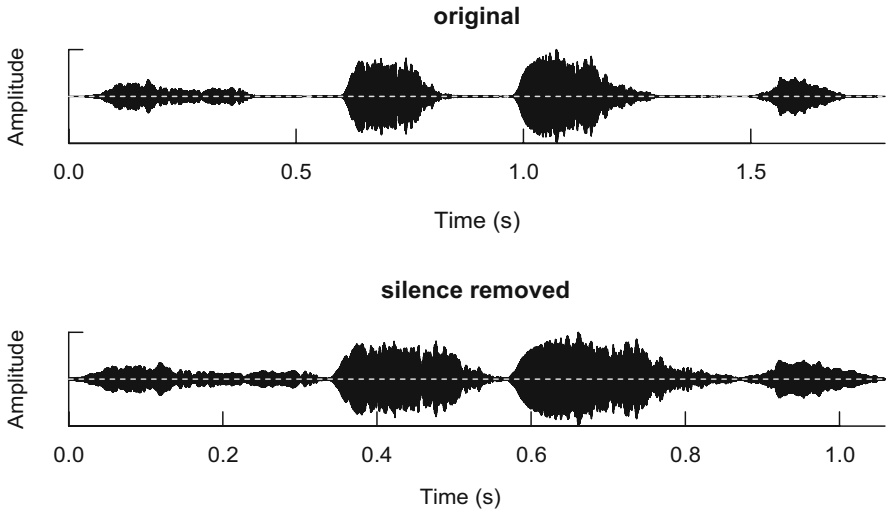
Fig. 6.7 Histogram of `tico` absolute amplitude envelope. Distribution of the absolute values (absolute amplitude envelope) of the `tico` samples. The first cell counts the numbers of samples between 0 and 1000, the vertical red bar indicates the center of the first cell at 500

The first cell is mainly due to low values varying between 0 and 1000, that is, to quiet sections of the sound. The middle of this cell (500) can be therefore used for setting the argument `level` of `noSilence()` (Fig. 6.8):

```
tico.nosilence <- noSilence(tico, level=first.cell/2)
```

The function `zapsilw()` does not do exactly the same as `noSilence()`: `zapsilw()` removes every samples that are below a defined amplitude threshold anywhere, not only at the start or at the end of the sound. The argument `threshold` of `zapsilw()` is homologous of the argument `level` of `noSilence()`, but it is defined along a relative scale in percentage. Setting a threshold of 5% means that values lower than 5 of the absolute amplitude envelope rescaled between 0 and 100 will be removed. Applying a threshold of 1% to `tico` removes the silent parts around the four syllables (Fig. 6.8):

```
tico.zapsilw <- zapsilw(tico, threshold=1, output="Wave")
```



Both operations change the duration or length of the input sound but other properties are not altered:

```
length(tico)
[1] 39578
length(tico.nosilence)
[1] 36861
length(tico.zapsilw)
[1] 23352
```

`seewave` includes two functions that operate an opposite action by inserting or adding silence sections. The function `mutew()` replace sampled values by 0 values between two time limits set with the arguments `from` and `to`. This can be used to replace a section by silence without changing the duration of the original sound. For instance, the following code replaces the second syllable of `tico` by silence (Fig. 6.9):

```
mutew(tico, from=0.6, to=0.87, output="Wave")
```

The function `addsilw()` can be used to insert silence bouts. The duration of the inserted silence bout is set in s with the argument `d`, and the position within the original sound is specified with the argument `at` in s as well. This latter argument is similar to the argument `at` of `pastew()`. It can be either a character string—(`"start"`, `"middle"`, or `"end"`)—or a time position expressed in s. It is often

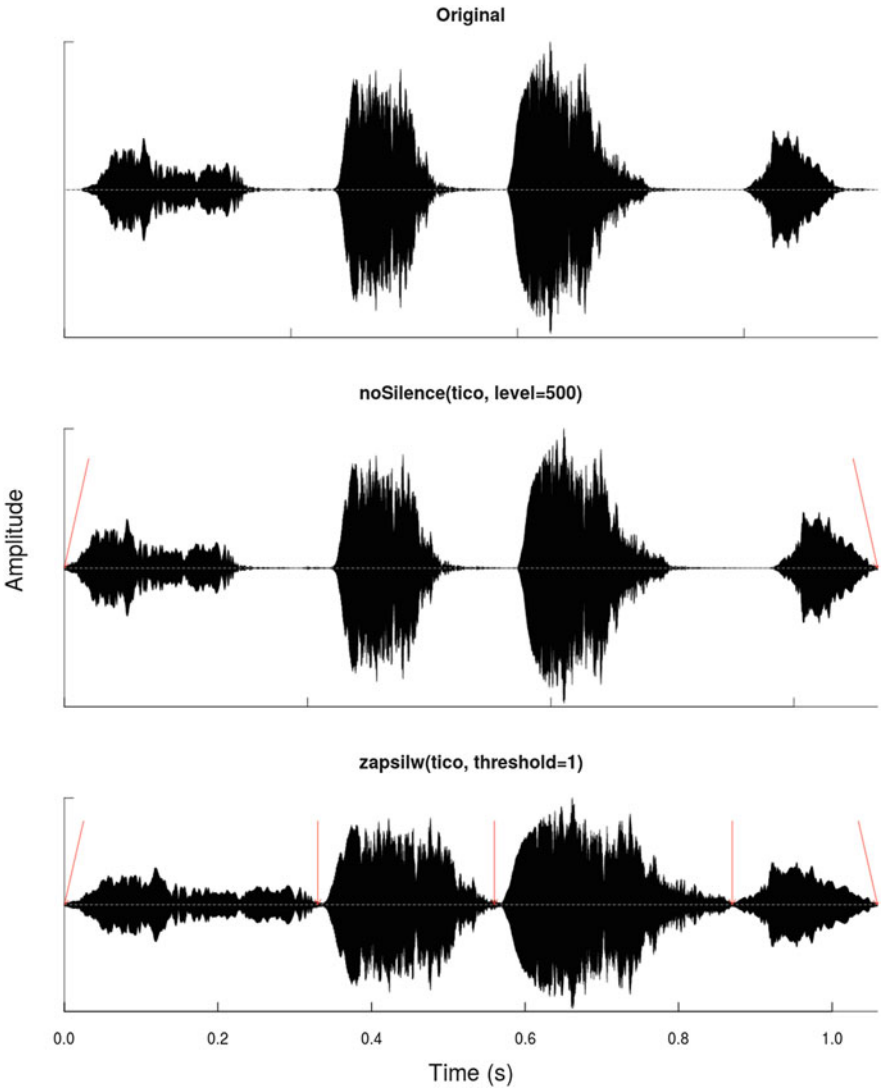


Fig. 6.8 Removing silence. The figure shows the results of both `noSilence()` and `zapsilw()` functions. The first function works at start and end of the signal operating as a trim function when the second function removes every silence sections. Sections modified are highlighted with red arrows drawn with `arrows()`

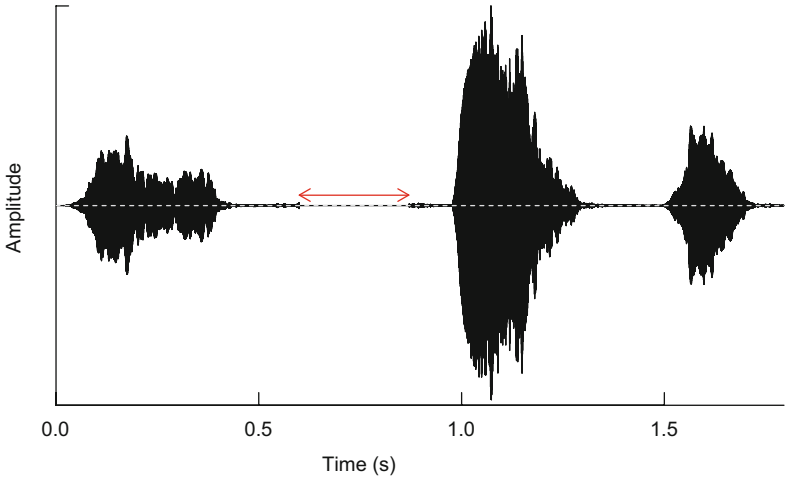


Fig. 6.9 Muting. The second syllable of `tico`, which starts at 0.6 sand stops at 0.87 s is muted by replacing original samples values with 0 values. The new silence section is highlighted with a red arrow drawn `arrows()`

useful to add some silence at both start and end of a sound to zoom out or to improve automatic temporal measurements (see Chap. 8). Such action can be operated with two lines of code (Fig. 6.10):

```
tico.start <- addsilw(tico, d=0.2,
                      at="start", output="Wave")
tico.start.end <- addsilw(tico.start, d=0.2,
                          at="end", output="Wave")
```

6.5 Changing Amplitude

6.5.1 Offset

An amplitude offset occurs when sample values do not equally vary around the reference value (for instance, p_0) but are all shifted toward low or high amplitude values (Fig. 6.11).

Such amplitude shift can be removed by a simple correction consisting in subtracting all samples values by either the mean, the median, or any other position statistical parameter of the wave. By default, the function `rmoffset()` of `seewave` uses the mean, but other function can be called with the argument

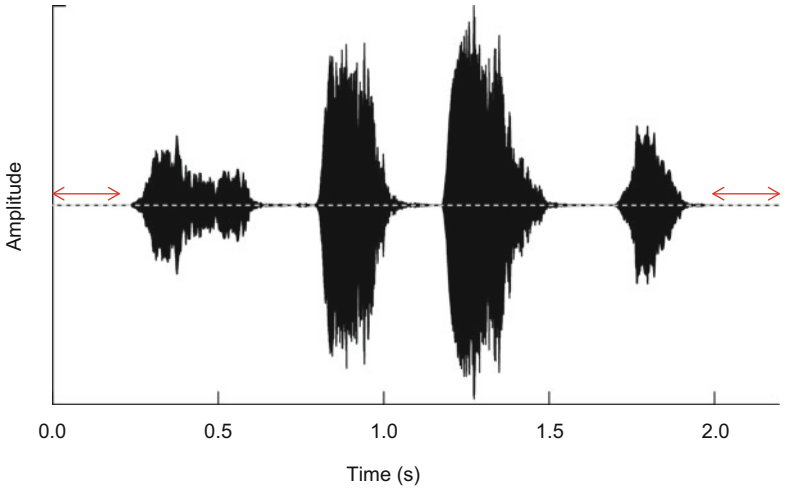


Fig. 6.10 Adding silence. Silence sections can be added with the function `addwilw()` as demonstrated here by adding 0.2 s bouts at both start and end of `tico`. The new silence sections are highlighted with red arrows drawn with `arrows()`

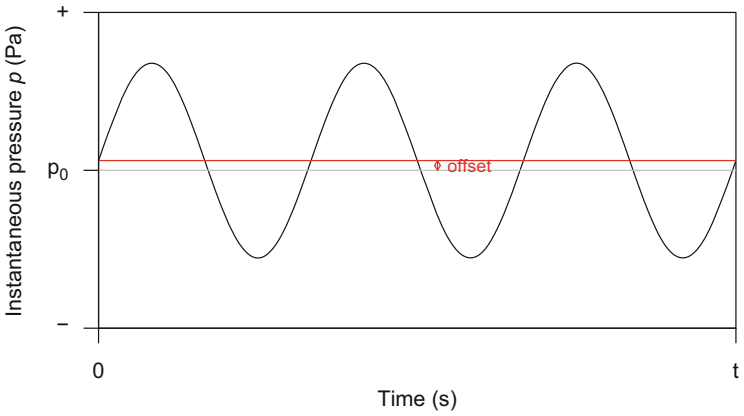


Fig. 6.11 Amplitude offset. This wave is shifted toward high amplitude values, departing from the p_0 reference value

`FUN`. For a hypothetical sound `wave` affected by an offset, the code to correct this amplitude shift would be:

```
nooffset.mean <- rmoffset(wave)    # offset remove with mean
nooffset.median <- rmoffset(wave,
                      FUN=median) # offset remove with median
```

6.5.2 Amplitude Level

There is no dedicated function to apply a change of the amplitude level. However, arithmetic operations work on the objects of class `Wave` such that a multiplication or a division can be directly applied. Here, two new objects are derived from `tico` through a division and a multiplication by 2 to produce a “quiet" and a “loud" object, respectively. The changes are checked by computing the root-mean-square (RMS) of the amplitude envelope with the function `rms()` (see Sect. 7.1):

```
rms(env(tico, plot=FALSE))       # rms of the original object
[1] 5554.346
tico.quiet <- tico/2
rms(env(tico.quiet, plot=FALSE)) # rms of the 'quiet' object
[1] 2777.173
tico.loud <- 2*tico
rms(env(tico.loud, plot=FALSE))  # rms of the 'loud' object
[1] 11108.69
```

These changes are not kept when saving the objects in a `.wav` file with the function `savewav()` because `savewav()` applies by default a 16 bit normalization. The trick is to call the argument `rescale` of `savewav()` for rescaling the amplitude between a lower and an upper limit. The argument `rescale` waits then a numeric vector of length 2. We can here use the range of the left channel to apply the rescaling:

```
savewav(tico.quiet, rescale=range(tico.quiet@left))
savewav(tico.loud, rescale=range(tico.loud@left))
```

The argument `rescale` can be actually used with any value such that any change of the amplitude level can be applied when exporting in `.wav`. Here is an example ranging the values of the output file along a 12 bit scale:

```
savewav(tico, file="tico-12bit.wav", rescale=c(-2^11, 2^11))
```

The function `normalize()` of `tuneR` can also proceed global amplitude changes. The function can first change the scale of the sample values. The argument `unit` can be used to rescale the samples values between $[-1, 1]$ or along a 8, 16, 32, or 64 bit scales.

`tico` is a 16-bit sound with sample values varying between $-18{,}596$ and 19,125:

```
summary(tico)

Wave Object
Number of Samples:      39578
Duration (seconds):     1.79
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16

Summary statistics for channel(s):

      Min.    1st Qu.     Median       Mean    3rd Qu.
-18596.000   -583.750      0.000      0.495    585.750
      Max.
 19125.000
```

The 16-bit scale can be switched to a 32-bit scale with `normalize()` such that sample values $a_i \in [-2^{32-1}, 2^{32-1}] = [-2.147 \cdot 10^9, 2.147 \cdot 10^9]$ by setting the argument `unit` to `"32"`, the value being quoted as `unit` waits for a character string and not a numeric vector:

```
tico.32 <- normalize(tico, unit="32")
summary(tico.32)

Wave Object
Number of Samples:      39578
Duration (seconds):     1.79
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    32

Summary statistics for channel(s):

      Min.    1st Qu.     Median       Mean    3rd Qu.
-2.088e+09 -6.560e+07 -5.554e+04  0.000e+00  6.572e+07
      Max.
 2.147e+09
```

It is important to note that this rescaling is somehow artificial: the quantization has changed but the accuracy of the sampling has not.

The argument `level` can be used to rescale the sample values between any limits. This argument is defined as the maximal percentage of the amplitude. Keeping with `tico.32`, setting `level` to 0.75 will rescale the values $a_i \in [-0.75 \times 2^{32-1}, 0.75 \times 2^{32-1}] = [-1.61 \cdot 10^9, 1.61 \cdot 10^9]$:

```
tico.32b <- normalize(tico, unit="32", level=0.75)
summary(tico.32b@left)
      Min.    1st Qu.     Median       Mean    3rd Qu.
-1.566e+09 -4.920e+07 -4.166e+04  0.000e+00  4.929e+07
      Max.
 1.611e+09
```

Therefore `normalize()` can be used to change the amplitude of a sound, and this modification can be saved in an external `.wav` file. Here `tico` is first normalized at 100% and saved in a file named `tico-loud.wav` and then normalized at 50% and written in another file named `tico-soft.wav`:

```
writeWave(normalize(tico, unit="16"),# maximal loudness
          filename="tico-loud.wav")
writeWave(normalize(tico, unit="16", # loudness divided by 2
           level=0.5),
           filename="tico-soft.wav")
```

In all cases, sample values are by default recentered around 0 by `normalize()` so that any offset is removed. Centering can be turned off by setting the argument `center` to `FALSE`.

6.5.3 *Fade-In and Fade-Out*

A usual edit action is to increase gradually the amplitude of the beginning of the recording and to decrease it in a similar way at the end of the recording. Such amplitude effect, known as fade-in and fade-out, respectively, is available in the `seewave` function `fadew()`. The argument `shape` gives the choice between three fade shapes, namely, a linear, an exponential, and a cosine shape, and the arguments `din` and `dout` specify the duration of the effect at the beginning and at the end of the sound, respectively. To illustrate this function, we will refer to the synthetic sound mimicking an A vibrating tuning fork, as already used in Sect. 4.2.1. The sound is enclosed in the file `tuning-fork.wav`:

```
tuningfork <- readWave("sample/tuning-fork.wav")
tuningfork

Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The following code exemplify three fade-in and fade-out examples with linear, exponential, and cosine attack and tail (Fig. 6.12):

```
layout(matrix(1:4, nrow=4))
par(mar=rep(2,4), oma=c(3,2,0,0))
din <- 0.1 ; dout=0.3
oscillo(tuningfork,
      alab="", tlab="", xaxt="n",
      title="Original")
fadew(tuningfork, din=din, dout=dout, shape="linear",
      alab="", tlab="", xaxt="n", plot=TRUE,
      title="Linear fade in/out with shape='linear'")
fadew(tuningfork, din=din, dout=dout, shape="exp",
      alab="", tlab="", xaxt="n", plot=TRUE,
      title="Exponential fade in/out with shape='exp'")
fadew(tuningfork, din=din, dout=dout, shape="cos",
      alab="", tlab="", plot=TRUE,
      title="Cosine fade in/out with shape='cos'")
mtext("Times (s)", side=1, line=1, outer=TRUE)
mtext("Amplitude", side=2, outer=TRUE)
```

The function `crossFade()` of the package `soundgen` can paste two sounds and apply at the same time a fade-out effect on the first sound and a fade-in effect at the start of the second sound so that that the transition between the two sounds is smoothed.

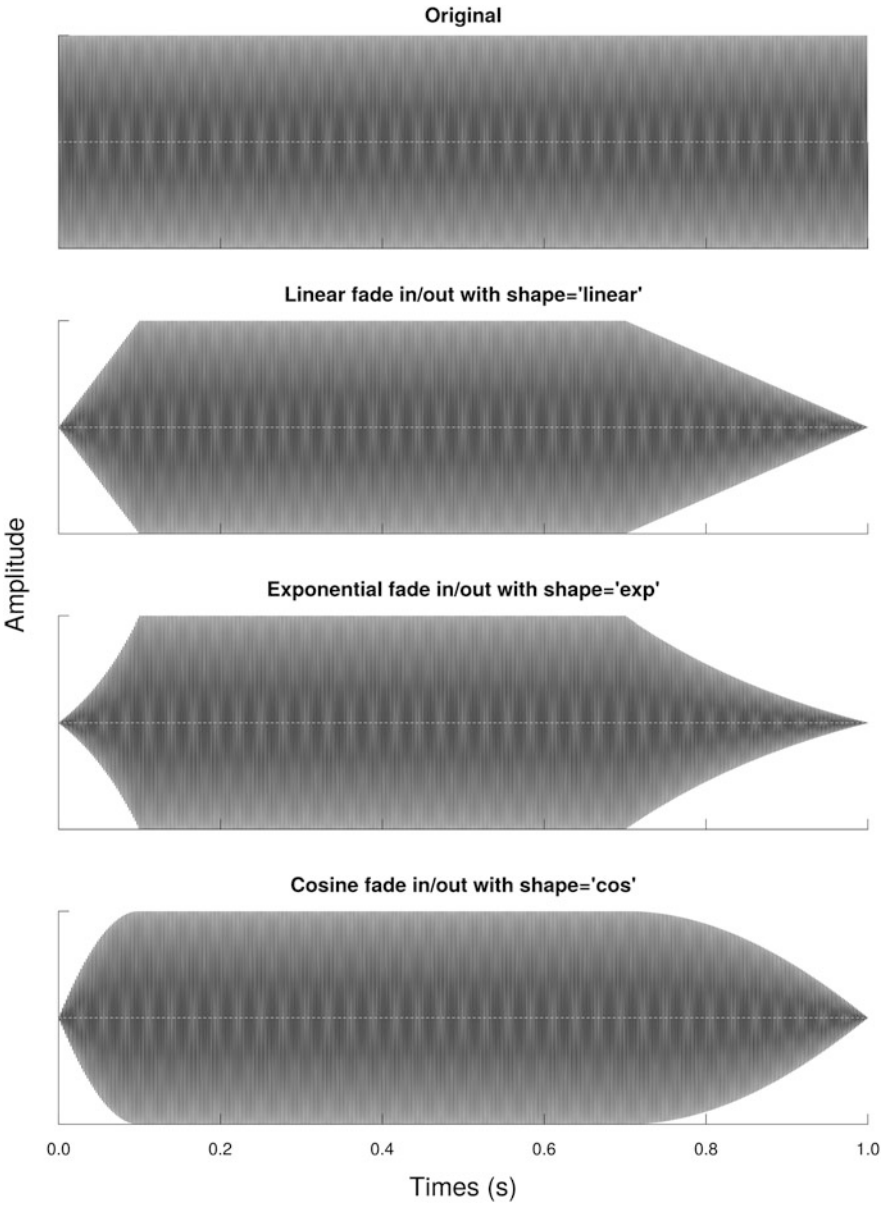


Fig. 6.12 Fade-in and fade-out. Fade-in and fade-out are applied to the tuning fork sound with three different amplitude shapes: linear, exponential and cosine

Chapter 7
Amplitude Parametrization

The amplitude is one of the first parameter that needs to be reported when describing a sound. It is important to know how much energy contains a sound and how the amplitude is distributed along time. As we have seen in Sect. 2.2.3, amplitude can be measured along a linear scale or along the logarithmic dB scale that was designed to comply with main human audition properties. Amplitude can also be expressed along a relative or absolute scale, the latter requiring a calibration of the full recording chain. We will first see what `R` can do in terms of relative linear and dB measurements and then in terms of absolute measurements.

7.1 Linear Relative Scale

The amplitude of a wave can be characterized by (1) the instantaneous amplitude $a(t)$, (2) the maximum amplitude $A = \max a(t)$, (3) the peak-to-peak amplitude $pk\text{-}pk(a(t))$, or (4) the root-mean-square amplitude $\mathtt{RMS}(a(t))$ (Fig. 2.3).

The **instantaneous** amplitude is the value of any sample along the digitization scale. A sample value can be returned by indexing the raw data. For instance, the value of the 1234th sample of `tico` is obtained by using the `[]` index syntax on the `S4` slot `@left`:

```
tico@left[1234]
[1] 412
```

The value returned will of course depends on the scale used. Here `tico` has been digitized on a 16 bit scale, but sample values could have been scaled within $[-1, +1]$ and $[-100, +100]$ or within any other limits. It is therefore crucial to

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_7

check the range of values with the function `range()` before interpreting or using sample values:

```
range(tico@left)
[1] -18596  19125
```

The **maximum** amplitude can be directly obtained with the function `max()`:

```
max(tico@left)
[1] 19125
```

However, this action returns a single value when a wave can reach a maximum several times. The `seewave` function `crest()` provides the values and the positions of the wave maxima also named crests. This function computes as well the crest factor according to:

$$\text{crest}(a(t)) = \frac{\max|a(t)|}{\text{RMS}(a(t))}$$

where RMS is the root-mean-square as defined below. The highest is the crest factor, the more pronounced is the amplitude peak around the maximum values. For `tico`, `crest()` returns:

```
crest(tico)
$C
[1] 4.86949

$val
[1] 19125

$loc
[1] 1.072653
```

The item `$C` is the crest factor, the item `$val` is the sample value, and `$loc` is the position of the maximum expressed in s.

The **peak-to-peak** amplitude is the difference between the maximum and the minimum and is obtained with:

```
max(tico@left) - min(tico@left)
[1] 37721
```

The **root-mean-square** RMS derives from the computation of the energy E and the average power $\bar{P}$. The energy is simply the sum of the squared sample values:

$$E = \sum_{i=1}^{n} x_i^2$$

obtained with:

```
E <- sum(tico@left^2)
E
[1] 610505784856
```

The average power is the energy per unit of time, here the number of samples n:

$$\bar{P} = \frac{E}{n} = \frac{1}{n} \sum_{i=1}^{n} x_i^2$$

so that $\bar{P}$ of `tico` is:

```
n <- length(tico@left)
P <- E/n
P
[1] 15425382
```

Eventually, the root-mean-square is the square-root of the average power $\bar{P}$:

$$\text{RMS} = \sqrt{\bar{P}} = \sqrt{\frac{E}{n}} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}$$

So that in `R` the root-mean-square is returned with:

```
RMS <- sqrt(P)
RMS
[1] 3927.516
```

or by calling directly the function `rms()` of `seewave`:

```
RMS <- rms(tico@left)
RMS
[1] 3927.516
```

So to summarize we have for `tico`:

```
data.frame(E,P,RMS)
              E        P       RMS
1 610505784856 15425382 3927.516
```

Several other parameters can be computed on the wave as the classical statistical measurements of position and dispersion:

```
min(tico@left)    # minimum
[1] -18596
max(tico@left)    # maximum
[1] 19125
mean(tico@left)   # mean
[1] 0.4946182
var(tico@left)    # variance
[1] 15425772
sd(tico@left)     # standard deviation
[1] 3927.566
median(tico@left) # median
[1] 0
mad(tico@left)    # median absolute deviation
[1] 865.8384
```

Some of these measurements can be returned at once with `summary()`:

```
summary(tico@left)
      Min.    1st Qu.     Median       Mean    3rd Qu.
-18596.000   -583.750      0.000      0.495    585.750
      Max.
 19125.000
```

Higher-order measurements can be computed to provide information on the data shape. For instance, a moment of order k is defined in the following:

$$\overline{\mu_k} = \frac{1}{n}\sum_{i=1}^{n} x_i^k$$

such that the first moment is the arithmetic mean and the second moment is the average power $\bar{P}$. We can write a function to compute the kth moment, but, as it often happens with `R`, there is almost always someone from the `R` community who thought about it before and shared a useful function. This is here the case with the function `moment()` of the package `moments`:

```
library(moments)
moment(tico@left, order=1) # 1st moment, or mean
[1] 0.4946182
moment(tico@left, order=2) # 2nd moment, or average power
[1] 15425382
moment(tico@left, order=3) # 3rd moment
[1] -36610586
moment(tico@left, order=4) # 4th moment
[1] 1.675247e+15
```

Similarly, the kth central moment is written as:

$$\mu_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^k$$

Such that the central moment of order 2 is the uncorrected variance:

```
moment(tico@left, order=2, central=TRUE)
[1] 15425382
```

which is equivalent to:

```
(n-1)/(n)*var(tico@left)
[1] 15425382
```

The third and fourth normalized central moments are called the skewness and kurtosis. Skewness is a measure of symmetry, while kurtosis is a way to estimate

the degree of “peakedness" of a statistical distribution. Skewness and kurtosis can be therefore used to quantify the global shape of the amplitude envelope.

Skewness is written according to:

$$\gamma_1 = \frac{\mu_3}{\mu_2^{3/2}}$$

where μ_2 and μ_3 are, respectively, the second and the third central moment. A negative (resp. positive) skewness indicates that the distribution is skewed to the left (resp. right). The skewness can be computed with the eponymous function of the package `moments`:

```
skewness(tico@left)
[1] -0.0009821102
```

which is equivalent to:

```
moment(tico@left, order=3, central=TRUE)/
    moment(tico@left, order=2)^(3/2)
[1] -0.0009821102
```

The kurtosis is written according to:

$$\beta_2 = \frac{\mu_4}{\mu_2^2}$$

where μ_2 and μ_4 are, respectively, the second and the fourth central moment. If $\beta_2 < 3$, the distribution is said platikurtic, i.e., it has fewer items at the center and at the tails but has more items in the shoulders; if $\beta_2 = 3$, the distribution shows a normal shape; and when $\beta_2 > 3$, the distribution is qualified as leptokurtic, having more items near the center and at the tails, with fewer items in the shoulders. The function `kurtosis()` of the package `moments` computes β_2:

```
kurtosis(tico@left)
[1] 7.040555
```

which is equivalent to:

```
moment(tico@left, order=4, central=TRUE)/
    moment(tico@left, order=2)^2
[1] 7.040555
```

All the amplitude measures that we have seen above were applied on the wave through the call of the raw data `tico@left`, but the same computations can also be applied to the Hilbert envelope, like:

```
summary(env(tico, plot=FALSE))
       V1
 Min.   :    0.006
 1st Qu.:   68.788
 Median : 1548.144
 Mean   : 3391.518
 3rd Qu.: 4850.072
 Max.   :19254.481
```

We can refer to the same functions to characterize a frequency spectrum (see Sect. 10.1.6.4).

7.2 Logarithm Relative Scale

7.2.1 Signal-to-Noise Ratio

The dB unit can be used to compute the signal-to-noise ratio, or SNR, which quantifies the relative importance of a signal over the background noise. The SNR formula consists in computing the ratio of the signal amplitude over the background noise amplitude. These amplitudes can be estimated with the RMS, and the ratio can be expressed in dB, following:

$$\text{SNR} = 20 \times \log_{10}\left(\frac{\text{RMS}_{\text{signal}}}{\text{RMS}_{\text{noise}}}\right)$$

If we had to estimate the SNR of the second syllable of `tico`, we should estimate the RMS of this syllable and the RMS of the background noise for a similar duration. To do this, we first cut the second syllable with `cutw()`, cut a section of background noise following the second syllable with a similar duration,

and compute the SNR:

```
signal <- cutw(tico, from=0.62, to=0.80)
length(signal)
[1] 3970
noise <- cutw(tico, from=0.82, to=1)
length(noise)
[1] 3970
SNR <- 20*log10(rms(signal)/rms(noise))
SNR
[1] 15.93484
```

If the noise does not stop when the signal is produced, another expression of SNR could be:

$$\text{SNR} = 20 \times \log_{10}\left(\frac{|\text{RMS}_{\text{signal}} - \text{RMS}_{\text{noise}}|}{\text{RMS}_{\text{noise}}}\right)$$

which is translated in R language in:

```
SNR <- 20*log10(abs(rms(signal)-rms(noise))/rms(noise))
SNR
[1] 14.42371
```

7.2.2 dB *Weightings*

We have seen in Sect. 2.2.3 the occurrence of dB weightings to take into account the nonlinearity in frequency sensitivity of the human ear. The conversion from dB SPL to weighted dB is ensured by the function `dBweight()` that waits a frequency in Hz as a first argument, here 440 Hz:

```
dBweight(440)
$A
[1] -4.095091

$B
[1] -0.3901849

$C
[1] 0.02969125
```

(continued)

```
$D
[1] -0.288377
```

The function returns by default the dB(A), dB(B), dB(C), and dB(D) values for a sound produced at 0 dB. It therefore provides the values for corrections. The corrections can be applied directly on any dB measure by using the argument `dB`, as exemplified here with an 80 dB sound:

```
dBweight(440, dB=80)
$A
[1] 75.90491

$B
[1] 79.60982

$C
[1] 80.02969

$D
[1] 79.71162
```

A quick way to reconstruct the dB weighting curves is to generate a vector of frequencies and to plot the result of `dBweight()` with the function `matplot()` that plots the column of a `matrix` or a `data.frame` (Fig. 2.5):

```
freq <- seq(10, 20000, by=10)
head(freq)
[1] 10 20 30 40 50 60
tail(freq)
[1] 19950 19960 19970 19980 19990 20000
res <- dBweight(freq)
matplot(x=freq, y=as.data.frame(res),
       xlim=c(10,21000), ylim=c(-80,20),
        type="l", lty=1, log="x",
        xlab="Frequency (Hz)", ylab="dB")
```

7.2.3 dB *Arithmetic*

dB is a logarithm scale unit that should be manipulated carefully when running arithmetic operations. Logarithms have their own rules ($\log(a \times b) = \log(a) +$

$\log(b)$, $\log(a \div b) = \log(a) - \log(b)$, $\log(a^b) = b \times \log(a)$) that prevent the use of basic arithmetic functions such as `sum()`, `mean()`, or `sd()`. For instance, the sum of 60 and 70 dB is not 130 dB but ≈70.41 dB. `seewave` offers three functions to deal with dB operations: `moredB()` for the sum, `meandB()` for the average, and `sddB()` for the standard deviation:

```
data <- seq(50,100, by=10)
data
[1]  50  60  70  80  90 100
moredB(data)
[1] 100.4576
meandB(data)
[1] 92.67606
sddB(data)
[1] 9.406182
```

These functions mainly convert the dB values to pressure values along the linear Pa scale, run the linear arithmetic operation, and apply the conversion back to dB. The conversion of dB into linear units can be processed by the function `convSPL()` that ensures the switch from dB to power P in W, intensity I in W m^{-2}, and pressure p in Pa. By default the function refers to airborne sound. Here are the results for a 80 dB sound:

```
convSPL(80)
$P
[1] 0.001256637

$I
[1] 1e-04

$p
[1] 0.2
```

The reference intensity I_{ref} and the reference pressure p_{ref} can be changed to fit, for instance, with water references:

```
convSPL(80, Iref=6.7*10^-19, pref=10^-6)
$P
[1] 8.419468e-10

$I
[1] 6.7e-11
```

(continued)

```
$p
[1] 0.01
```

The function `convSPL()` also includes an argument `d` expressed in m to change the distance from the source that is used in the computation of the power P:

```
convSPL(80, d=2)
$P
[1] 0.005026548

$I
[1] 1e-04

$p
[1] 0.2
```

7.2.4 Sound Attenuation Through Spreading Losses

The distance from the source is an important factor of sound attenuation. As detailed in Sect. 2.2.3, spreading losses are the part of sound attenuation due to the dispersion of the energy. This decrease of amplitude can be modelized for a spherical point source with the following equation:

$$l = l_{ref} - 20 \times \log_{10}\left(\frac{d}{d_{ref}}\right)$$

where l is a either the sound intensity level (SIL) or the sound pressure level (SPL), d the distance from the source, and d_{ref} the distance from the source where an amplitude level, l_{ref}, has been measured. The function `attenuation()` is an implementation of this equation to get the amplitude level in dB at a distance d. The following code plots 200 points (argument `n`) spreading losses curve starting at 80 dB (argument `lref`) measured at 1 m (argument `dref`) up to a distance of 150 m (argument `dstop`) (Fig. 7.1):

```
attenuation(lref=80, dref=1, dstop=150, n=200)
```

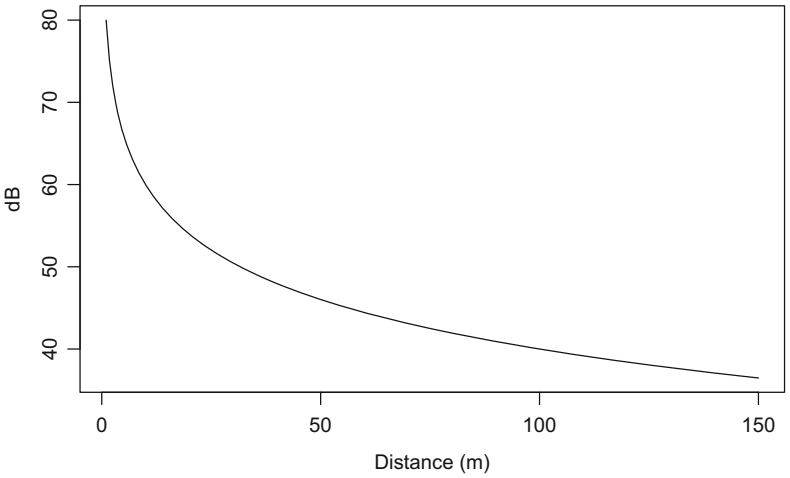


Fig. 7.1 Attenuation due to spreading losses. The curve of attenuation due to spreading losses for a sound source of 80 dB measured at 1 m is shown up to 150 m. This curve was obtained using the function `attenuation()`

The function `attenuation()` is not only a plot function; it can be used to know a specific dB value at a specific distance. The trick is to turn off the graphical display (`plot=FALSE`), to set the number of points to 2 (`n=2`), and to use the indexing syntax `([])` to extract the right value from the result vector:

```
attenuation(lref=80, dref=1, dstop=16, n=2, plot=FALSE)[2]
[1] 55.9176
```

It is of course possible to get the values for a series of distances. It is often advised when running propagation experiments to choose powers of 2 distances following:

```
d <- 2^(1:8)
d
[1]   2   4   8  16  32  64 128 256
```

What would be then the dB values expected at the distances `d` due to spreading losses only for a sound played back at 80 dB at 1 m? The function `attenuation()` cannot answer directly; it is necessary to include the precedent code for a single value into a loop. A `for` loop could be written, but the functions

of the family `apply` can do the job in a condensed form:

```
res <- sapply(X=d, FUN=function(x)
    attenuation(lref=80, dstop=x, plot=FALSE, n=2)[2])
res
[1] 73.9794 67.9588 61.9382 55.9176 49.8970 43.8764 37.8558
[8] 31.8352
```

The vector d consists of distances that are doubled at each step. We know that doubling the distance result in decreasing the amplitude by a factor of ≈6 dB. The difference between successive cells of the vector `res` should then equal to ≈6 dB. We can check this with the base function `diff()` that computes lagged differences or the first derivative:

```
diff(res)
[1] -6.0206 -6.0206 -6.0206 -6.0206 -6.0206 -6.0206 -6.0206
```

Another question could be to find the distance of propagation knowing an amplitude level in dB. For instance, one would like to know at which distance a sound produced at 80 dB at 1 m reaches 63 dB. There is no such function, but writing it should not be a problem as shown in the box DIY 7.1.

DIY 7.1 — How to estimate a distance of attenuation

To estimate the distance of propagation for a specific dB level, we first need to write the function of attenuation in respect with l rather than in respect with d. Therefore:

$$l = l_{ref} - 20 \times \log_{10}\left(\frac{d}{d_{ref}}\right)$$

becomes

$$d = d_{ref} \times 10^{\frac{l_{ref}-l}{20}}$$

We can then translate this equation into `R` language and design a new function named `distance()`:

```
distance <- function(dref=1, lref, l){
    dref*10^((lref-l)/20)
}
```

(continued)

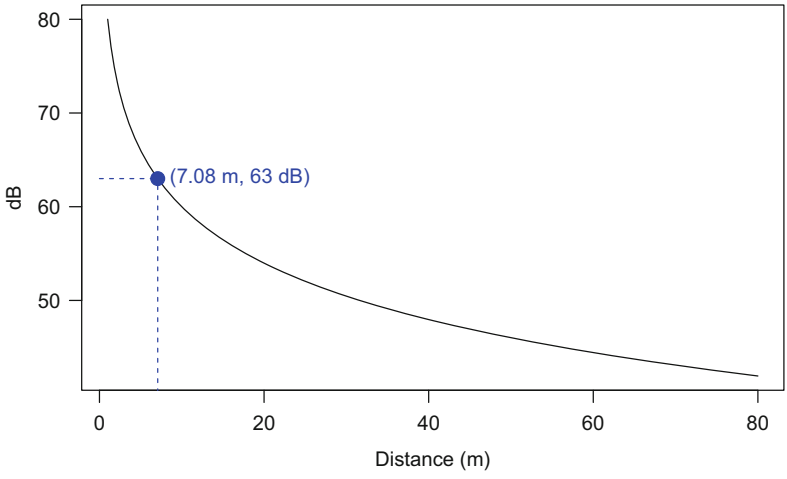
DIY 7.1 (continued)

What would be then the distance attained by a sound emanating from a point source recorded at 80 dB at 1 m? The new function `distance()` answers:

```
d63 <- distance(lref=80, d=1, l=63)
d63
[1] 7.079458
```

We can check if the result is correct by plotting the attenuation curve and adding the values as a singular point:

```
par(las=1)                          # horizontal axis labels
col <- "blue"                       # color
attenuation(lref=80, dref=1,        # attenuation curve
            dstop=80, n=200)
points(x=d63, y=63, pch=19,         # add point
       col=col,cex=1.5)
segments(x0=0, x1=d63, y0=63, y1=63,# add point coordinates
         lty=2, col=col)
segments(x0=d63, x1=d63, y0=0, y1=63,
         lty=2, col=col)
text(x=d63, y=63,                   # add text close to point
     labels=paste("(",round(d63,2) ," m, 63 dB)", sep=""),
     pos=4, col=col)
```



7.3 Absolute Scale

Most sound analyses involve amplitude data that vary along a relative scale, that is along a scale with units but without absolute reference. The amplitude envelope can for instance be limited between ± 1 or, if digitized with a 16 bit depth, between $\pm 2^{16-1}$. The information regarding absolute sound pressure or intensity level are therefore lacking limiting the description of the sound sources and the interpretation of the observations. Absolute measurements of sound level demand to use a fully calibrated recording chain. The recording chain, schematized in Fig. 7.2, involves having information on the sensitivity of the sensor (microphone, hydrophone, accelerometer, or any other transducer), the gain of the preamplifier, the quantization and filter properties of the analogue-digital converter, and finally the conversion into a digital file (`.wav` or other).

A calibration of the recording chain can be achieved with the help of a calibrator that produces a sound at a known frequency and sound pressure level (usually a 1 kHz pure tone produced at 94 dB SPL at source). The sound emitted by the calibrator is recorded and generates a reference digital file. The RMS of the absolute envelope of the reference file can be then used as a reference to estimate the absolute SPL of files obtained during recording sessions. This process assumes that the frequency responses of all the items of the chain are all linear. Another way to obtain calibrated data is to have access to the frequency-dependent properties of the successive stages of the recording chain, that is, the sensor sensitivity, the preamplifier gain, and the digitizer voltage, or to use a calibrator at several different frequencies.

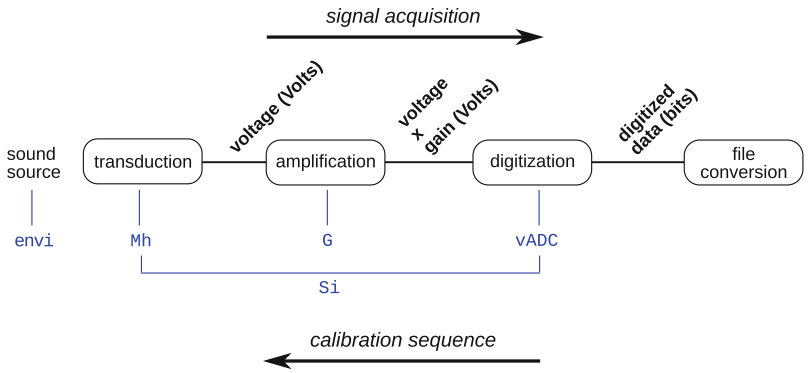


Fig. 7.2 Signal path and calibration sequence. The recording chain goes through several stages from the initial sound source to the terminal digital file passing through processes of transduction (microphone, hydrophone, accelerometer, or other), amplification (pre-amplifier), digitization (analogue-digital converter), and file conversion (computer algorithm). The arguments of the function `PAMGuide()` are indicated below the process they are related to. The argument `Si` covers the chain from transduction to digitization. Modified from Merchant et al. (2015)

The PAMGuide R code provides functions to calibrate sound data. PAMGuide source and complete documentation can be downloaded as an appendix of Merchant et al. (2015) or on sourceforge.[1]

As PAMGuide is not an R package but a script saved into an .r file, we load *PAMGuide* with the function source() that reads at once an R script, here saved in a subdirectory named pamguide:

```
source("pamguide/PAMGuide.R")
```

We then call the eponymous function PAMGuide() to get calibrated information on a sound. The sound to be treated is not an R object but an external .wav file that has to be localized interactively excluding the possibility to include PAMGuide() in scripts. The function has several arguments to specify where and how the recording chain was calibrated and to run as well some spectral analysis (see Chaps. 10 and 11).

To get calibrated data, it is first necessary to set the argument calib to 1. Then, the transmission medium should be specified with the argument envi that has two options, either "Air" for airborne sound or "Wat" for waterborne sound. The next argument, ctype, is the type of calibration with three possibilities: (1) "EE" to use the end-to-end sensitivity of the entire recording system, (2) "RC" to use the sensitivity of the microphone/hydrophone and the sensitivity of the recording device separately, and (3) "TS" to use the technical specifications of the recorder and transducer as provided by the manufacturer.

There are then four arguments to provide the sensitivity of the entire device (argument Si), the sensitivity of the sensor (argument Mh), the gain of the preamplifier (argument G), and the ADC zero-to-peak voltage (argument vADC). If the sensitivity of the sensor is known in mV Pa^{-1}, it can be converted into dB with the seewave function micsens(). For instance, if the sensitivity of a microphone is 10 mV Pa^{-1} with a reference of 1 V Pa^{-1} at 1 kHz, then the sensitivity in dB *re* 1 V Pa^{-1} will be:

```
micsens(10)
[1] -40
```

The arguments that need to be set according to the ctype argument are detailed in Table 7.1 and indicated on the calibration sequence of Fig. 7.2.

The output of PAMGuide() depends on the type of frequency analysis chosen with the argument atype. To get SPL values of the file, it is necessary to set

[1] http://sourceforge.net/projects/pamguide/.

Table 7.1 Main calibration arguments of `PAMGuide()` function

ctype	Arguments to set
"EE"	Si
"RC"	Mh, G
"TS"	Mh, G, vADC

`atype="Broadband"`. By default, the function plots results as time or frequency plots. This graphical can be turned off with `plottype="None"`.

The following call of `PAMGuide()` specifies that the recording was achieved in the air that we refer to the manufacturer calibration specifications with a microphone sensitivity of −36 dB, a preamplifier gain set by the user to +24 dB and zero-to-peak voltage of 1.414. These parameters correspond actually to the specifications of a Wildlife Acoutics© SongMeter SM2+ recorder with built-in microphones:

```
PAMGuide(calib = 1,          # get calibrated data
         atype="Broadband", # type of analysis
         plottype="None",   # no plot
         envi = "Air",      # air medium
         ctype = "TS",      # manufacturer specifications
         Mh = -36,          # microphone sensitivity
         G = 24,            # pre-amplifier gain
         vADC = 1.414,      # ADC voltage parameter
         )
```

The code would be as follows for the SongMeter SM2+ associated to a HTI-96© hydrophone with a gain of +12 dB:

```
PAMGuide(calib = 1,          # get calibrated data
         atype="Broadband", # broadband analysis
         plottype="None",   # no plot
         envi = "Wat",      # water medium
         ctype = "TS",      # manufacturer specifications
         Mh = -165,         # HTI-96 hydrophone sensitivity
         G = 12,            # pre-amplifier gain
         vADC = 1.414       # ADC voltage parameter
        )
```

The results of the function `PAMGuide()` are not returned as `R` objects but are directly printed in the console.

Chapter 8
Time-Amplitude Parametrization

Now that we know how much relative or absolute energy is produced by a sound, we need to get into other details. The first challenge is to estimate the duration of a sound. An endless sound does not exist; there is always a time when a sound starts and another time when it stops. Basic sound description includes the manual or automatic identification of these particular time boundaries that frame out a sound event as described in Sects. 8.2 and 8.3, respectively. The last Sect. 8.4 deals with the estimation of the regular amplitude modulations (AMs) that may occur in a sound.

8.1 What and How to Measure?

Measuring the duration of a signal consists in identifying, either manually or automatically, the start, or attack, and the end, or tail, of a sound. This identification should lead to the delimitation of sound events or **signal** sections (a note, a syllable, a pulse, etc.) and, indirectly, in delimiting the intervals between these events or **pause** sections (an inter-note interval, an inter-syllable interval, an inter-pulse interval, etc.). The sum of a signal section and its following pause section constitutes a **period** (a note period, a syllable period, a pulse period, etc.), and the number of events produced per s is expressed as a **rate** (a note rate, a syllable rate, a pulse rate, etc.). Duration is expressed in s and rate in s^{-1}.

Taking time measurements is more difficult than one believes. There are several factors that make the task demanding: (1) the sound can be of poor quality due to low-quality recording, (2) the occurrence of interfering sounds can blur the start and end of the sound, (3) a complex medium of propagation with several objects can induce echoes and artificially extends sound duration, (4) the number of sounds to treat can be so important that manual measurements are impossible and automatic measurements cannot be checked, and (5) the architecture of the sound can be

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_8

intricate with nested items such as phrases, words, syllables, and phonemes for speech or songs, bouts, phrases, syllables, notes, and pulses for animal vocalizations.

8.2 Manual Measurements

To introduce manual time measurements, we will first refer to the calling song of the Mediterranean cicada *Cicada orni* (Fig. 8.1). This sound, which constitutes the soundmark of the Mediterranean landscape, is a regular repetition of items, also named echemes by experts in insect acoustics. A sample of this cicada sound comes with `seewave` as `R` data under the name `orni`:

```
data(orni)
orni

Wave Object
Number of Samples:      15842
Duration (seconds):     0.72
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

This brief recording lasting only 0.72 s was achieved in the south of France at a short distance from an isolated male so that the signal-to-noise ratio is high. As shown in Fig. 8.2, the `orni` waveform is made of five items or echemes. Cicadas produce sound by rapid buckling of a pair of domed tymbals situated on the sides of the first abdominal segment. Each echeme results of the buckling of a single

Fig. 8.1 Pictures of soniferous animals: the Mediterranean cicada *Cicada orni* (Jérôme Sueur) and the Martinique Robber frog *Eleutherodactylus martinicensis* (reproduced with the kind permission of Renaud Boistel)

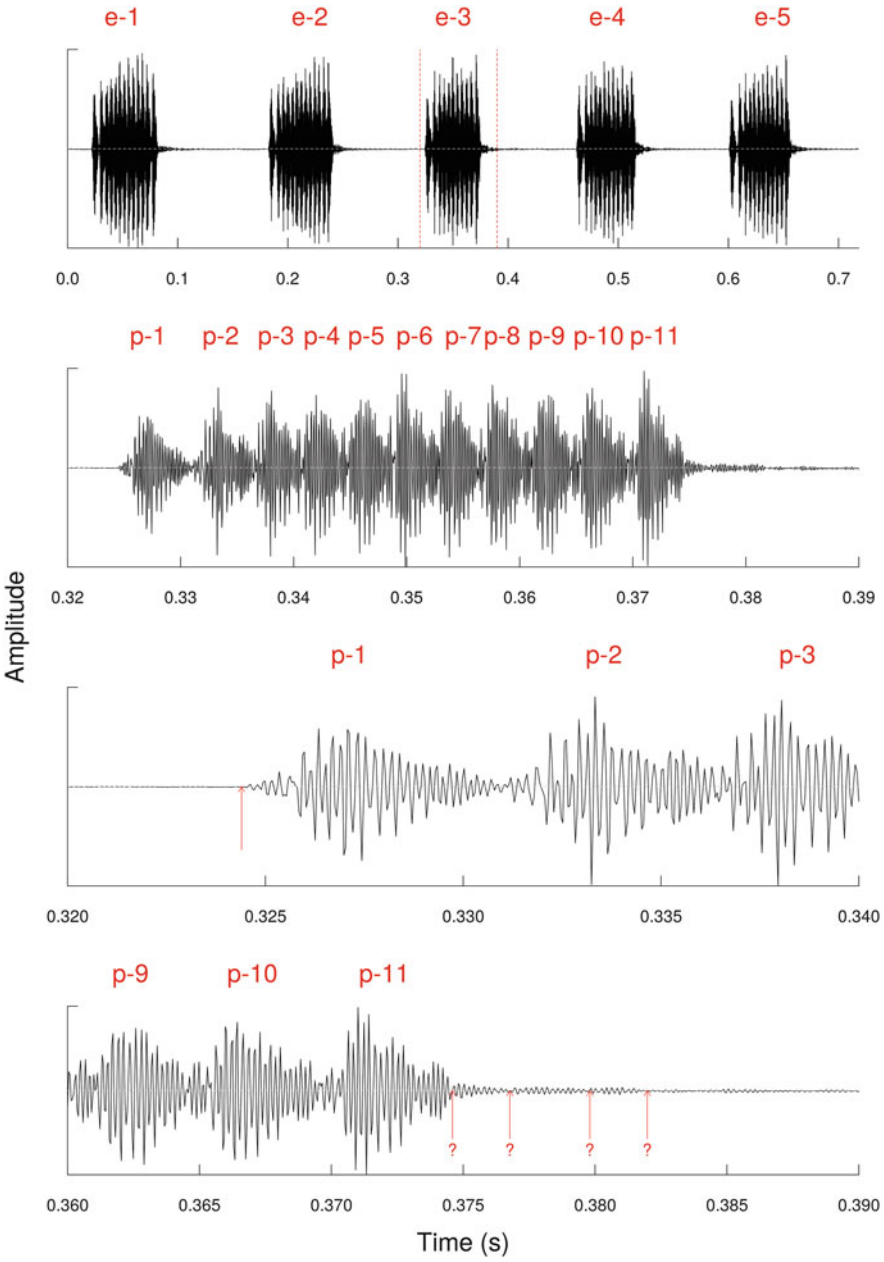


Fig. 8.2 Calling song of *Cicada orni* saved in the dataset `orni`. The song is made by the regular repetition of five syllables or echemes (e-*i*) (first panel). Each echeme is made of about ten pulses (p-*i*) as shown here by zooming in on the third echeme (e-*3*) (second panel). The start of echeme 3 (e-*3*) can be identified clearly (third and fourth panels). The end of the echeme 3 (e-*3*) is more difficult to localize due to echoes (bottom, upward arrows with question marks)

tymbal. If we detail a single echeme, we can see that each echeme is made of about ten pulses that are generated by the successive buckling of stiff ribs that are localized on the surface of each tymbal. It is rather easy to identify the start of each echeme, but there is more confusion regarding the end probably due to some environmental echoes (Fig. 8.2, fourth panel). We will try in the following sections to measure manually and automatically these five echemes and to get an estimation of the pulse repetition rate.

Manual measurement consists in displaying a time × amplitude representation of the sound and to localize visually with mouse clicks time regions of interest (i.e., signal and pause sections). A first direct way to achieve this is to use the function `oscillo()` that was previously detailed in Sect. 5.1, with the argument `identify` turned to `TRUE`:

```
res <- oscillo(orni, identify=TRUE)
```

The oscillogram is then displayed, and the console invites the user with the message `"choose points on the wave"` to click on the graphical device to identify points of interest. The coordinates of the wave's closest point are then displayed in red on the device and saved into the object where the values of `oscillo()` were assigned to, here the object `res`. A try returns the following ten values framing out the five echemes:

```
            time amp
 [1,] 0.02204221  90
 [2,] 0.08290773 278
 [3,] 0.18286869 277
 [4,] 0.24160255 583
 [5,] 0.32455563 157
 [6,] 0.37580604 693
 [7,] 0.46225140  49
 [8,] 0.51581487 910
 [9,] 0.60135315 471
[10,] 0.65845426 333
```

The results are saved into a two-column matrix. The first column includes the time position in s, and the second column returns the amplitude value of the corresponding sample. Here, the first echeme starts at `res[1,]` and stops at `res[2,]`. The first pause between `res[2,]` and `res[3,]` follows.

The argument `identify` of `oscillo()` calls the base function `identify()` that has the great advantage to return the exact values of the plotted object but not the position of the mouse icon on the screen as several other softwares do.

This means that the precision of the measurement is the inverse of the sampling frequency, here $1 \div f_s = 0.000454$ s.

However, the precision of the measurement still depends on the mouse position and so on the size of the graphical display and on the size of the computer screen. This precision can potentially be increased by displaying the amplitude envelope obtained through the Hilbert transform rather than the oscillogram. The Hilbert amplitude envelope better follows the amplitude variations and returns only positive values facilitating the mouse localization.

The precision of the measurements can be assessed empirically by repeating the same measurement on the graphical display. In the following example, we measure 15 times the echeme duration of `orni`. This is obtained by first preparing a matrix made of 15 lines corresponding to the $n = 15$ repetitions and of 10 columns corresponding to the $p = 5 \times 2 = 10$ time measurements. Then, the repetitions are processed through a `for` loop involving `oscillo()` with the argument `identify` turned to `TRUE`, the argument `nidentify` set to 10 to limit the number of measurements to 10 for each repetition, and by indexing the first column of the value where time is stored:

```
n <- 15
p <- 10
res.osc <- matrix(numeric(n*p), nrow=n, ncol=p)
for(i in 1:n){
    res.osc[i,] <- oscillo(orni,
                           identify=TRUE, nidentify=p)[,1]
}
```

We can achieve exactly the same process focusing on a single echeme such that the resolution of the display is increased. Here for the echeme found between 0.32 and 0.39 s, we run:

```
n <- 15
p <- 2
res.osc.single <- matrix(numeric(n*p), nrow=n, ncol=p)
for(i in 1:n){
    res.osc.single[i,] <- oscillo(orni, from=0.32, to=0.39,
                                  identify=TRUE, nidentify=p)[,1]
}
```

We can also run an identical procedure on the Hilbert envelope. In that case, it is first necessary to use `env()` to compute the envelope and then to use the results with `oscillo()` to display the envelope and use the `identify` argument. When calling `oscillo()` it is here necessary to specify the sampling frequency f_s as the value returned by `env()` is a matrix without information on the sampling frequency:

```
n <- 15
p <- 10
res.env <- matrix(numeric(n*p), nrow=n, ncol=p)
envlpe <- env(orni, plot=FALSE)
for(i in 1:n){
    res.env[i,] <- oscillo(envlpe, f=orni@samp.rate,
                           identify=TRUE, nidentify=p)[,1]
}
```

and for the single echeme:

```
n <- 15
p <- 2
res.env.single <- matrix(numeric(n*p), nrow=n, ncol=p)
for(i in 1:n){res.env.single[i,] <-
                  oscillo(envlpe, f=orni@samp.rate,
                          from=0.32, to=0.39,
                          identify=TRUE, nidentify=p)[,1]
           }
```

Eventually, the standard deviation of the results can be computed for each case (all echemes/single echeme; oscillogram/envelope). The results for a trial achieved on 24 inch wide screen (= 60.96 cm) are provided in Table 8.1. As expected, the test shows that measuring on the envelope increases measurement precision and that zooming, through the focus of a single echeme, also enhances the measurement precision.

The DIY box 8.1 provides a solution for manual measurements on a bundle of files; however, automatic measurements might be preferred when handling many files.

Table 8.1 Precision of manual time measurements on the `orni` sound

	Start (5 echemes)	End (5 echemes)	Start (1 echeme)	End (1 echeme)
Oscillogram	0.00067	0.00079	0.00007	0.00093
Envelope	0.00028	0.00041	0.00005	0.00002

The standard deviation of the repeated measurements are shown for the start and end of the echemes, either measured with a complete display ("five-echeme" columns) or with a display focusing on a single echeme ("one-echeme" columns) on the oscillogram or on the amplitude envelope

DIY 8.1 — How to take manually time measurements on a group of `.wav` files

How should we do it when we have a series of `.wav` files to inspect and take time measurements on them? The idea, as usual with batch processing, is to write a loop which each iteration corresponds to a file.
Imagine that we have a series of n `.wav` files named `rec_1.wav`, `rec_2.wav`, ..., `rec_n.wav` in a directory named `sample`. The first step consists in listing the files and to store their names in a character vector. This vector is used in a second step to read successively in a `for` loop the files, to plot the oscillogram, and to save the measurements in a list:

```
setwd("sample")                     # working directory
files <- dir(pattern="^rec.*wav$") # files selection
n <- length(files)                  # number of files
res <- vector("list", n)            # empty list of n items
for(i in 1:n) {                     # loop with n iterations
    res[[i]] <- oscillo(readWave(files[i]), identify=TRUE)
}
```

8.3 Automatic Measurements

The automatization of time measurements relies on the detection of signal and pause events occurring in a recording. The detection of signal events is slightly different from the detection of particular sound patterns, such as a specific bird song or a specific frog call. Here, there is no attempt to identify any peculiar sound but just to infer when there is interesting sound (signal) and when there is nothing interesting to consider (pause). The identification of target sounds is the play game of automatic identification through, for instance, template matching, a question treated in Chap. 17.

A classical and simple solution to detect signal and pause events is to track the amplitude variations along time in reference to a certain amplitude threshold. Anything below a fixed threshold is considered as a pause event and anything above as a signal event. The amplitude variations are estimated through the amplitude envelope, usually the Hilbert amplitude envelope, and the threshold is expressed as a ratio in relation with the maximum of the amplitude envelope. For instance, the maximum value of the Hilbert amplitude envelope of `orni` is 1.7491854×10^4 as obtained with:

```
envlpe <- env(orni, plot=FALSE)
max.env <- max(envlpe)
max.env
[1] 17491.85
```

An amplitude threshold of 5% corresponds to an amplitude value of 874.593 acting as a barrier for a signal/pause decision. A way to refine the method is to apply a duration threshold. The latter, set in s, allows to eliminate signal events that would be too short. For instance, if we know that cicadas usually do not produce sound briefer than 0.025 s, a time threshold set to this value can be applied to remove any signal event shorter than 0.025 s.

The function `timer()` of `seewave` applies both amplitude and time thresholds on the amplitude envelope, either absolute or Hilbert.[1] The functions return graphically and/or numerically the position, the duration of the sound, and pause events. The function `timer()` has several arguments that can be grouped according to their main roles:

input : `wave`, `f`
envelope properties : `envt`, `power`, `msmooth`, `ksmooth`, `ssmooth`, `tlim`,
detection thresholds : `threshold`, `dmin`,
graphical options : `plot`, `plotthreshold`, `col`, `colval`, `xlab`, `ylab`.

The values returned by `timer()` are organized in a list containing six items:

`s` duration of signal event(s) in seconds,
`p` duration of pause event(s) in seconds,
`r` ratio between the signal and silence events(s),
`s.start` start position(s) of signal event(s),
`s.end` end position(s) of signal event(s),
`first` whether the first event detected is a signal or a pause.

Setting the parameters of `timer()` can be rather tricky and, in some way, empiric. There are two important facts to take into account. First, the precision of the measurement should be lower than the differences that the user needs to highlight. For instance, it would be nonsense to try to find time differences about 10^{-3} s with a sound sampled at $f_s = 44{,}100$ Hz averaged with a moving average with a 200 sample window dropping down to $f_s = 44{,}100 \div 200 = 220.5$ Hz that is equivalent to 0.00454 s. Second, it is absolutely mandatory to keep always the same argument settings for all the sounds processed. Departing from this rule may induce artificial time differences due to the use of different methods rather than to true acoustic differences.

[1] The package `soundgen` embeds an alternative to `timer()` named `segment()` which is particularly adapted to voice analysis.

8.3.1 The Cicada Case

What does return `timer()` on `orni` sound? We can start with an amplitude threshold of 5% applied on the Hilbert amplitude envelope:

```
res <- timer(orni, threshold=5, envt="hil", plot=FALSE)
```

To check if the function returned expected results, we can count the number of sound events detected that should equal to five:

```
length(res$s)
[1] 85
```

This is obviously all wrong. The amplitude threshold did not work properly because the amplitude variations of `orni` are rather important, rising and declining strongly. A way to improve the results is to smooth the envelope to reduce these amplitude modulations. We have seen in Sect. 5.2.3 that the envelope can be smoothed calling different techniques, namely, with a moving average, a moving sum, or a moving kernel. The arguments `msmooth`, `ssmooth`, and `ksmooth` of the function `env()` are parsed by the function `timer()` so that we can use them directly with `timer()`. For instance, if we apply a simple moving average with a 30 sample non-overlapping sliding window, we get:

```
res <- timer(orni, threshold=5, msmooth=c(30,0),
             envt="hil", plot=FALSE)
length(res$s)
[1] 8
```

The function detected eight events; this is more reasonable but still wrong. We can increase the size of the smoothing average window to 50:

```
res <- timer(orni, threshold=5, msmooth=c(50,0), envt="hil")
```

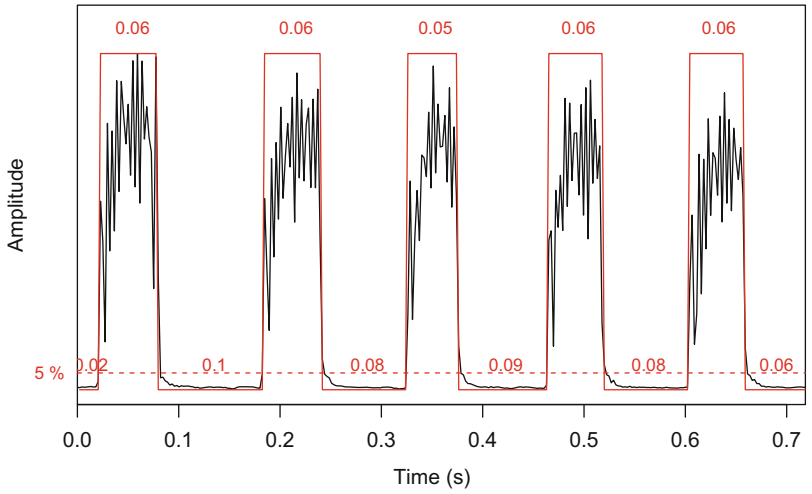


Fig. 8.3 Automatic time measurement of the `orni` sound. The five echemes (signal) and the inter-echeme (pause) separating them are automatically detected with the function `timer()`. The Hilbert amplitude envelope (`envt="hil"`) was smoothed with a moving average (`msmooth=c(50,0)`)

and check the results similarly:

```
length(res$s)
[1] 5
```

The number of echemes is in that case correct, as testified by the plot returned by `timer()` (Fig. 8.3).

We can have a closer look at the results by printing them:

```
res
$s
[1] 0.05930130 0.05930130 0.05245884 0.05702048 0.05702048

$p
[1] 0.02280819 0.10263686 0.08210949 0.08667113 0.08210949
[6] 0.05702048

$r
[1] 0.6578947

$s.start
```

(continued)

```
[1] 0.02280819 0.18474636 0.32615715 0.46528712 0.60441709

$s.end
[1] 0.08210949 0.24404766 0.37861599 0.52230760 0.66143757

$first
[1] "pause"
```

We can get the mean and standard deviation of the echeme (signal) duration and echeme interval (pause) duration:

```
mean(res$s) # mean of signal duration
[1] 0.05702048
sd(res$s)   # standard-deviation of signal duration
[1] 0.002793422
mean(res$p) # mean of pause duration
[1] 0.07222594
sd(res$p)   # standard-deviation of signal duration
[1] 0.02829197
```

To get the mean echeme duration, we first have to know if the recording starts with a pause or not. If it does, as it is the case here, we need to remove the first value of `res$p` to get the correct correspondence between a signal and its following pause:

```
res$first=="pause"        # is first event a pause?
[1] TRUE
period <- res$s+res$p[-1] # period
mean(period)              # mean of the period
[1] 0.13913
sd(period)                # standard-deviation of the period
[1] 0.01699172
```

The ratio between signal and pause duration can be directly printed with:

```
res$r
[1] 0.6578947
```

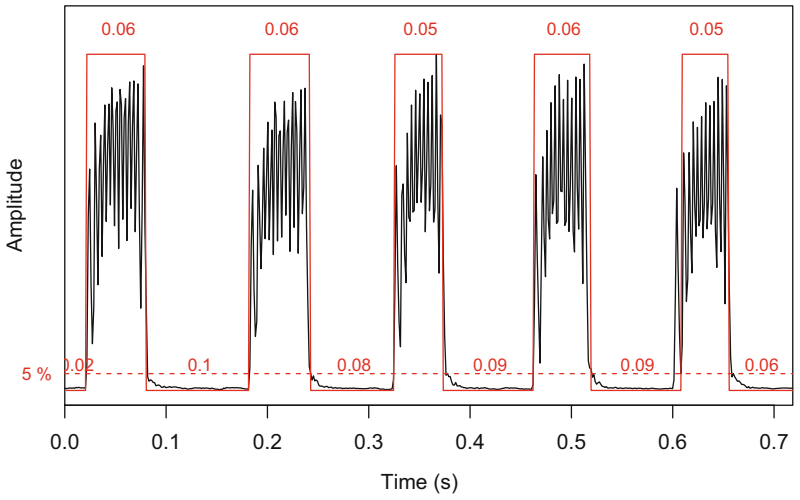


Fig. 8.4 Automatic measurement of the `orni` sound with amplitude and time thresholds. The figure is the graphical output of `timer()` with a smoothing parameter (`msmooth=c(30,0)`), an amplitude threshold (`threshold=5`), and a time threshold (`dmin=0.04`)

We can therefore write that the calling song of the male of *C. orni* saved in the recording `orni` is made of five echemes with a duration of 0.057 ± 0.003 s (mean $\pm$ s.d.) separated by pauses of 0.072 ± 0.028 s, equivalent to a period of 0.139 ± 0.017 s. The ratio between the echeme and the pause duration is 0.658.

We found a solution with the argument `msmooth`, but the problem of detecting the right numbers of echemes could have been solved with other options. We could also have used the time threshold set with the argument `dmin`. If, by chance, we had an a priori knowledge on the duration of the echemes of `orni`, we could stipulate that the duration of the signal events could not be shorter than 0.04 s. We would have done (Fig. 8.4):

```
res <- timer(orni, threshold=5, msmooth=c(30,0),
             dmin=0.04, envt="hil")
```

and we would have obtained:

```
length(res$s)
[1] 5
```

In both case, the use of a smoothing filter seems to be mandatory to get appropriate results. However, this smoothing step has consequences on the precision of the measurements as it corresponds to a downsampling process, as detailed in Chap. 5. Averaging the envelope with a 50 sample sliding window means that the original sampling frequency f_s that was 22,050 Hz corresponding to 0.0000454 s is changed in a new sampling frequency of $22{,}050 \div 50 = 441$ Hz, equivalent to 0.00227 s. This can induce undesired results. A solution to this, as explained in Sects. 5.2.3.1 and 5.2.3, is to use overlapping windows. Using an overlap of 75% for the moving average will reduce the sampling frequency to $(22{,}050 \times 4) \div 50 = 1764$, equivalent to 0.00131 s. The results are obtained with the following code:

```
res <- timer(orni, threshold=5, msmooth=c(50,75),
             envt="hil", plot=FALSE)
length(res$s)
[1] 6
```

This causes another error with one false positive. This can be corrected by increasing slightly the amplitude threshold to 6:

```
res <- timer(orni, threshold=6, msmooth=c(50,75),
             envt="hil", plot=FALSE)
length(res$s)
[1] 5
```

We can also try to smooth the envelope with a moving sum set up with the argument `ssmooth` of `timer()`. The results with the moving sum seems to be correct with a window of 110 samples, that is, with a rather large window (Fig. 8.5):

```
res <- timer(orni, threshold=5, ssmooth=110, envt="hil")
```

```
length(res$s)
[1] 5
```

We can now compare the results obtained with each method, including a manual measurement done as accurately as possible zooming successively on each echeme. This manual measurement can be considered as a reference.

The results that are summarized in Table 8.2 show that the measurements are quite close to each other and that the results obtained with `msmooth=c(50,75)`

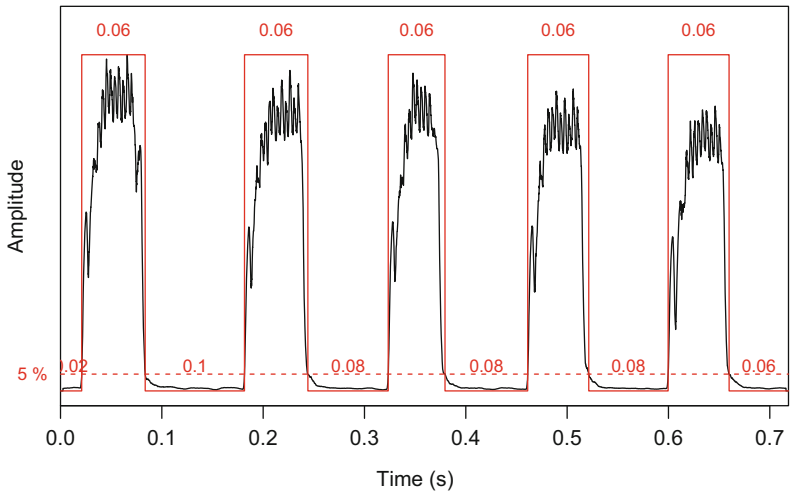


Fig. 8.5 Automatic measurement of the `orni` sound with a moving sum. The figure is the graphical output of `timer()` with a smoothing parameter using sum (`ssmooth=100`) and an amplitude threshold (`threshold=6`)

Table 8.2 Comparison of automatic time measurements on the `orni` sound

	Manual	Automatic 1	Automatic 2	Automatic 3
Automatic 1	0.0022			
Automatic 2	0.0187	0.0188		
Automatic 3	0.0090	0.0082	0.0116	
Automatic 4	0.0143	0.0149	0.0089	0.0096

The comparison is made between a manual accurate measurement and four automatic trials with the function `timer()`. Signal duration is compared by computing an Euclidian distance matrix. The lower is the value, the closest are the methods. Automatic 1: `threshold=5` and `msmooth=c(50,0)`; automatic 2: `threshold=5` and `msmooth=c(30,0)`; automatic 3: `threshold=6` and `msmooth=c(50,75)`; automatic 4: `threshold=5` and `ssmooth=110`

are the closest from those obtained manually with a high precision. Running `timer()` with a moving average and good window overlap seems therefore to be the best option in terms of precision. However, such process can be quite demanding in terms of computing and then difficult to undertake. There is therefore a trade-off between precision and time process.

8.3.2 The Frog Case

The `orni` sound is a rather nice recording with some echoes but with a good signal-to-noise ratio. However, recordings made outdoor are often filled with background sound (or noise) that can make the detection of the sounds of interest difficult. In

the following example, the automatic detection of the vocalizations of a frog is more challenging. This small frog, the Martinique robber frog *Eleutherodactylus martinicensis* commonly found in forested areas of the Martinique island in the Lesser Antilles (Fig. 8.1), produces a loud call that is easy to localize and identify (Lemon 1971). However, the frog is not alone; it is accompanied by numerous congeners which generate an important background sound reducing the signal-to-noise ratio and therefore challenging the automatic measurements of the vocalizations of the focal male.

This recording, which does not come with any package, is stored in a directory `sample` and can be loaded with the function `readWave()`:

```
frog <- readWave("sample/Eleutherodactylus_martinicensis.wav")
```

The resulting `Wave` object has the following main properties:

```
frog

Wave Object
Number of Samples:      316602
Duration (seconds):     19.79
Samplingrate (Hertz):   16000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The waveform of `wave` is obtained with (Fig. 8.6):

```
oscillo(frog)
```

The recording is a succession of 17 vocalizations produced by a territorial male. Each vocalization can be divided into two notes with different frequencies. In terms of frequency, the first note can be described as a linear frequency modulation sound starting at 1850 Hz and stopping at 2100 Hz, and the second note as a linear frequency modulation sound beginning as well at 2800 Hz and ending at 3750 Hz (Fig. 18.23). We wish to know the duration of the two-note vocalizations and the duration of the pauses in between. We run a first try with `timer()` set with a moving average with a 90% overlapping window made of 100 samples (`msmooth = c(100,90)`) and with an amplitude threshold set to 5% (`threshold =5`):

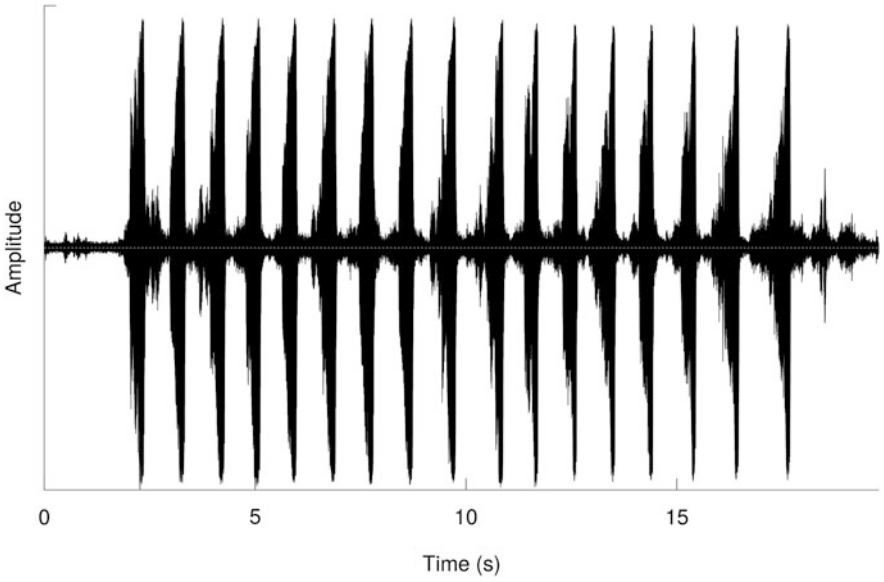


Fig. 8.6 Oscillogram of the frog *Eleutherodactylus martinicensis*. The recording made Renaud Boistel is a succession of 17 two-note calls of a focal recorded male, with important background sound due to other vocalizing males

```
res <- timer(frog, msmooth=c(100,90), threshold=5,
             env="hil", plot=FALSE)
```

As for the case of `orni`, the results could be visualized with `plot=TRUE` but to facilitate the process we control the output of `timer()` by numbering the number of sound events detected with:

```
length(res$s)
[1] 204
```

It clearly appears that the detection took a bad direction. The main issue comes from the low signal-to-noise ratio of the recording. A solution is to apply a power function to the amplitude envelope increasing artificially the difference between the amplitude values of the close-up male and the amplitude values of the background congeners. This can be processed with the help of the argument `power`, here set to 2 meaning that the smoothed amplitude envelope is squared before to apply the amplitude threshold:

```
res <- timer(frog, msmooth=c(100,90), threshold=5, power=2,
             envt="hil", plot=FALSE)
length(res$s)
[1] 34
```

The result improved but is still far away from reality. To tune the detection, we specify that there should not be sound events shorter than 0.2 s with the argument `dmin`:

```
res <- timer(frog, msmooth=c(100,90), threshold=5,
             power=2, envt="hil", dmin=0.2)
```

```
length(res$s)
[1] 17
```

This time, the detection seemed to have worked properly as shown in Fig. 8.7. The temporal features of the recording can be then calculated with:

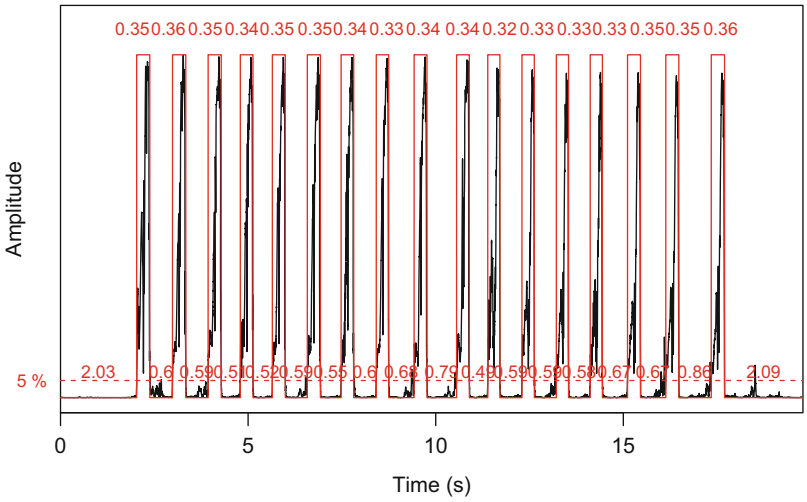


Fig. 8.7 Automatic time measurement of the frog *Eleutherodactylus martinicensis*. The 17 two-note vocalizations (signals) and the pauses separating them are automatically detected with the function `timer()`. The Hilbert amplitude envelope (`envt="hil"`) was squared (`power=2`) and smoothed with a moving average (`msmooth=c(100,90)`). The results were filtered with a 0.2 s time threshold (`dmin=0.2`)

```
mean(res$s)                 # mean of signal duration
[1] 0.3409554
sd(res$s)                   # standard deviation of signal duration
[1] 0.01032279
mean(res$p)                 # mean of the pause duration
[1] 0.777299
sd(res$p)                       # signal/pause  ratio
[1] 0.4757469
res$r
[1] 0.4142723
res$first=="pause"          # is first event a pause?
[1] TRUE
period <- res$s+res$p[-1]   # calculation of the period
mean(period)                # mean of the period
[1] 1.044564
sd(period)                  # standard-deviation of the period
[1] 0.3734283
```

We can therefore describe the vocalization of this peculiar male of *E. martinicensis* as a sequence of 17 vocalizations lasting 0.341 ± 0.01 s (mean $\pm$ s.d.) separated by pauses of 0.777 ± 0.476 s, equivalent to a period of 1.045 ± 0.373 s. The ratio between the vocalization and the pause duration is 0.414.

We can also use the time localization of the signals for zooming and annotating the oscillogram of the four first vocalizations. This is achieved by using the arguments `from` and `to` of `oscillo()` with a short delay of 0.1 s before and after the zoom limits to add some time around the selection (Fig. 8.8):

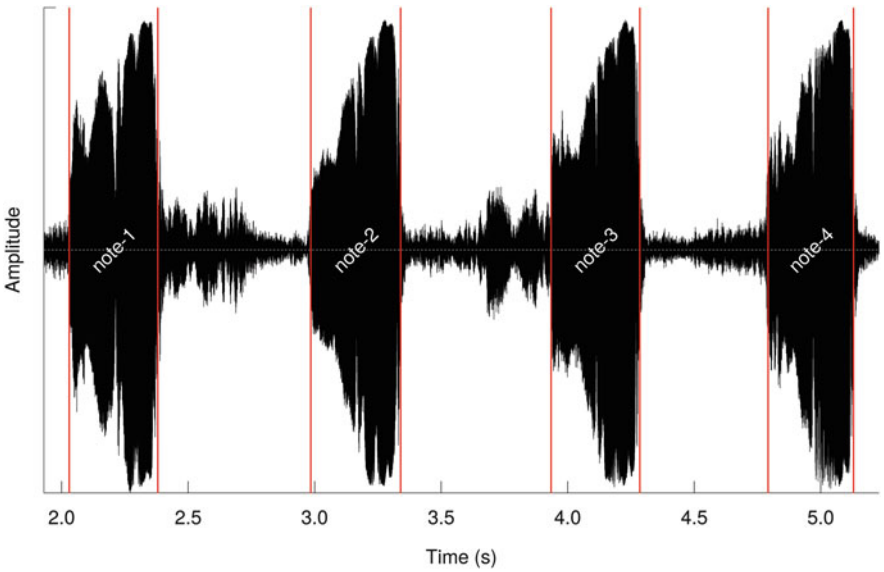


Fig. 8.8 Graphical use of `timer()` results. The results returned by `timer()` are used to zoom on the first four vocalizations, to label and to frame these vocalizations

```
oscillo(frog,
        from=res$s.start[1]-0.1,          # zoom lower limit
        to=res$s.end[4]+0.1)              # zoom upper limit
x <- res$s.start+
    (res$s.end-res$s.start)/2             # vocalization center
text(x=x, y=0,                            # label positions
     labels=paste("note", 1:4, sep="-"), # label texts
     col="white", srt=45)                 # label decoration
abline(v=c(res$s.start, res$s.end),       # vertical lines
       col="red", lwd=3)                  # to frame out each call
```

The results provided by `timer()` can be compared with the manual measurements obtained by zooming on each vocalization (Fig. 8.9). The manual and the automatic processes returned similar values, but the automatic option clearly returned shorter vocalization and longer pause than the manual options. This suggests that `timer()` framed the vocalizations more sharply. This difference

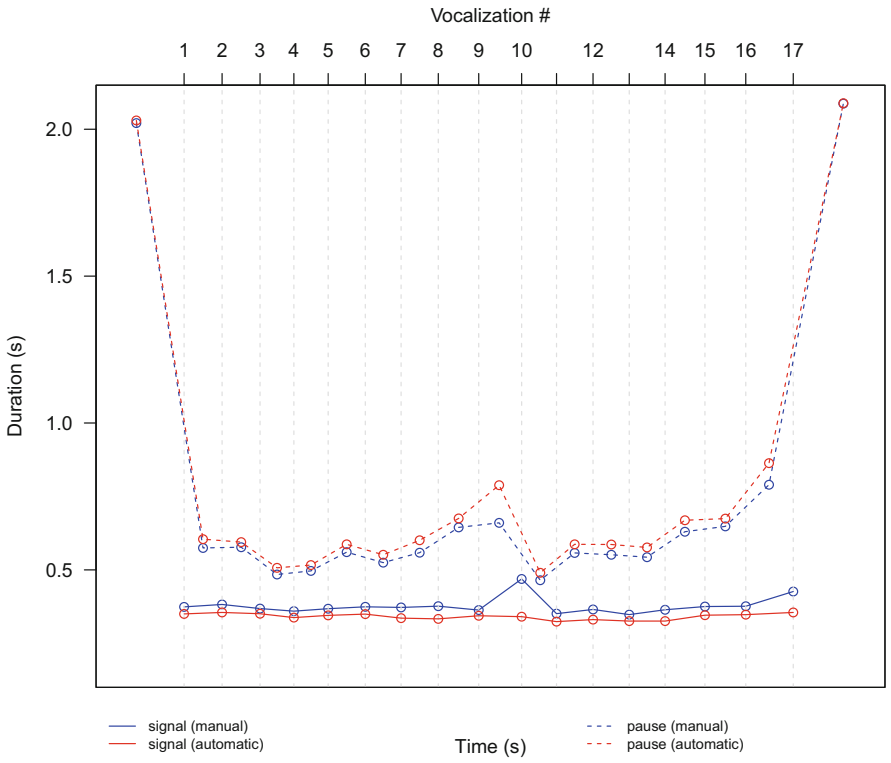


Fig. 8.9 Comparison of manual and automatic measurements. The plot shows against time the duration the 17 vocalizations (signal) and pauses of the calling sequence of the frog *E. martinicensis* obtained manually using the argument `identify` of `oscillo()` (manual) and the estimation returned by the function `timer()` (automatic)

is even more apparent for the tenth vocalization where the manual measurement was 0.469 s and the automatic one was estimated to be only 0.341 s. The start of this vocalization is actually very difficult to assess. This is mainly due to another male producing an almost synchronized vocalization in the vicinity of the focal male. The manual measurement seems to have overestimated the duration of the vocalization giving a duration out of the range of the duration obtained for the 16 other vocalizations.

This raises the question of the check of the results. How to detect if there was any mistake in the automatic process? How to be confident with the results? A solution is to look for any strange features in the distribution of the results according to the different settings of `timer()`. This can be realized with a boxplot as illustrated in Fig. 8.10. The first box has many outliers, the second box shows a wide and highly skewed distribution, and the third box looks symmetric, sharp, and without outliers suggesting relevant results.

There is no ideal method to take time measurements; a mistake can always be introduced either by the observer or by the automatic system. It is therefore of prime importance to always check the results visually, either directly on the waveform (oscillogram, amplitude envelope) or on a plot displaying the distribution of the results.

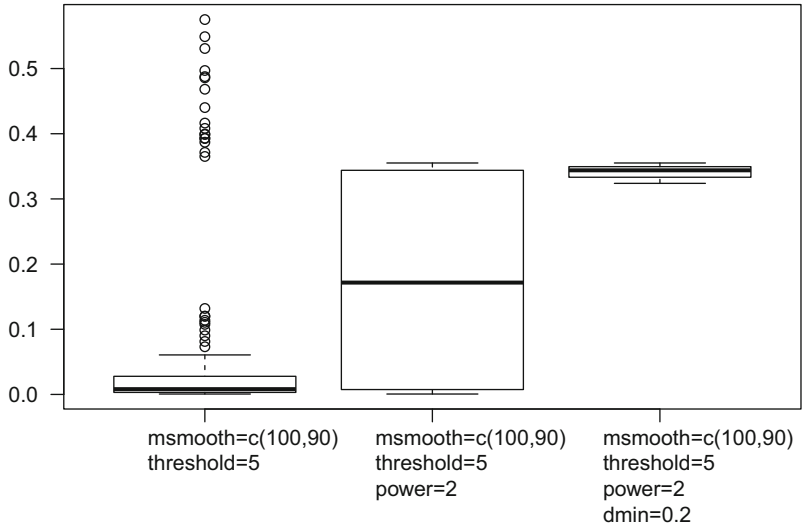


Fig. 8.10 Distribution of the automatic measurements according to different `timer()` settings on the 17 vocalizations (signal) and pauses of the calling sequence of the frog *Eleutherodactylus martinicensis*

8.4 Amplitude Modulation Analysis

Sound may show regular amplitude variations. These variations, or amplitude modulations (AMs), can be related to the start and stop of the sound production, to regular increases and decreases of the sound due to the mechanism of the sound apparatus. These amplitude modulations can be measured manually like any other time parameter, but they can also be obtained by a frequency analysis of the Hilbert amplitude envelope.

Sound frequency decomposition will not be detailed here as it constitutes the aim of the next chapters (Chaps. 9–11), but the idea is not that difficult to understand. Any pattern repeated regularly follows more or less a sinusoidal function with a peculiar frequency. This frequency, expressed in Hz, is an estimation of the number of times the event occurs per second. If we can get the frequency of the amplitude modulation due to the repetition of a syllable, a note, or a pulse, we can then know how many times this sound event was produced per second. This opens, for instance, the possibility to estimate the pulse repetition rate of *C. orni*, something which is fairly difficult to assess with the time methods introduced in the above sections.

The amplitude modulation analysis is divided into three steps: (1) computation of the Hilbert amplitude envelope, (2) decomposition in frequency of the envelope using a short-time Fourier transform (STFT) (see Chap. 11), and (3) identification of the resulting mean frequency spectrum of the main peaks that correspond to the repetition rate of each amplitude modulations occurring in the signal.

This amplitude modulation analysis is implemented in the function `ama()` of `seewave`. By default `ama()` computes the Hilbert envelope, the STFT, and returns the mean frequency spectrum. We already got through the Hilbert transform, but the short-time Fourier transform sounds more mysterious. Briefly, the STFT consists in decomposing the song with a window that slides along the sound. The largest the sliding is the finest is the frequency decomposition but the crudest is the time resolution. A way to reduce this uncertainty between frequency and time is to overlap successive windows as we already did when averaging the amplitude envelope with `timer()`. The window length and the window overlap are controlled with the arguments `wl` and `ovlp` of `ama()`.

8.4.1 The Cicada Case

Following this short introduction to frequency analysis, we first try `ama()` on the cicada song with a window length of 1024 samples (`wl=1024`) to test whether we can find the repetition rate of the echemes and of the pulses (Fig. 8.2). The function returns a line plot with frequency on the *x*-axis and relative amplitude without unit on the *y*-axis.

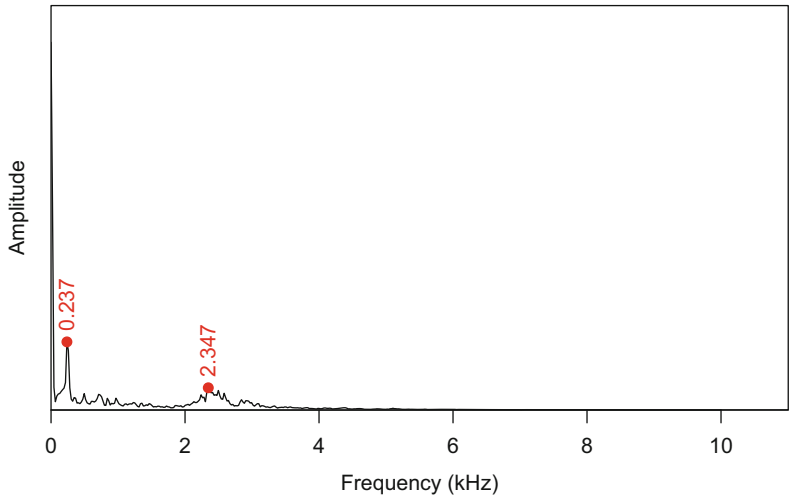


Fig. 8.11 Amplitude modulation analysis of the `orni` sound: fast amplitude modulations. The function `ama()` shows two peaks corresponding to the pulse repetition rate (0.237 kHz) and the carrier frequency (2.347 kHz)

```
ama(orni, wl=1024)
```

We can see two peaks that can be identified by turning to `TRUE` the argument `identify` in the same way we did with the oscillogram and the envelope[2]:

```
res <- ama(orni, wl=1024, identify=TRUE)
```

Once the peak coordinates have been caught, we add them on the graphic with the low-level plot functions `points()` and `text()` (Fig. 8.11):

```
ama(orni, f=22050, wl=1024)
points(x=res$freq, y=res$amp, pch=19, col=2) # peak point
text(x=res$freq, y=res$amp,                  # peak label
     labels=as.character(round(freq,3)),
     col=2, adj=-0.2, srt=90)
```

[2]Frequency peaks can also be identified automatically with the function `fpeaks()`, see Sect. 10.1.3.1.

The first 0.237 kHz frequency peak corresponds to the repetition rate of the pulses. The second frequency peak, found at 2.347 kHz, is the fundamental frequency of pulses, that is, the frequency of the elementary oscillations found in the pulses. It seems that we here miss a frequency peak for the repetition rate of the echemes. The window of the STFT was set to 1024 samples, that is, to $wl \div f_s = 1024 \div 22{,}050 = 0.0464$ s. This window is actually not wide enough to include the amplitude modulation due to the echemes that have a period of 0.14 s. We therefore drastically increase the window length to embrace all modulations. In the following, the window length is set to 15,000 samples, slightly less than the total number of samples of `orni` (15,842). We also need to zoom in on low frequencies using the argument `flim` that works as the argument `xlim` of `plot()`. `flim` requires frequency in kHz, so that a zoom between 0 and 100 Hz is coded with `flim=c(0,0.1)`:

```
res <- ama(orni, wl=15000, flim=c(0,0.1), identify=TRUE)
```

And the plot is obtained with (Fig. 8.12):

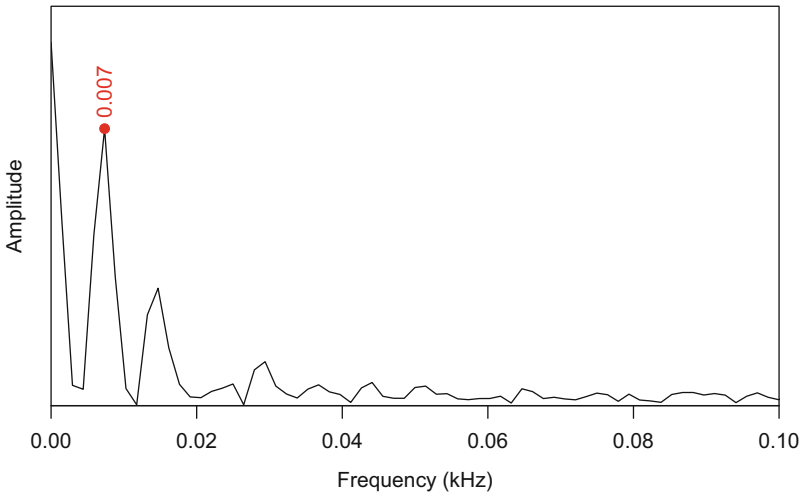


Fig. 8.12 Amplitude modulation analysis of the `orni` sound: slow amplitude modulations. The function `ama()` set with a large window shows a dominant peak corresponding to the echeme repetition rate (0.007 kHz)

```
ama(orni, wl=15000, flim=c(0,0.1))
points(x=res$freq, y=res$amp, pch=19, col=2) # peak point
text(x=res$freq, y=res$amp,                  # peak label
     labels=as.character(round(res$freq,3)),
     col=2, adj=-0.2, srt=90)
```

In that case, a new frequency peak appears at 0.00735 kHz which gives an echeme repetition rate of 7.35 Hz. This information could also be extrapolated from the `timer()` result by dividing the number of signals found by the duration of the recording obtained with `duration()`:

```
res <- timer(orni, threshold=5, msmooth=c(50,0),
             envt="hil", plot=FALSE)
length(res$s)/duration(orni)
[1] 6.959349
```

This estimation is rather good because there is no long pause neither at the start nor at the end of the recording. What would happen if we would artificially make the recording longer by adding 2 s of pause? Here is first the code to do this manipulation (see Sect. 6.4):

```
orni.with.silence <- addsilw(orni, d=2,
                             at="start", out="Wave")
orni.with.silence <- addsilw(orni.with.silence, d=2,
                             at="end", out="Wave")
```

We now run `timer()` so that we would obtain a similar number of signals (5):

```
res <- timer(orni.with.silence, threshold=5, msmooth=c(50,0),
             envt="hil", plot=FALSE)
length(res$s)
[1] 5
```

The repetition rate has changed and is manifestly not correct:

```
length(res$s)/duration(orni.with.silence)
[1] 1.059668
```

If `timer()` does not seem to properly do the job, `ama()` is not sensitive to such long period of silence surrounding the sequence of echemes as it returns exactly the same result as with the original data `orni`:

```
ama(orni.with.silence, wl=15000, flim=c(0,0.1), identify=TRUE)
```

```
Choose points on the spectrum
$freq
[1] 0.00735

$amp
[1] 0.7550001
```

In that case `ama()` should be then preferred to `timer()`.

8.4.2 The Frog Case

What does `ama()` return with the vocalization of *E. martinencis*? We here also use the argument `ovlp` to slightly increase the time resolution of the STFT, and we also zoom in amplitude with the argument `alim` because peaks above 300 Hz are flattened by an important amount of energy below 300 Hz:

```
res <- ama(frog, wl=1024, ovlp=85, alim=c(0,0.02), identify=TRUE)
```

The visual display is returned thanks to (Fig. 8.13):

```
ama(frog, wl=1024, ovlp=85, alim=c(0,0.02))
points(x=res$freq, y=res$amp, pch=19, col=2) # peak point
text(x=res$freq, y=res$amp,                  # peak label
     labels=as.character(round(res$freq,3)), col=2, pos=3)
```

We observe three peaks; the second peak is the fundamental frequency of the first note at $f_2 = 1.938$ kHz; the third peak is the fundamental frequency of the second note at $f_3 = 3.141$ kHz. The first peak is at $f_1 = 1.219$ kHz: it corresponds to $f_1 = f_3 - f_2 = 3.141 - 1.938 = 1.203$ kHz, a signature of an interference or beating between f_2 and f_3 (see Sects. 10.1.4.2 and 18.4.1).

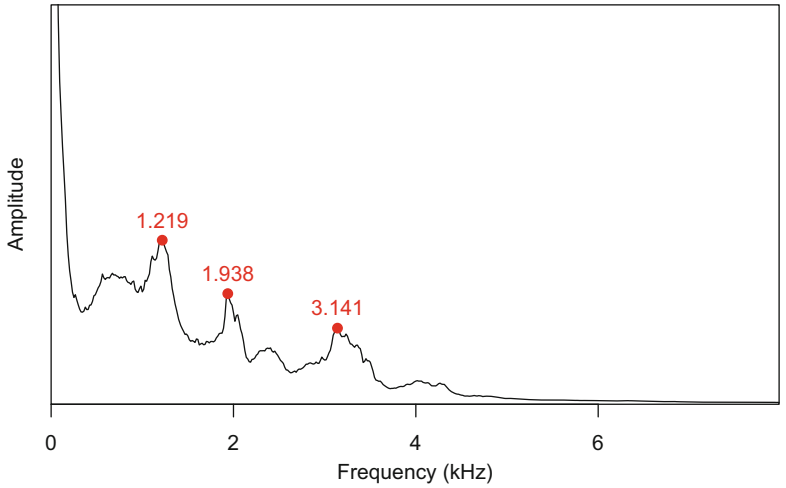


Fig. 8.13 Amplitude modulation analysis of the frog *Eleutherodactylus martinicensis*: fast amplitude modulations. The function `ama()` shows three peaks corresponding to the fundamental frequency of the first note (1.938 kHz), the fundamental frequency (3.141 kHz) and the beating between these two frequencies (1.219 kHz)

As it was the case for cicada sound `orni`, it seems that there is no peak dedicated to the vocalization repetition rate. We need to increase the resolution of the analysis by drastically increasing the window length so that the Fourier analysis get enough signal to detect slow AMs. This is obtained by setting `wl` $= 2^{18} = 262{,}144$ and by zooming this time along the frequency axis:

```
res <- ama(frog, wl=2^18, flim=c(0,0.010), ovlp=85, identify=TRUE)
```

The code for the plot is (Fig. 8.14):

```
ama(frog, wl=2^18, flim=c(0,0.010), ovlp=85)
points(x=res$freq, y=res$amp, pch=19, col=2) # peak point
text(x=res$freq, y=res$amp,                  # peak label
     labels=as.character(round(res$freq,3)), col=2, pos=3)
```

We find a repetition rate of 1.04 Hz that is a bit more than one vocalization per s.

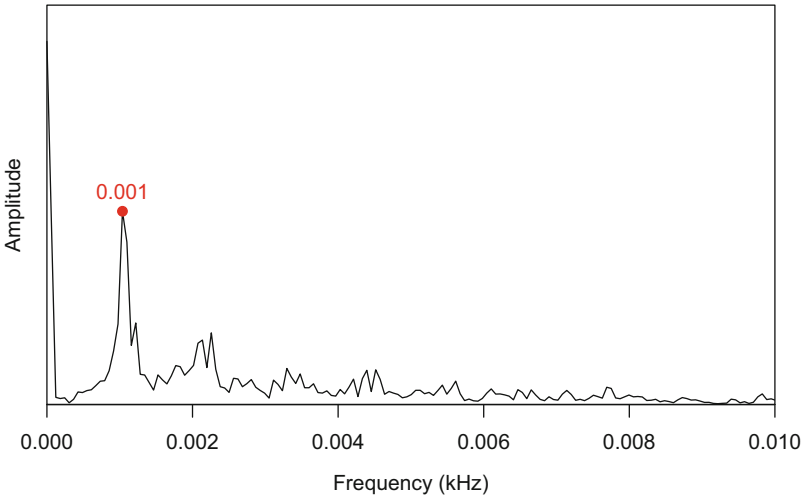


Fig. 8.14 Amplitude modulation analysis of the frog *Eleutherodactylus martinicensis*: slow amplitude modulation. The function `ama()` set with a large window shows a dominant peak corresponding to the vocalization repetition rate (0.001 kHz)

Chapter 9
Introduction to Frequency Analysis: The Fourier Transformation

The previous chapters were dedicated to amplitude and time; it is now time to explore the third dimension of sound: frequency. Frequency plays an important role in speech, music, and animal acoustic communication. It is therefore crucial to describe properly the frequency features of the studied sound. To measure the frequency of a pure tone is made possible by directly scrutating the waveform and by estimating the time period T. However, this measurement is arduous when the sound to analyze is made of several frequencies possibly changing with time. It is then necessary to find a way to travel back and forth between the time domain and the frequency domain. The wonderful adventure of time-frequency travel has been made possible thanks to the Fourier transformation and its inverse version, a mathematical vehicle which is almost systematically used in linear acoustics.

9.1 From Time to Frequency and Back

The Fourier transformation, or Fourier analysis, is due to the French mathematician Jean-Baptiste Joseph Fourier (1768–1830) (Fig. 9.1) who had the idea that a complex waveform can be expressed into an infinite sum of simple waveforms each with its own frequency, amplitude, and phase (Fig. 9.2).[1] The Fourier transformation operates like a triangular prism that separates white light into different color lights. The Fourier transformation is extremely popular as it can be used to analyze a majority of time data. Applications of the Fourier transform can be found in almost all scientific and engineering disciplines, including medicine, astronomy, optics, thermodynamics, statistics, economy, communication, and even imaging sciences. In acoustics, the Fourier transformation appears almost everywhere. It is commonly

[1]For a complete description of the Fourier analysis, see Hartmann (1998, Chapters 5 and 8) and Das (2012, Chapters 2 and 3).

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_9

Fig. 9.1 Jean-Baptiste Joseph Fourier (1768–1830). Engraving by Jules Boilly, around 1823 (Public Domain)

used to identify the individual frequency components of a sound (Fig. 9.2), whatever the nature of the sound: speech, music, animal vocalization, environmental sound, or others.

However, the Fourier transformation is not a magic tool that works in all situations. The Fourier transformation is adapted to the frequency decomposition of periodic signals or aperiodic but continuous signals. Other solutions, such as the wavelet transform, should be called to analyze transient signals.[2] The analytic signal and the zero-crossing method can also provide very valuable frequency information (see Chap. 13).

In the next section, we will discover the two main types of the Fourier transformation, namely, the Fourier series (FS) which works for periodic signals only and the Fourier transform (FT) which can treat any type of signal, periodic or aperiodic but continuous (see Table 9.1 for a list of Fourier family members).

9.2 Fourier Series

9.2.1 Periodicity

The Fourier series is a special case of the Fourier transformation for periodic functions. A sound $s(t)$ with a fundamental period T follows the rule:

$$s(t + mT) = s(t)$$

[2]The wavelet transforms are treated in other books, such as Nason (2008) that is accompanied by the package `wavethresh`, Percival and Walden (2000) by the package `wmtsa`, and Gençay et al. (2001) by the package `waveslim`.

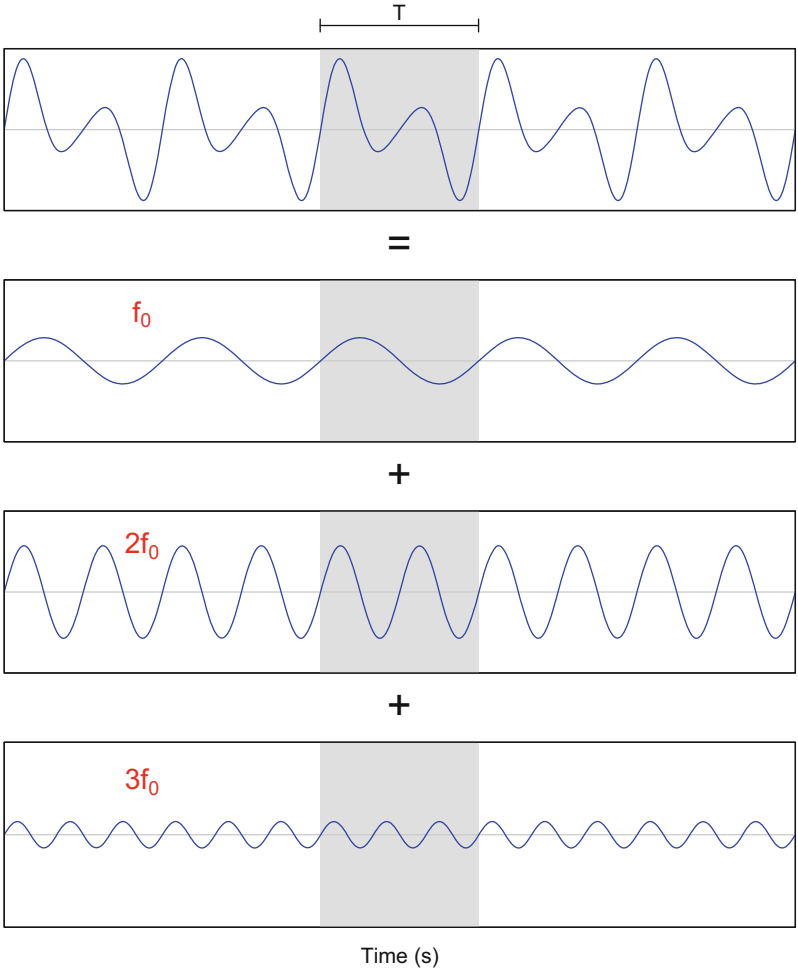


Fig. 9.2 Fourier transformation principle. Any complex waveform can be decomposed into a sum of simple waveforms. Here the top waveform with a period T is decomposed into the addition of three simple waveforms ($n = 3$) related by a fundamental frequency f_0

where m is an integer. This means that the sound $s(t)$ has the same value after having travelled along time during T s. A 440 Hz pure tone produced by a tuning fork, as introduced in Chap. 2, is a periodic function of time with a period $T = 1 \div 440 = 0.00227$ s.

However periodicity does not mean simplicity as periodic sound can have complex patterns due to the addition of several waves that cannot be identified manually (Fig. 9.3). The Fourier series operates a decomposition such that the amplitude, frequency, and phase of these primary waves can be estimated separately.

Table 9.1 The Fourier transformation family: acronyms, complete name, and short definition of the different Fourier transformations

FS	Fourier series	Decomposition of a periodic signal into an infinite sum of harmonics
FT	Fourier transform	Transformation of an infinite signal from the time (or spatial) domain to the frequency domain
DFT	Discrete Fourier transform	Transformation of a discrete time-limited signal to a discrete frequency spectrum
FFT	Fast Fourier transform	Mathematical tool and algorithm to compute the DFT
STFT	Short-time Fourier transform	Sliding FT along an infinite signal, see Chap. 11
STDFT	Short-time discrete Fourier transform	Sliding DFT along a time-limited signal, see Chap. 11

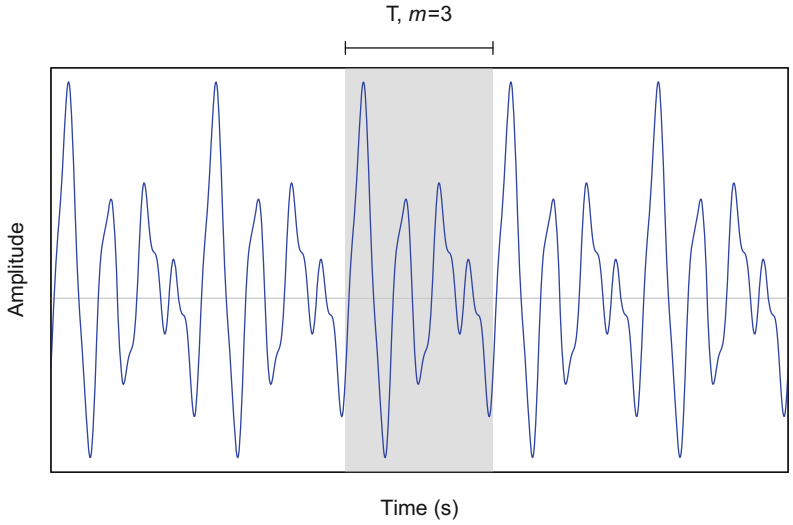


Fig. 9.3 A periodic waveform. The waveform, possibly a sound, is made of five repetitions of the same pattern. The waveform follows the equation $s(t + mT) = s(t)$, with T the period and $m = \{1, 2, 3, 4, 5\}$

The Fourier series can be written in different ways, either with developed trigonometric equations, compact trigonometric equations, or with the help of complex numbers.

9.2.2 Trigonometric Fourier Series

The Fourier transformation says that a periodic signal $s(t)$ can be written as a sum of a constant and an infinite series of sine and cosine functions:

$$s(t) = A_0 + \sum_{n=1}^{\infty} [A_n \cos(\omega_n t) + B_n \sin(\omega_n t)]$$

The constant A_0 corresponds to the average amplitude of the signal $s(t)$ or direct current (DC) in the electricity domain, written as:

$$A_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} s(t)\,\mathrm{dt}$$

The cosine and sine functions are time functions with a specific angular frequency ω_n, with ω_0 the fundamental frequency and $\omega_{n>1}$ the harmonics such that:

$$\omega_n = n\omega_0 = \frac{2\pi n}{T}$$

where n is an integer.

Each cosine and each sine function are weighted by a Fourier coefficient, A_n and B_n, respectively. These coefficients can be obtained with:

$$A_n = \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} s(t) \cos(\omega_n t)\,\mathrm{dt}$$

$$B_n = \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} s(t) \sin(\omega_n t)\,\mathrm{dt}$$

The symmetrical limits of the integral $[-\frac{T}{2}, \frac{T}{2}]$ can be changed to any limits that differ by one period T, such as $[0, T]$ or $[T, 2T]$.

The waveform shown in Fig. 9.3 obeys to the following equation:

$$s(t) = 0.5 + 0.3\cos(t) + 2\sin(2t) + 3\sin(3t) - 4\cos(4t) + \sin(10t)$$

Imagine that this waveform is a sound sampled at a frequency $f_s = 44{,}100$ during 1 s with a fundamental frequency $f_0 = 440\,\mathrm{Hz}$. The waveform is therefore made of 44,100 samples with a period $44{,}100 \div 440 = 100.2273$ samples that can be approximated to 100 samples. We can try to find the coefficients (A_n, B_n) with

the formula above. We start by generating the 440 Hz sound with:

```
f <- 44100                          # sampling frequency
t <- seq(1/f, 1, length.out=f)      # time
T <- 1/440                          # 440 Hz period
w0 <- 2*pi/T                        # 440 Hz angular frequency
h0 <- 0.3*cos(w0*t)                 # 440 Hz fundamental frequency
h1 <- 2*cos(2*w0*t)                 # 880 Hz harmonic
h2 <- 3*sin(3*w0*t)                 # 1320 Hz harmonic
h3 <- -4*cos(4*w0*t)                # 1760 Hz harmonic
h10 <- 10*sin(10*w0*t)              # 4400 Hz harmonic
s <- 0.5 + h0 + h1 + h2 + h3 + h10 # final signal
```

First, the constant A_0 is directly obtained by computing the average of the original sound, returning the expected value of 0.5:

```
A0 <- mean(s)
A0
[1] 0.5
```

We then write a function to compute each series of coefficients, A_n and B_n. We need to convert the period T which was in s in number of samples by applying $T_{samp} = T \times f_s$, and we calculate the angular frequencies ω_n with $\omega_n = 2\pi n \div T$. We also change the limits of the sum from $[-T/2, T/2]$ to $[0, T]$ to make the code more convenient:

```
Tsamp <- round(T*f)      # T in number of samples
dt <- 1:Tsamp            # time of integration
coeffA <- function(n) { # function to compute A_n
    2/Tsamp * sum(s[dt] * cos((2*pi*n/Tsamp)*dt))
}
coeffB <- function(n) { # function to compute B_n
    2/Tsamp * sum(s[dt] * sin((2*pi*n/Tsamp)*dt))
}
```

We apply straightforward the new functions to obtain A_n and B_n with n varying from 1 to 10. This operation is facilitated with the help of the base built-in loop function `sapply()`:

```
A <- sapply(1:10, FUN=coeffA)
B <- sapply(1:10, FUN=coeffB)
data.frame(A=round(A,1), B=round(B,1))
      A    B
1   0.3  0.0
2   2.0  0.0
3   0.0  3.0
4  -4.0 -0.1
5   0.0  0.0
6   0.0  0.0
7   0.0 -0.1
8   0.0 -0.1
9   0.0 -0.2
10 -0.7 10.0
```

We can now use these coefficients to reconstruct the original signal with (Fig. 9.4):

```
res <- matrix(rep(NA, f*10), nrow=10)
for(n in 1:10){
    res[n, ] <- A[n]*cos(2*pi*n*t/T) +
        B[n]*sin(2*pi*n*t/T)
    }
s.synth <- A0 + colSums(res)
```

9.2.3 Compact Fourier Series

There is a more direct way to write the Fourier series using an amplitude-phase form with a single series of cosine functions. This is also known as the compact Fourier series:

$$s(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(\omega_n t + \varphi_n)$$

where $C_0 = A_0$, C_n is the amplitude of the cosine, ω_n is the angular frequency, and φ_n is the phase.

C_n is computed with:

$$C_n = \sqrt{A_n^2 + A_n^2}$$

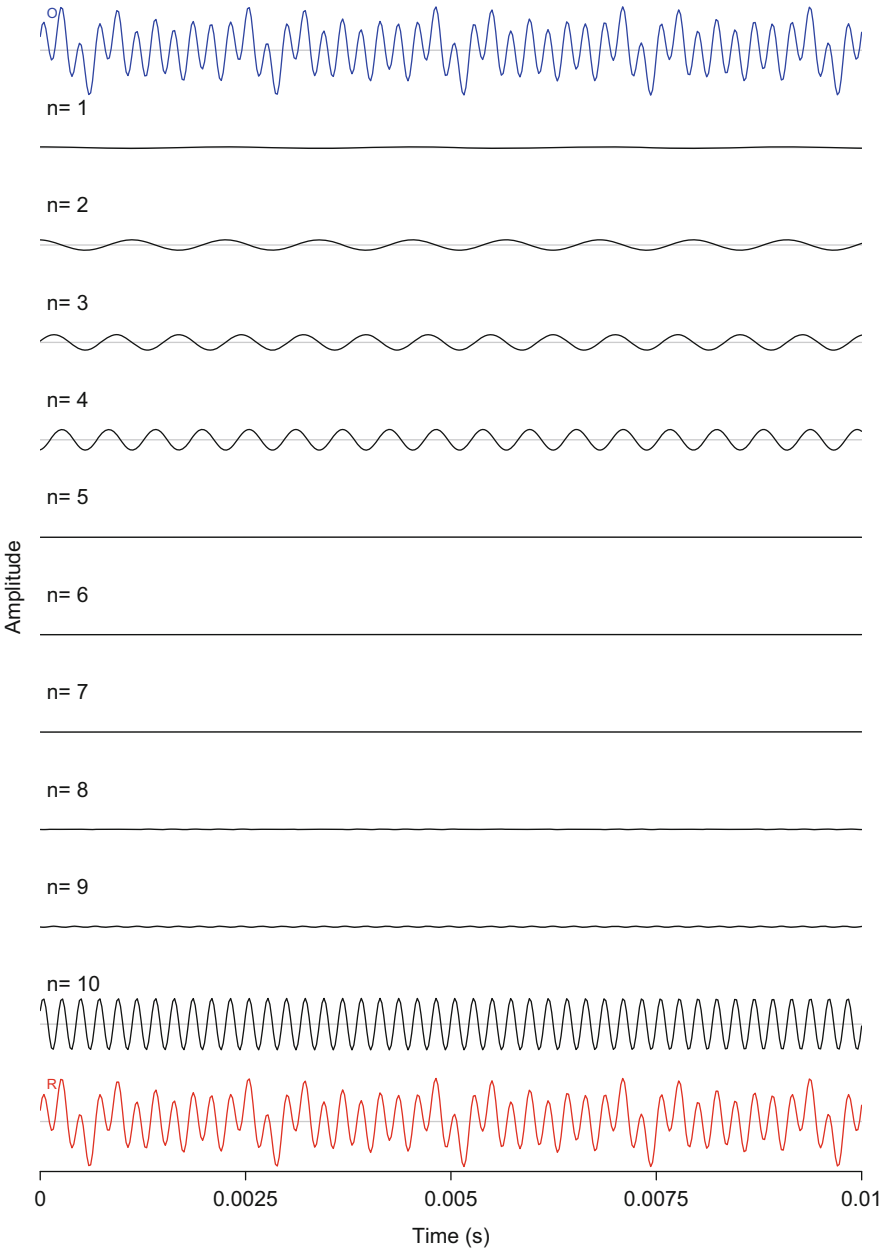


Fig. 9.4 Frequency decomposition and signal reconstruction. The original signal (O) is decomposed into a series of ten functions written as $[A_n \cos(\omega_n t) + B_n \sin(\omega_n t)]$ with $n = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. The last signal (R) is the reconstruction of the original signal (O) using the coefficients A_n and B_n and the angular frequencies ω_n

and φ_n in rad with:

$$\varphi_n = \tan^{-1}(B_n/A_n)$$

These new coefficients are easy to obtain. First the C_n coefficients:

```
C <- sqrt(A^2 + B^2)
C
 [1]  0.29497901  2.01045814  2.98819655  4.00392798
 [5]  0.04592934  0.04252931  0.05955912  0.09815747
 [9]  0.21598397 10.00671597
```

and then φ_n:

```
Phi <- atan(B/A)
Phi
 [1] -0.026059958  0.001309576 -1.561783883  0.028252904
 [5]  0.438390038  0.974051388  1.277216809  1.448149174
 [9]  1.560308390 -1.497873889
```

The coefficients C_n can be considered as the weights of each harmonic angular frequency ω_n. It could be a good idea to visualize them as a barplot. This graph constitutes the well-known **frequency spectrum** which is the visual output of a transformation from the time to the frequency domain (Fig. 9.5):

```
plot(C, type="h", lwd=2, las=1, col="blue",
     xlab="Index (n) of the angular frequency of
      successive Fourier harmonics",
     ylab=expression(paste("Amplitude ", C[n])))
points(C, col="blue")
abline(h=0, col="grey")
axis(side=3, at=1:10, labels=as.character(440*(1:10)))
mtext("Frequency (Hz)", line=3)
```

Similarly, the **phase spectrum**, more commonly used in image than in sound analysis, can be produced with the following code (Fig. 9.6):

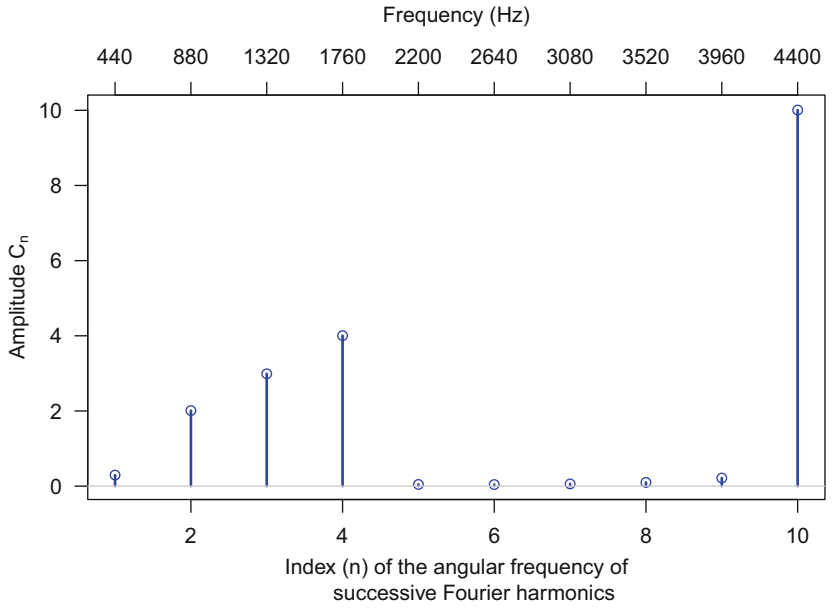


Fig. 9.5 Frequency spectrum. The frequency spectrum is a barplot of the Fourier coefficients C_n against the n angular frequency indices. The top frequency scale in Hz was manually added with the graphical function `axis()` and `mtext()`

```
plot(Phi, type="h", lwd=2, col="blue", las=1,
     xlab="Frequency index (n)",
     ylab=expression(paste("Phase ", varphi[n], " (rad)")),
     ylim=c(-pi, pi))
points(Phi, col="blue")
abline(h=0, col="grey")
```

9.2.4 *Exponential Fourier Series*

Another way to write the Fourier series is to involve complex numbers and more specifically their exponential form with, this time, limits between $-\infty$ and $+\infty$:

$$s(t) = \sum_{-\infty}^{\infty} D_n e^{i\omega_n t}$$

where $i^2 = -1$, $e^{ix} = \cos x + i \sin x$ (Euler's formula), and e is the base of the natural logarithm.

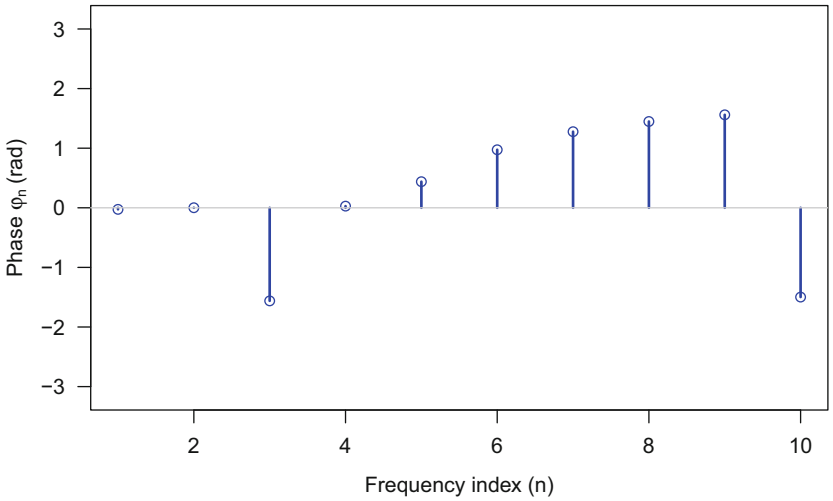


Fig. 9.6 Phase spectrum. The phase spectrum is a barplot of the phase coefficients φ_n against the n angular frequencies

The Fourier coefficients, now named D_n, are obtained with:

$$D_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} s(t) e^{-i\omega_n t} \, \mathrm{dt}$$

We can compute the indices D_n by writing the following new function and applying it directly to the first ten harmonics:

```
coeffD <- function(n) {
   wnt <- n*2*pi/Tsamp*dt
   res <- 1/Tsamp*sum(s[dt]*exp(complex(imaginary=-wnt)))
   return(res)
  }
D <- sapply(1:10, FUN=coeffD)
D
 [1]  0.14743943+0.00384314i  1.00522821-0.00131642i
 [3] -0.01346529-1.49403760i -2.00116503+0.05655377i
 [5] -0.02079304+0.00974810i -0.01194973+0.01758947i
 [7] -0.00861762+0.02850542i -0.00600429+0.04871007i
 [9] -0.00113259+0.10798605i -0.36453378-4.99006074i
```

Plotting the modulus of `D` returns a similar plot to the one shown in Fig. 9.5:

```
plot(Mod(D), type="h", col="blue")
```

9.3 Fourier Transform

The next step in the Fourier transformation is the Fourier transform (FT) which is an extension of the Fourier series (FS) to periodic and aperiodic signals. The Fourier transform is therefore a more general formulation as it can describe an important variety of time functions. The Fourier transform is commonly used in sound analysis to produce the frequency spectrum, less often the phase spectrum. The Fourier transform is therefore a bridge built between the time and frequency domains.

The continuous FT of a function of time, here $s(t)$, is a function of angular frequency ω. We therefore switch from time t to frequency ω as shown in the following general equation with a function of frequency ω on the left and a function of time t on the right:

$$F(\omega) = \int_{-\infty}^{\infty} s(t)e^{-i\omega t}\,\mathrm{dt}$$

The frequency spectrum is obtained by computing the modulus of the Fourier transform $|F(\omega)|$ and the phase spectrum by calculating the argument of the Fourier transform $\arg F(\omega)$.

How can we process the Fourier transform of a digitized time-limited sound? As we have seen in Sect. 2.4, the process of digital recording is based on regular sampling—sound pressure variations are measured following a specified sampling frequency f_s—and this during a limited time interval, the recording starts and stops at specific time. This means that a digitized sound is a discrete and finite object. Such mathematical object is treated with a specific kind of Fourier transform, the so-called discrete Fourier transform (DFT). The DFT can produce a discrete frequency spectrum (sampled spectrum) from a discrete time signal (sampled signal) over a limited time interval.

The general Fourier transform equation can be written for a finite sound made of N samples taking successive values $s[n]$ ($s[10]$ being, for instance, the 10th sample):

$$F[\omega] = \sum_{n=0}^{\infty} s[n]e^{-i\omega_k n}$$

Knowing that we should obtain $F(\omega)$ values for $\omega_k = \frac{2\pi}{N}k$, with $k = \{0, 1, 2, \ldots, (N-1)\}$, the discrete Fourier transform (DFT) can be written as a function of k with a sum limited between 0 and $N-1$:

$$F[k] = \sum_{n=0}^{N-1} s[n] e^{-i\frac{2\pi}{N}kn}$$

The computational complexity of the DFT is very important and can be dramatically time-consuming. Using data parallelization, the fast Fourier transform (FFT) can reduce considerably the computing time of the DFT. The FFT is not another kind of Fourier transformation but a tool in mathematics, or algorithm in computer sciences, that can process the DFT very efficiently. Coined by Cooley and Tukey (1965), the FFT decreases the number of operations to compute the DFT from N^2 to $N \times \log_2(N)$ for a sound made of N samples. This reduction is significant, a 1 s sound sampled at 44,100 Hz would require $44{,}100^2 = 1{,}944{,}810{,}000$ operations without the FFT and only $44{,}100 \times \log_2(44{,}100) = 680{,}397$ with the FFT, that is, a reduction of $N \div \log_2(N) = 44{,}100 \div \log_2(44{,}100) = 2858$.

In `R`, the FFT is implemented in the function `fft()`. The use of this function is rather simple; we just need to provide the signal as an input:

```
fft <- fft(s)
```

The object returned by `fft()` is a vector made of 44,100 complex numbers:

```
is.vector(fft)
[1] TRUE
class(fft)
[1] "complex"
length(fft)
[1] 44100
```

The input and output of `fft()` have always the same size. Here the original signal $s[n]$ was a succession of 44,100 samples leading to a FFT of similar length. To produce the discrete frequency spectrum, we have to compute the complex modulus of the FFT, directly obtained with the function `Mod()`:

```
fspec <- Mod(fft)
```

The class and the length of the object `fspec` are the same as those of the object `fft`. However `fspec` does not contain complex numbers anymore but numeric numbers:

```
is.vector(fspec)
[1] TRUE
class(fspec)
[1] "numeric"
length(fspec)
[1] 44100
```

The FFT transformed the N time samples into N frequency samples. The values of the original signal and of the frequency spectrum may appear to vary over different ranges:

```
range(s)
[1] -17.45387  16.88494
range(fspec)
[1] 1.860777e-12 2.205000e+05
```

However the total energy of the time and frequency signals are preserved as stated by Parseval's theorem which stipulates that the sum of the square of a function is equal to the sum of the square of its transform. Applied to the Fourier transform, Parseval's theorem can be written as:

$$\sum_{n=0}^{N-1} s[n]^2 = \frac{1}{N} \sum_{k=0}^{N-1} |F[k]|^2$$

We can check the equality here with:

```
sum(s^2)
[1] 2857459
N <- length(fspec)
1/N*sum(fspec^2)
[1] 2857459
```

The frequency spectrum can be visualized by calling `plot()` with `type="h"` to draw a vertical line corresponding to the amplitude of each angular frequency. The frequency resolution is here 1 Hz because the sound analyzed lasts 1 s. The

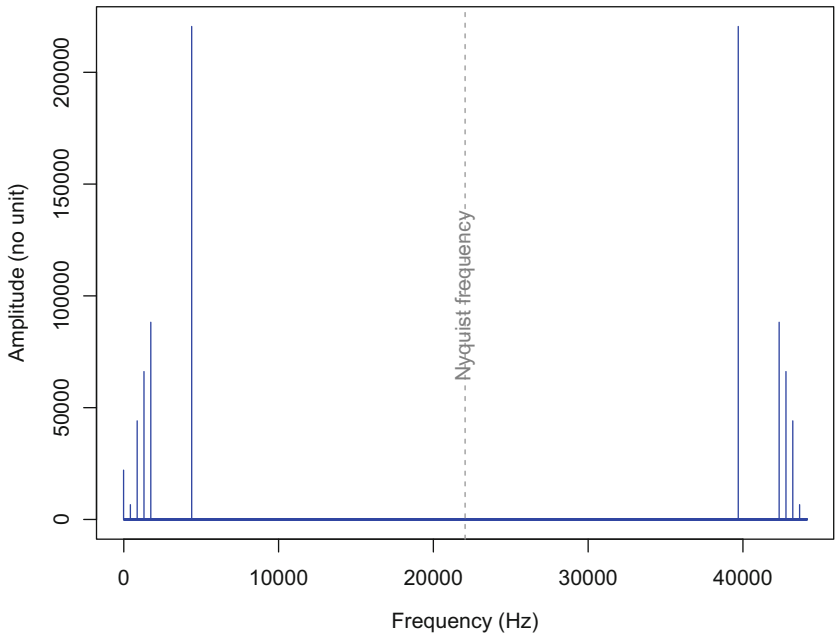


Fig. 9.7 Mirrored frequency spectrum of the FFT. The modulus of the FFT is a symmetric (mirrored) function of the angular (or regular) frequency around the Nyquist frequency f_N

amplitude varies along a scale without unit (Fig. 9.7):

```
plot(fspec, type="h", col="blue",
     xlab="Frequency (Hz)",
     ylab="Amplitude (no unit)")
```

Figure 9.7 shows that the modulus of the Fourier transform is a symmetric, or mirrored, function around half the sampling frequency or Nyquist frequency f_N, here $f_N = f_s \div 2 = 22{,}050\,\text{Hz}$. This symmetry is one of the main properties of the Fourier transform due to the symmetric properties of the cosine and sine functions.

We zoom in on the left half of the spectrum, and we multiply the amplitude values by 2 to keep the total amount of energy (Fig. 9.8):

```
fspec.left <- 2*fspec[1:(f/2)]
length(fspec.left)
[1] 22050
plot(fspec.left, type="h", las=1, col="blue",
     xlab="Frequency (Hz)",
     ylab="Amplitude (no unit)")
```

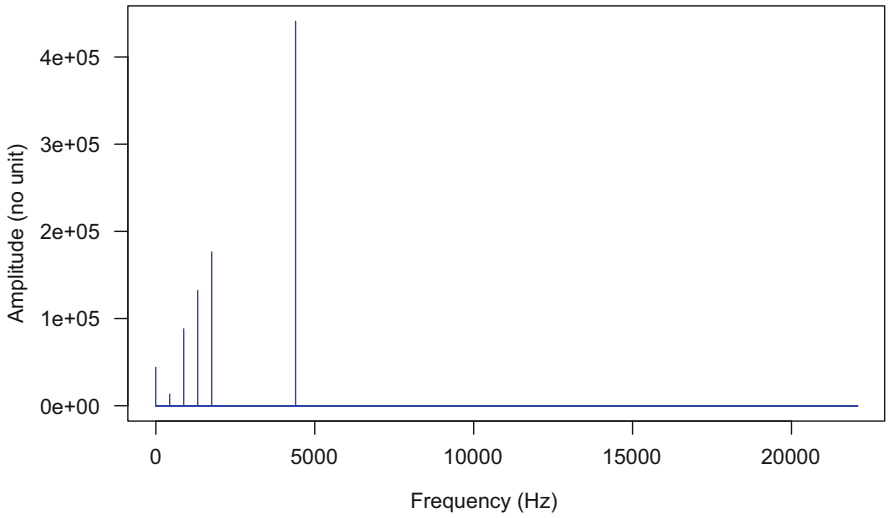


Fig. 9.8 Frequency spectrum of the FFT. This spectrum includes all the Fourier coefficients from C_0 to the Nyquist frequency f_N

We then look more precisely to the peaks of the spectrum and identify them with the function `identify()`:

```
peaks <- identify(fspec.left)
```

The peaks are found at the following index values, that is, the positions along the vector `fspec.left`:

```
peaks
[1]    1  441  881 1321 1761 4401
```

To convert these values in Hz, we just need to subtract 1, the first value corresponding to C_0 or 0 Hz frequency:

```
peaks <- peaks-1
peaks
[1]    0  440  880 1320 1760 4400
```

Do these frequencies match with the frequencies we used to build the signal $s[n]$? Remember that the original signal obeys to the following equation, here in its continuous form, with a fundamental frequency of 440 Hz:

$$s(t) = 0.5 + 0.3\cos(t) + 2\sin(2t) + 3\sin(3t) - 4\cos(4t) + \sin(10t)$$

so that it is made of the following harmonics:

```
h <- c(0,1,2,3,4,10)  # harmonics numbers
freq <- 440*h         # linear frequency
freq
[1]    0  440  880 1320 1760 4400
```

The FFT did properly the frequency decomposition and found the right harmonics.

9.4 Frequency Scales

9.4.1 Bark and Mel Scales

We have already seen that the radian is the unit of angular frequency and the Hertz the unit of regular frequency. These two units vary along a linear scale. However, the perception of frequency by humans is nonlinear due to nonlinear neural and psychological auditory mechanisms. The basilar membrane of the human cochlea discriminates frequency as a bank of frequency filters would do. However, these set of filters are not symmetric and do not have all the same width. The system is therefore not totally linear and motivated the definition of 24 critical bands along the cochlea corresponding more or less to 24 different filters (Zwicker 1961). The Bark scale is defined so that one unit (1 Bark) corresponds to the width of one critical band. The Bark scale is therefore limited between 1 and 24 (Fig. 9.9). A mathematical way to convert Hz and Bark is to apply the following equation:

$$f_{\text{Hz}} = 600 \times \sinh\left(\frac{f_{\text{bark}}}{6}\right)$$

where sinh is the hyperbolic sine function.

The inverse equation is:

$$f_{\text{bark}} = 6 \times \sinh^{-1}\left(\frac{f_{\text{Hz}}}{600}\right)$$

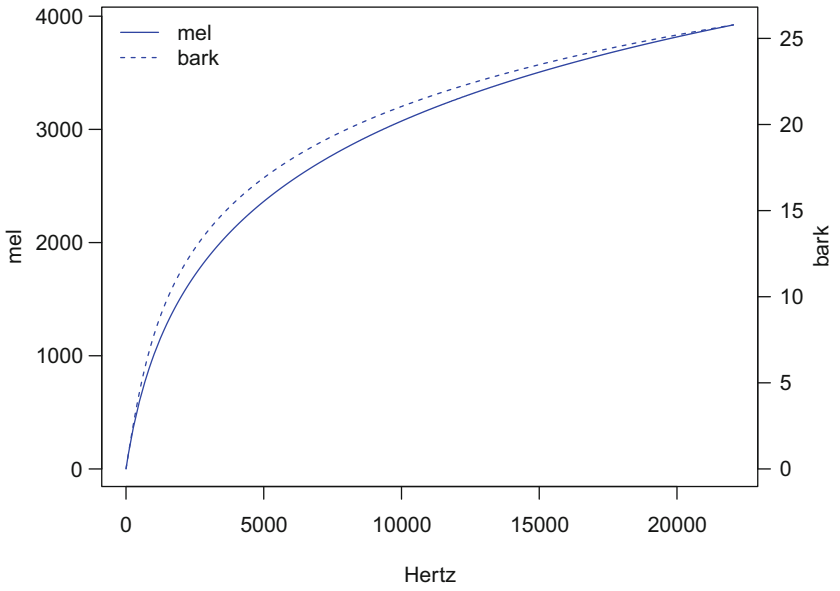


Fig. 9.9 Hertz (*x*-axis), mel (left *y*-axis), and Bark (right *y*-axis) scales. Bark and mel scale are closely related even if defined differently and evolving on different ranges

The functions `bark2hz()` and `hz2bark()` of the package `tuneR` perform this two-way conversion:

```
bark2hz(1:24) # the 24 Bark units converted to Hertz
 [1]   100.4636   203.7243   312.6572   430.2951   559.9133
 [6]   705.1207   869.9602  1059.0212  1277.5677  1531.6843
[11]  1828.4464  2176.1162  2584.3739  3064.5860  3630.1227
[16]  4296.7298  5082.9670  6010.7250  7105.8343  8398.7853
[21]  9925.5764 11728.7167 13858.4097 16373.9503
hz2bark(440)  # 440 Hz converted into Bark
[1] 4.078563
```

The mel(ody) scale is a subjective and logarithmic frequency scale established on the results obtained with human perception tests related to frequency and loudness. There are actually several mel scales that can differ significantly. Historically, the first, and most used, mel scale was due to Stevens et al. (1937) (Fig. 9.9). Mathematical equations to convert Hz into mel were provided later. One of these equations is:

$$f_{mel} = 1127.01048 \times \log\left(1 + \left(\frac{f_{\text{Hz}}}{700}\right)\right)$$

with the inverse equation:

$$f_{\text{Hz}} = 700 \times \left(e^{\frac{f_{mel}}{1127.01048}} - 1\right)$$

These formulae are coded in the function `mel()` of `seewave`:

```
mel(440)              # 440 Hz converted into mel
[1] 549.6466
mel(10, inverse=TRUE) # 10 mel converted into Hertz
[1] 6.23876
```

A similar conversion can be processed with the `tuneR` functions `hz2mel()` and `mel2hz()`. In this case the argument `htk`, which refers to the `C` program Hidden Markov Model Toolkit (HTK)[3], should be set to `TRUE`:

```
hz2mel(440, htk=TRUE) # 440 Hz converted into mel
[1] 549.6387
mel2hz(10, htk=TRUE)  # 10 mel converted into Hertz
[1] 6.23885
```

9.4.2 Musical Scale

Western music uses its own frequency scale based on note intervals. An octave is an interval corresponding to a 2:1 ratio, such that a note vibrating at a frequency f_1 and a second note vibrating at $f_2 = 2 \times f_1$ are separated by one octave. The function `octaves()` of `seewave` returns the frequency values of the octaves below and above a specific frequency. For instance, the following call of `octaves()` provides the different frequencies of a A note over seven octaves:

```
octaves(440)
[1]   55  110  220  440  880 1760 3520
```

Tones are frequency intervals defining the frequency position of Western musical notes. There are 12 tones related to 7 notes (Latin letters C, D, E, F, G, A, B in English and Dutch notation) that can be raised (accidental ♯) or lowered (accidental

[3] http://htk.eng.cam.ac.uk/

♭) by halftones giving 5 additional notes (C♯ = D♭, D♯ = E♭, F♯ = G♭, G♯ = A♭, A♯ = B♭). Each Western note has a peculiar frequency, usually expressed in Hz. The conversion from a note to its corresponding frequency can be achieved with the function `notefreq()` of `seewave`, here for the A note:

```
notefreq("A")
[1] 440
```

This result can be extended over several octaves with the argument `octaves`, here for the octaves 1 to 3:

```
notefreq("A", octave=1:3)
[1] 110 220 440
```

We can therefore obtain quite simply the frequency of the 12 notes over the first 6 octaves with the following code (Fig. 9.10):

```
notes <- c("C","C#","D","D#","E","F","F#","G","G#","A","A#","B")
freq <- sapply(X=notes, FUN=notefreq, octave=1:6)
freq
              C         C#          D         D#          E
[1,]   65.40639   69.29566   73.41619   77.78175   82.40689
[2,]  130.81278  138.59132  146.83238  155.56349  164.81378
[3,]  261.62557  277.18263  293.66477  311.12698  329.62756
[4,]  523.25113  554.36526  587.32954  622.25397  659.25511
[5,] 1046.50226 1108.73052 1174.65907 1244.50793 1318.51023
[6,] 2093.00452 2217.46105 2349.31814 2489.01587 2637.02046
              F         F#          G         G#    A
[1,]   87.30706   92.49861   97.99886  103.8262  110
[2,]  174.61412  184.99721  195.99772  207.6523  220
[3,]  349.22823  369.99442  391.99544  415.3047  440
[4,]  698.45646  739.98885  783.99087  830.6094  880
[5,] 1396.91293 1479.97769 1567.98174 1661.2188 1760
[6,] 2793.82585 2959.95538 3135.96349 3322.4376 3520
            A#         B
[1,]  116.5409  123.4708
[2,]  233.0819  246.9417
[3,]  466.1638  493.8833
[4,]  932.3275  987.7666
[5,] 1864.6550 1975.5332
[6,] 3729.3101 3951.0664
```

The inverse action, that is, the identification of a note given a frequency, can be processed with the function `noteFromFF()` of `tuneR`. The function returns the difference in position of the tone in reference to the A note, that is, the A of the third octave also written a':

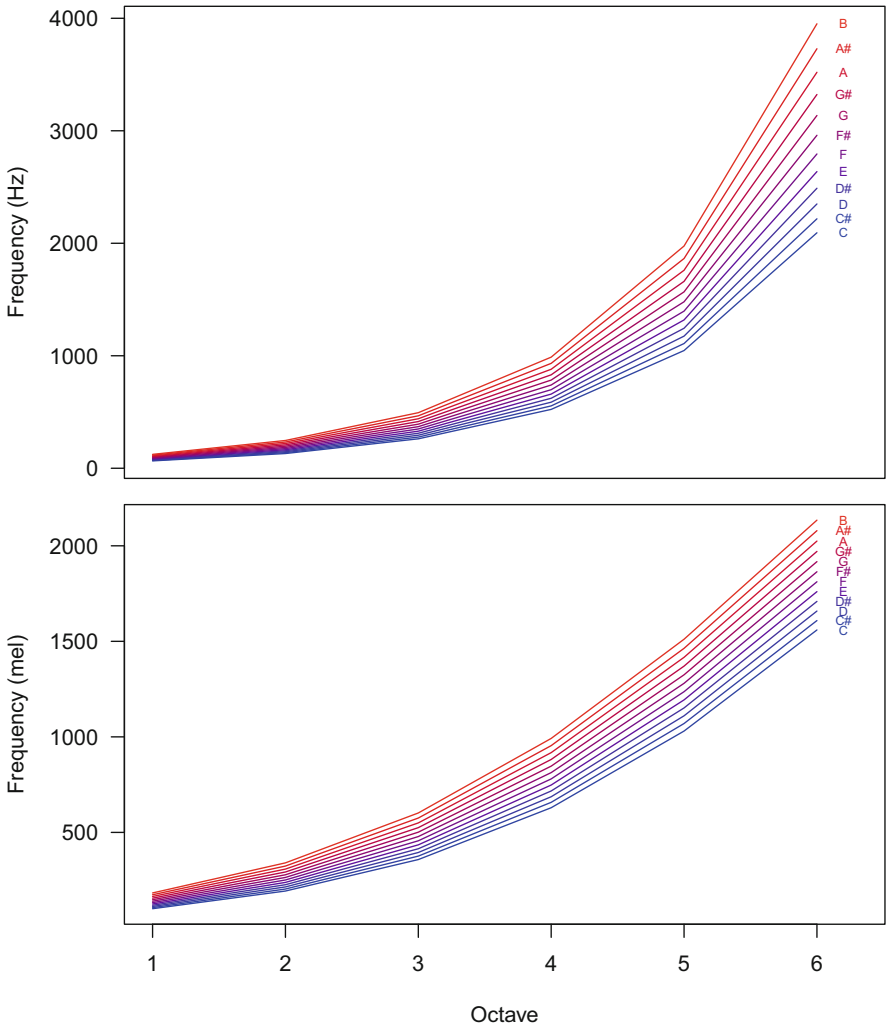


Fig. 9.10 Frequency of Western musical notes. The frequency in Hertz and mel of the 12 Western musical notes is plotted over the first 6 octaves. The mel scale, through its logarithm properties, spaces more equally the notes than the Hertz scale along the octaves

```
noteFromFF(440) # returns  0 = reference = A third octave
[1] 0
noteFromFF(493) # returns  2 = 2nd note  = B third octave
[1] 2
noteFromFF(622) # returns  6 = 6th note  = D# third octave
[1] 6
```

(continued)

```
noteFromFF(880) # returns 12 = 12th note = A fourth octave
[1] 12
```

The function `notenames()` of `tuneR` can return the name of notes according to this numeric difference:

```
notenames(0)   # A third octave
[1] "a'"
notenames(2)   # B third octave
[1] "b'"
notenames(6)   # D third octave
[1] "d#''"
notenames(12)  # A fourth octave
[1] "a''"
```

We can of course combine `notenames()` and `notefromFF()` to derive the note names directly from a frequency:

```
notenames(noteFromFF(440)) # A third octave
[1] "a'"
```

The package `soundgen` proposes also a pair of functions, `HzToSemitones()` and `semitonesToHz()`, that convert a frequency in Hz into a halftone position and vice versa along a full scale of musical notes, that is, a scale starting with a C note at 16.4 Hz in the infrasound domain and ending with a B note at 31,608.5 Hz in the ultrasound domain.

```
HzToSemitones(c(440, 493, 622, 880))
[1] 56.9999977038643 58.9690071781033 62.9929303831356
[4] 68.9999977038643
```

This value can be used with the accompanying dataset `notesDict` which is a `data.frame` making the link between position and note:

```
head(notesDict)
  note freq
```

(continued)

```
1   C0 16.4
2  C0 17.3
3   D0 18.4
4  D0 19.4
5   E0 20.6
6   F0 21.8
tail(notesDict)
    note    freq
127 F10 23679.6
128  G10 25087.7
129 G10 26579.5
130  A10 28160.0
131 B10 29834.5
132  B10 31608.5
```

The function and the dataset can also be combined to get the notes corresponding to a specific frequency:

```
notesDict[1+round(HzToSemitones(c(440, 493, 622, 880))),1]
[1] "A4"  "B4"  "D5" "A5"
```

9.5 Amplitude Scales

The FT attributes to each frequency an amplitude coefficient. All the possible options of expressing amplitude in the time domain can be transferred to the frequency domain. Therefore, and as detailed in Chap. 2, the frequency amplitudes can be expressed along an absolute scale related to the measure of medium pressure variations or along a relative scale without any calibrated reference. The amplitude values can follow a linear or a logarithm scale, typically a dB scale.

On a linear scale, the raw values obtained by the FFT can be kept, or they can be scaled in three different ways: (1) division of the mirrored FFT by the length of the FFT (N), (2) division by the maximum of the FFT to get values scaled between 0 and 1 such that spectra deriving from sounds recorded with different recording levels could be contrasted, and (3) division by the sum of the FFT values so that the resulting frequency spectrum can be regarded as a probability mass function (PMF) (Fig. 9.11).

On a dB scale, the values can be scaled to vary between a maximum of 0 and negative values such that the amplitude scale is a relative and comparable scale. If the recording chain is fully calibrated (see Sect. 7.3), then the spectrum can be expressed with absolute dB values. The dB weightings (dB(A), dB(B), dB(C), and dB(D)) can also be used either with relative or absolute scale (Fig. 9.11).

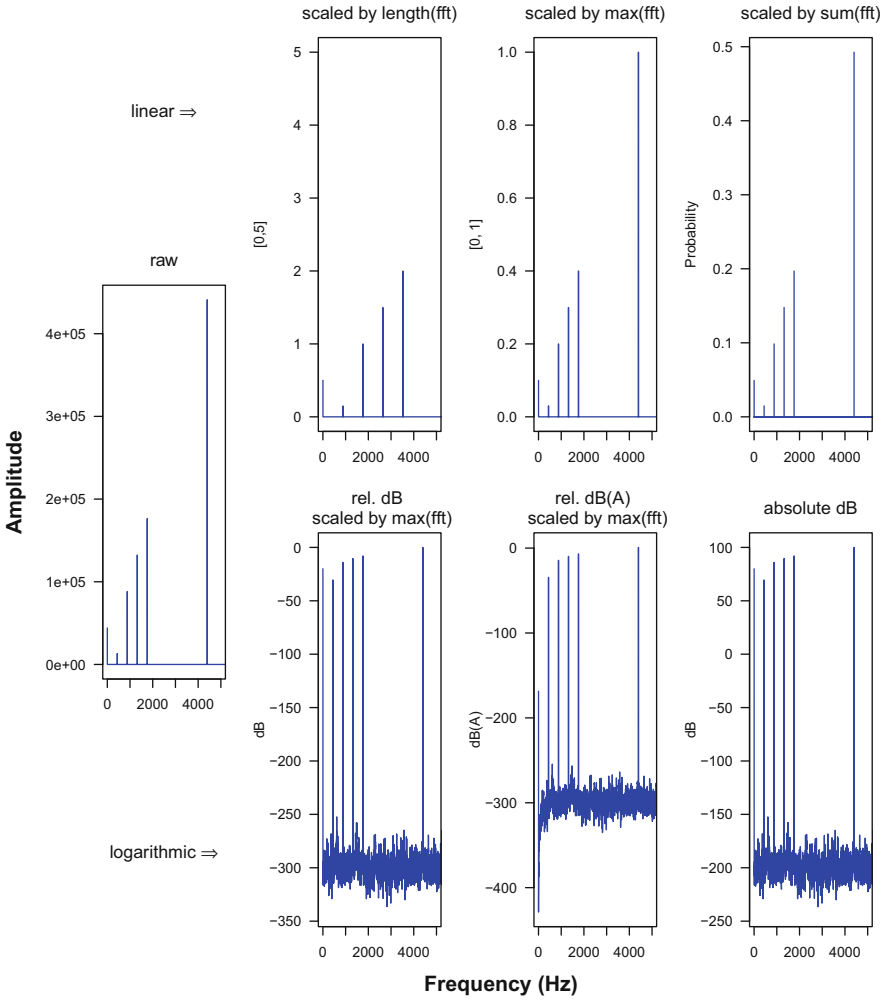


Fig. 9.11 Amplitude scale of the frequency spectrum. Seven examples of amplitude scales used to show a frequency spectrum, from raw data directly returned by the FFT to linear and scaled scales and logarithmic scales based on the dB unit. A zoom between 0 and 5000 Hz was operated on the frequency axis

9.6 Fourier Windows

The computation of the DFT through the FFT is operated over a limited number of samples. The DFT and of course the FFT make the tacit assumption that data constitutes a periodic time series, that is, the data are repeated over and over again. However, the signal of interest is not always periodic, and this induces artifacts in the frequency spectrum, as changes in amplitude, modifications in the overall shape,

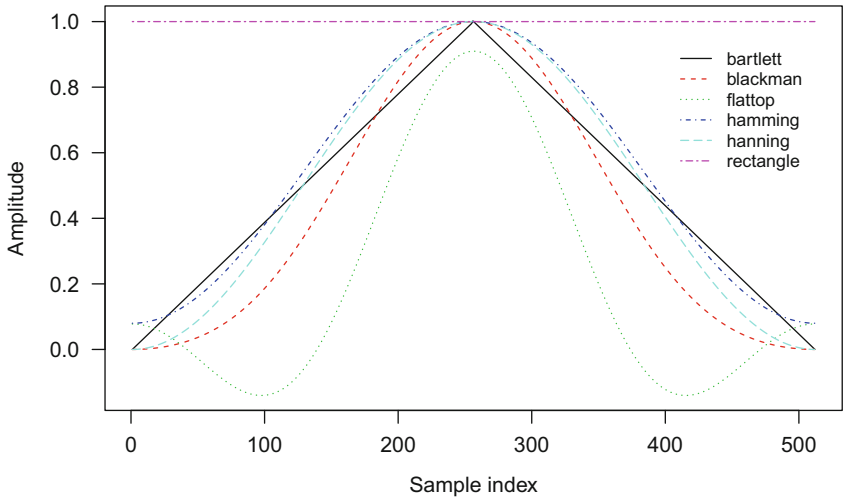


Fig. 9.12 FFT window shape. Shapes of the six FFT windows implemented in `seewave`. The windows includes here $N = 512$ samples

and generation of undesirable sine lobes. This phenomenon, known as spectral leakage, can be reduced—but not eliminated—by multiplying the time series $s[n]$ by a window function $w[n]$ or taper:

$$s_w[n] = s[n] \times w[n]$$

There is an important choice of window functions $w[n]$, but most of them have a bell shape decreasing smoothly to zero or near zero at the beginning and end (Fig. 9.12). The shape of six windows can be viewed with the function `ftwindow` of `seewave`. For instance, the following code plots the Hanning window for $N = 512$ samples:

```
plot(ftwindow(wl=512, wn="hanning"), type="l", col="blue",
     xlab="Sample", ylab="Amplitude")
```

Follow the equations of the most popular windows for N samples. The rectangular or Dirichlet window is the simplest window basically keeping the signal intact and considering the rest of data as zero values:

$$w_{\texttt{rectangle}}[n] = \begin{cases} 1 & \text{if } n \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

The Bartlett window is a triangular window:

$$w_{\texttt{bartlett}}[n] = \begin{cases} \frac{2n}{N} & \text{if } n \in \left[0, \frac{N}{2}\right[\\ \frac{2(N-n)}{N} & \text{if } n \in \left[\frac{N}{2}, N\right] \\ 0 & \text{elsewhere} \end{cases}$$

The Blackman window is a cosine function:

$$w_{\texttt{blackman}}[n] = \begin{cases} 0.42 - 0.5 \times \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \times \cos\left(\frac{4\pi n}{N-1}\right) & \text{if } n \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

The flattop window has negative values:

$$w_{\texttt{flattop}}[n] = \begin{cases} 0.2156 - 0.416 \times \cos\left(\frac{2\pi n}{N-1}\right) + 0.2781 \times \cos\left(\frac{4\pi n}{N-1}\right) \\ -0.0836 \times \cos\left(\frac{6\pi n}{N-1}\right) + 0.0069 \times \cos\left(\frac{8\pi n}{N-1}\right) & \text{if } n \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

The Hanning, or Hann window, is a raised cosine function:

$$w_{\texttt{hanning}}[n] = \begin{cases} 0.5 - 0.5 \times \cos\left(\frac{2\pi n}{N-1}\right) & \text{if } n \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

The Hamming window is closely related to the Hanning (Hann) window but with slightly modified coefficients:

$$w_{\texttt{hamming}}[n] = \begin{cases} 0.54 - 0.46 \times \cos\left(\frac{2\pi n}{N-1}\right) & \text{if } n \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

As illustrated in Fig. 9.13, the effects on the frequency spectrum of these window functions can differ markedly in particular when using a dB scale so that their selection should be considered carefully in regard with the nature of the signal (sinusoidal or not), the frequency resolution, the reduction of the side lobes, and the amplitude accuracy. It seems that in most cases, at least for continuous animal or environmental sounds, the Hanning (Hann) window is the best choice so that it has been selected as the default window for most `seewave` spectral functions.

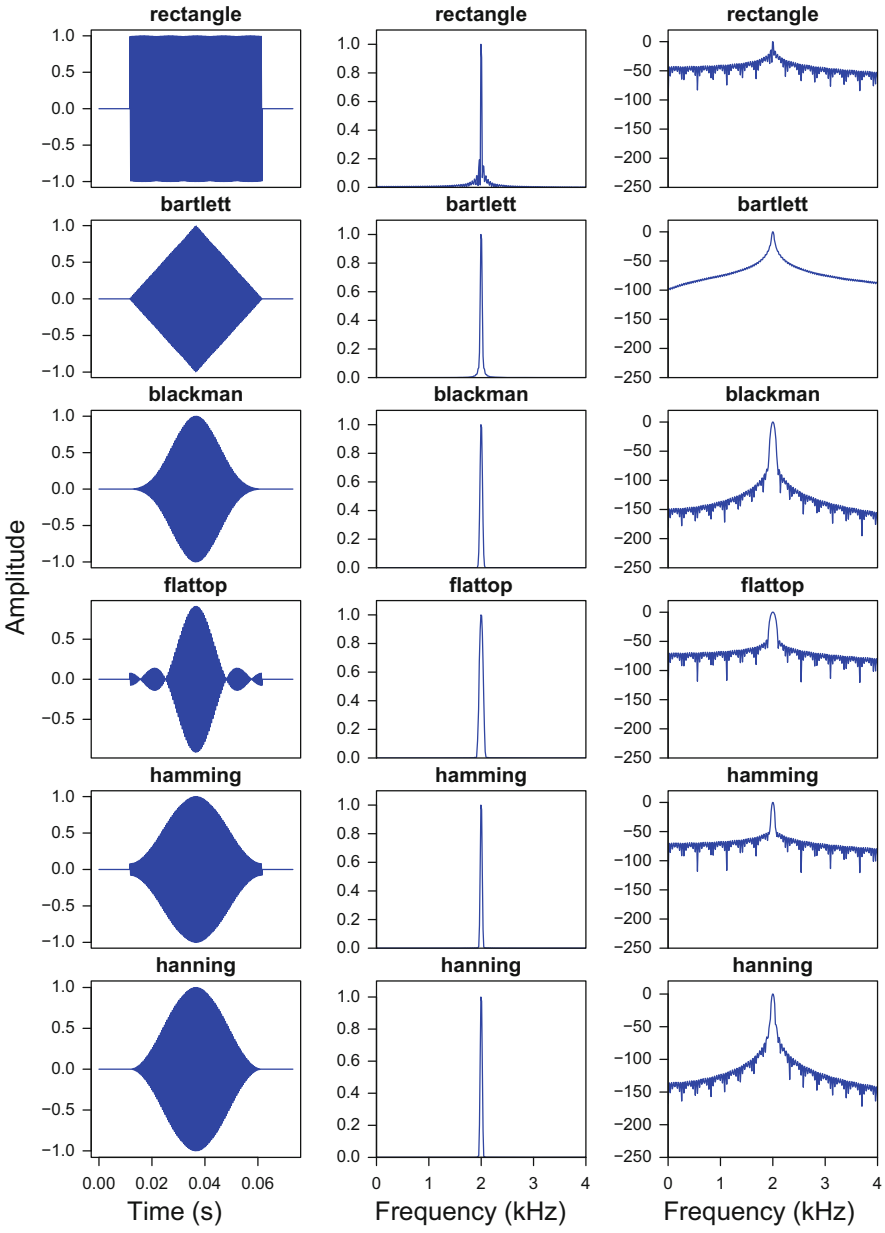


Fig. 9.13 FFT window effects on the frequency spectrum. A 2 kHz sound lasting 0.02 s is windowed (syn. tapered) with a function that reduces errors in the frequency spectrum. Basically the rectangular window (first line of graphics) has no effect when the remaining (Bartlett, Blackman, flattop, Hamming, and Hanning) changes the shape of the spectrum. The effects are less visible on a spectrum with a linear amplitude scale (second column) than on a spectrum with a dB amplitude scale (third column)

9.7 Inverse Fourier Transform

The inverse Fourier transform is the way to travel back from the frequency domain to the time domain. The inverse Fourier transform is therefore a way to recover the original time signal $s(t)$ from its transformation into the frequency domain. Changes of the frequency domain, as filtering, can be applied before to process the inverse transform such that the original time signal can be modified using frequency parameters. This opens important possibilities for sound design (see Sect. 15.4).

Each member of the transformation Fourier family (Table 9.1) has a corresponding inverse member: the inverse Fourier series (IFS), the inverse Fourier transform (IFT), the inverse discrete Fourier transform (IDFT), the inverse fast Fourier transform (IFFT), the inverse short-time Fourier transform (ISTFT), and the inverse short-time discrete Fourier transform (ISTDFT).

The inverse Fourier transform (IFT) for an infinite signal is:

$$s(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} \mathrm{d}\omega$$

The discrete form of the inverse Fourier transform (IDFT), that is the inverse Fourier transform of a time-limited signal, is written according to:

$$s[n] = \frac{1}{N} \sum_{k=0}^{N-1} F[k] e^{i \frac{2\pi}{N} kn}$$

In `R`, the inverse Fourier transform is computed by using the argument `inverse` of the function `fft()`. Remembering that:

```
fft <- fft(s)
```

The inverse Fourier transform is computed with:

```
fft(fft, inverse=TRUE)
```

To recover the original signal $s(t)$, we need to scale by the length of the FFT and keep only the real part of the inverse Fourier transform:

```
s.recovered <- Re(fft(fft, inverse=TRUE)/length(fft))
```

We check that the differences between the original and the recovered signals are negligible:

```
sum(s-s.recovered)
[1] -1.540001e-15
```

9.8 Cepstrum

The *cepstrum*, an anagram of *spectrum* that should be pronounced ['kɛpstɹəm], is an innovation of Bogert et al. (1963) to identify and potentially remove an echo appearing after a delay τ in a time signal.[4] The first definition given by Bogert et al. (1963) is "the power spectrum of the logarithm of the power spectrum." In a less concise style, the cepstrum can be described as a three-step process: (1) computation of the square of the Fourier frequency spectrum, (2) computation of the logarithm of this spectrum, and (3) computation of the square of the Fourier frequency spectrum of the log spectrum. The cepstrum function $C(\tau)$ can be therefore written as:

$$C(\tau) = |F(\log(|F(\omega)|^2))|^2$$

This would be translated in R with:

```
Mod(fft(log(Mod(fft(s))^2)))^2
```

However, this transformation was not reversible to the time domain limiting the action of lifters (i.e. filters) to remove, for instance, an echo. A few years later, Oppenheim and Schafer (1975) defined a new type of cepstrum, named the complex cepstrum, by using the inverse of the Fourier transform. The "complex cepstrum" is the inverse Fourier transform of the logarithm of the complex spectrum and can be written as:

$$C(\tau) = F^{-1}(\log(|F(\omega)|))$$

which would give in R:

```
fft(log(abs(fft(s))), inverse=TRUE)
```

[4] See Oppenheim and Schafer (2004) for a brilliant story of the cepstrum.

Table 9.2 Spectral-cepstral dictionary

Spectrum language	Cepstrum language
Frequency	Quefrency
Period	Repiod
Harmonic	Rahmonic
Phase	Saphe
Filter	Lifter
Magnitude	Gamnitude

Bogert et al. (1963) played with letters to create a terminology associated with their new "spectral" transformation

This transform has the great advantage to be reversible to the time domain. Changes can be applied on the cepstrum and applied back to the time series.

The domain of the cepstral transformation is neither conventional time nor conventional frequency; it was therefore named *quefrency*, an anagram of *frequency*. Actually, Bogert et al. (1963) coined several cepstral terms all based on puzzling anagrams (Table 9.2).

We have seen that the independent variable of a cepstrum is the quefrency τ. The quefrency is a measure of time, though not in the sense of a signal in the time domain. A correspondence with the frequency domain is, however, obtained by simply computing the inverse of τ. For instance, if a cepstral peak appears at 0.005, this reveals a frequency peak at ($\tau = 1 \div 0.005 = 200$) that can be converted in 200 Hz in the classical spectral domain.

Figure 9.14 shows a basic use of the cepstrum for a signal with an echo. The delay of the echo is seen as a peak on the cepstrum.

The cepstrum is actually used in many more applications than the sole detection of echoes. The cepstrum has been repeatedly employed in industry, communications, seismology, speech analysis, and life sciences. In particular, the cepstral transform can help in determining the fundamental frequency of a harmonic series (see Sect. 10.1.3.4). An echo with a delay τ in s generates an amplitude modulation with a frequency $f_{\text{am}} = 1 \div \tau$ in Hz that can be viewed as the lowest frequency or fundamental frequency of a harmonic series. By definition, the frequency spectrum of a harmonic series shows frequency lobes regularly. For instance, the frequency spectrum of a harmonic series with a fundamental frequency $f_0 = 440$ Hz will show regular lobes spaced by 440 Hz. The cepstral transform first compresses the amplitude range of the frequency lobes by computing the logarithm of the frequency spectrum and, second, considers this log spectrum as a time series with a regular amplitude variation that can be detected with a FFT and that corresponds to the fundamental frequency, in our example 440 Hz (Fig. 9.15).

Similarly, the cepstrum can be used to detect the frequency of an amplitude modulation. Indeed, the frequency spectrum of a signal with a regular amplitude modulation shows frequency lobes or sidebands spaced by the frequency of the amplitude modulation (see Sect. 10.1.4.2). For instance, the frequency spectrum of a sine wave with a 100 Hz amplitude modulation will show regular lobes spaced by

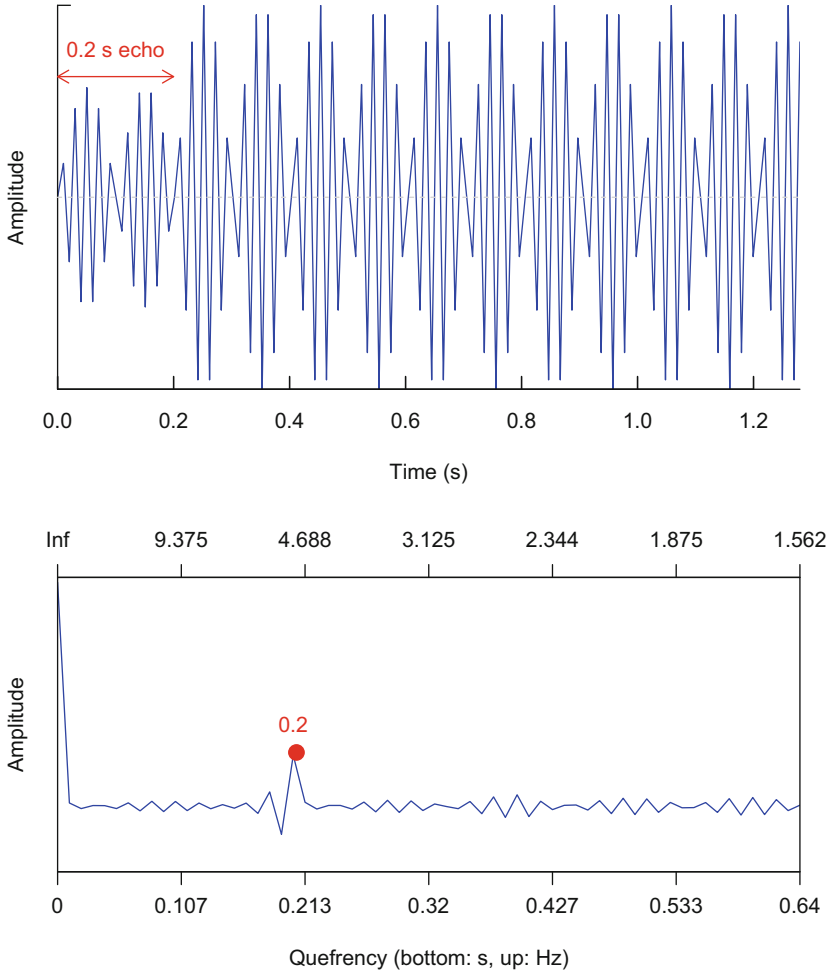


Fig. 9.14 Cepstrum: echo detection. The original signal is a 45 Hz signal affected by an echo arriving with a delay of 0.2 s and an increase of 50% of amplitude (upper panel). Applying the complex cepstral transform returns a graphic with a quefrency x-axis and an amplitude y-axis. A peak appears at 0.2 s (bottom x-axis scale) corresponding to 5 Hz (top x-axis scale) (bottom panel) corresponding to the echo delay

100 Hz. The cepstrum will then have a peak at 100 Hz. However, the result might be less accurate as the spectrum includes less frequency lobes to be analyzed by the FFT (Fig. 9.16).

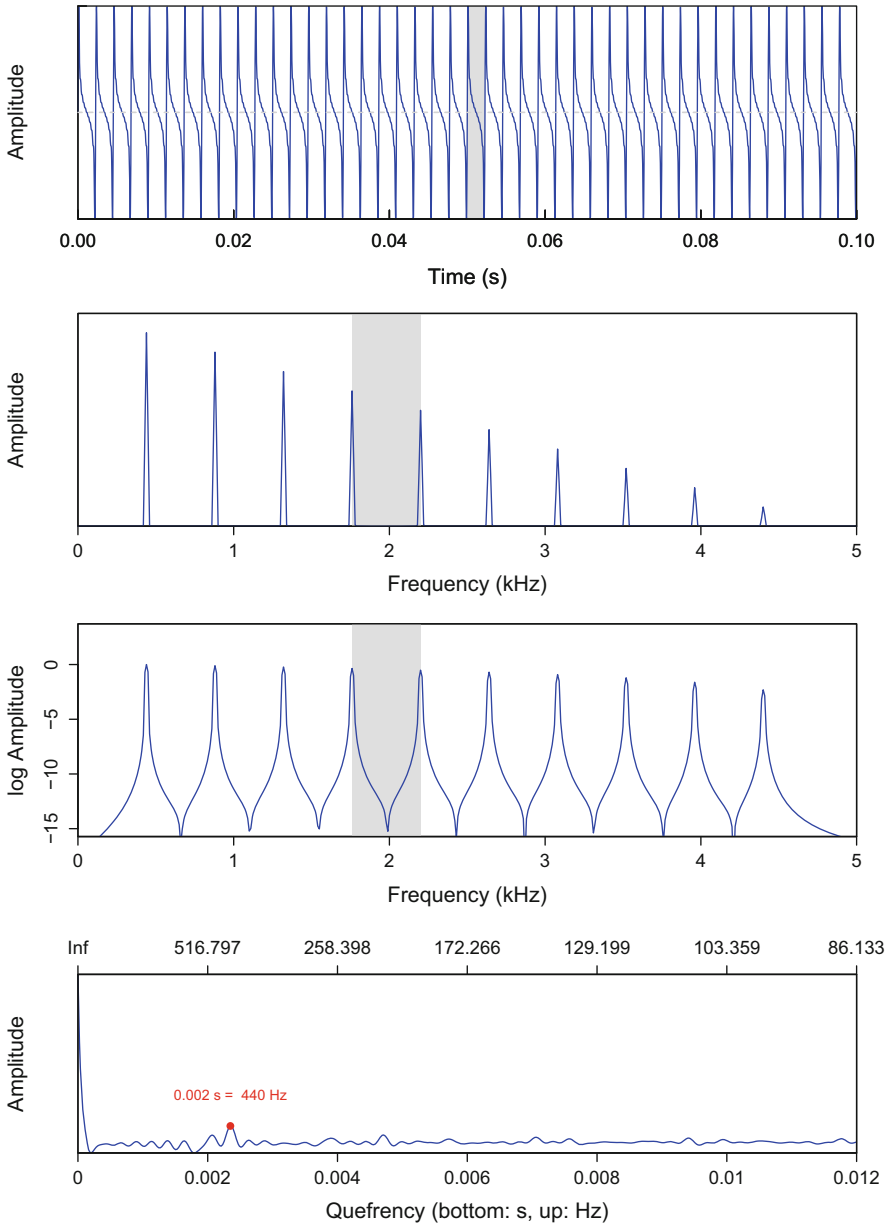


Fig. 9.15 Cepstrum of a harmonic series. The original signal is a 0.1 s harmonic series with a 440 Hz fundamental frequency and nine harmonics regularly and linearly decreasing in amplitude. The 440 Hz fundamental frequency can be seen as a regular amplitude modulation (gray area) (first panel). The spectrum is therefore made of ten frequency peaks spaced by 440 Hz (gray area) (second panel). The logarithm of the frequency spectrum shows the same profile with the same distance between peaks, but frequency peaks are compressed (third panel). The cepstrum shows a peak at a quefrency of 0.002 s equivalent to 440 Hz (fourth panel)

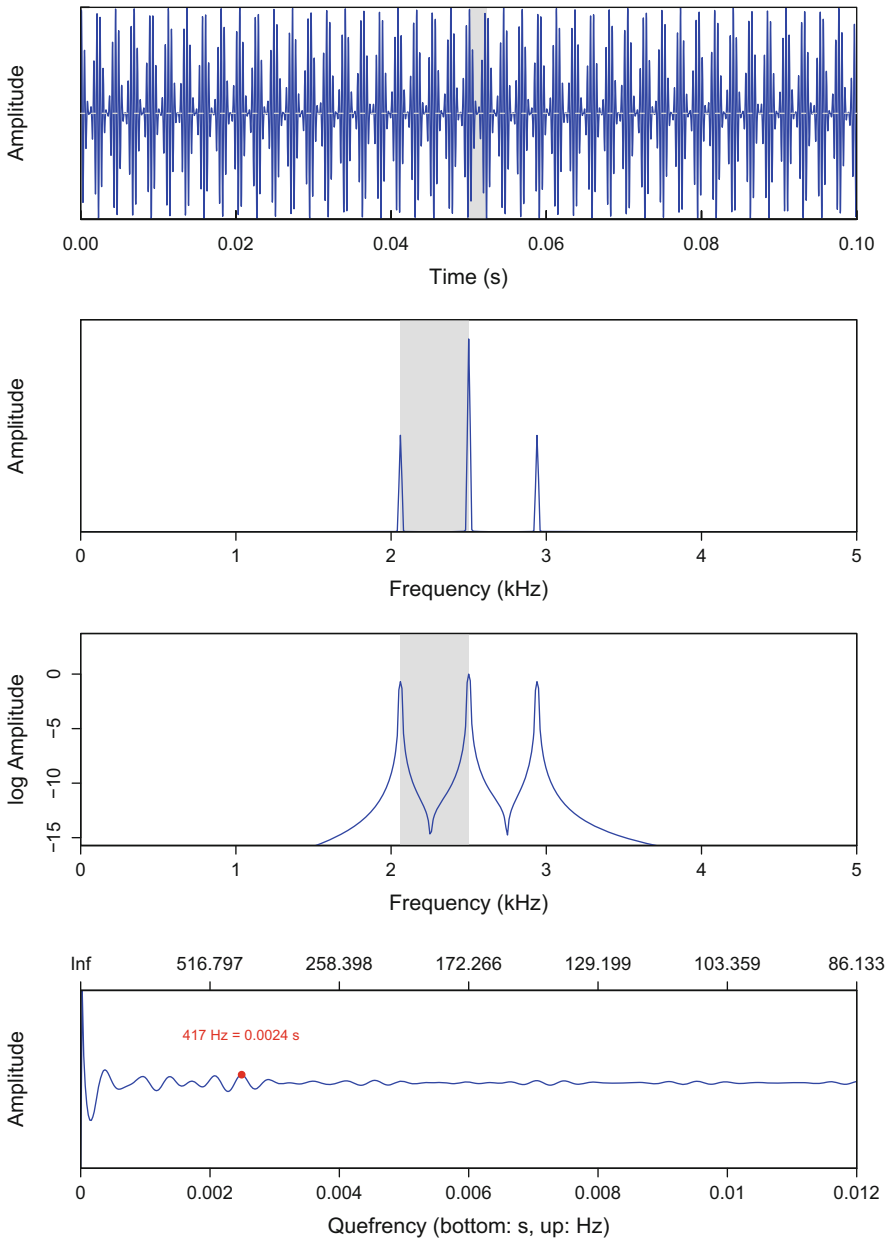


Fig. 9.16 Cepstrum of an amplitude modulated signal. The original signal is a 2500 Hz pure tone signal with an amplitude modulation of 440 Hz (gray area) lasting 0.1 s (first panel). The spectrum is made of three frequency peaks, a dominant frequency peak at 2500 Hz and two lateral frequency peaks at $2500 - 440 = 2060$ Hz and $2500 + 440 = 2990$ Hz (gray area) (second panel). The logarithm of the frequency spectrum shows the same profile with the same distance between peaks, but frequency peaks are compressed (third panel). The cepstrum shows a peak at a quefrency of 0.0024 s equivalent to 417 Hz, slightly departing from the 440 Hz modulation frequency (fourth panel)

Chapter 10
Frequency, Quefrency, and Phase in Practice

Now that we know a little bit about the Fourier transform (see Chap. 9), we can explore and describe sound in the frequency domain using `R` functions dedicated to the frequency spectrum but also to the quefrency cepstrum and the phase portrait.

10.1 Frequency Spectrum

To introduce the frequency spectrum, we will mainly refer to the sound produced by the northern lapwing *Vanellus vanellus*, a bird commonly found in Eurasia that produces a short and loud contact call that can be translated into the onomatopoeia "peewit" (Fig. 10.1). The sample is included in the data `peewit` of `seewave`:

```
data(peewit)
peewit

Wave Object
Number of Samples:      15561
Duration (seconds):     0.71
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```


J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_10

Fig. 10.1 Pictures of soniferous animals: the northern lapwing *Vanellus vanellus* (reproduced with the kind permission of Andreas Trepte, http://www.photo-natur.de) and the Italian tree cricket *Oecanthus pellucens* (reproduced with the kind permission of Christian Roesti, http://www.orthoptera.ch)

10.1.1 Functions of the Package `tuneR`

The function `periodogram()` of `tuneR`, which is based on `spec.pgram()` from the `R` base, computes the power spectral density (PSD) which is the square of the frequency spectrum of the DFT. The frequency spectrum is scaled by the sum of the DFT values returning therefore a probability mass function. The function accepts `Wave` and `WaveMC` objects and returns an object of class `Wspec()` that is plotted as a barplot by the generic function `plot()`. We can then compute the power spectrum of `peewit`

```
p <- periodogram(peewit)
p
Wspec Object (use summary() for more details)

Number of Periodograms: 1
Estimated at 8192 Frequencies: 1.345825 ... 11025

Further parameters:
width:   16384
overlap: 0
normal.: TRUE
```

and plot it with (Fig. 10.2):

```
plot(p)
```

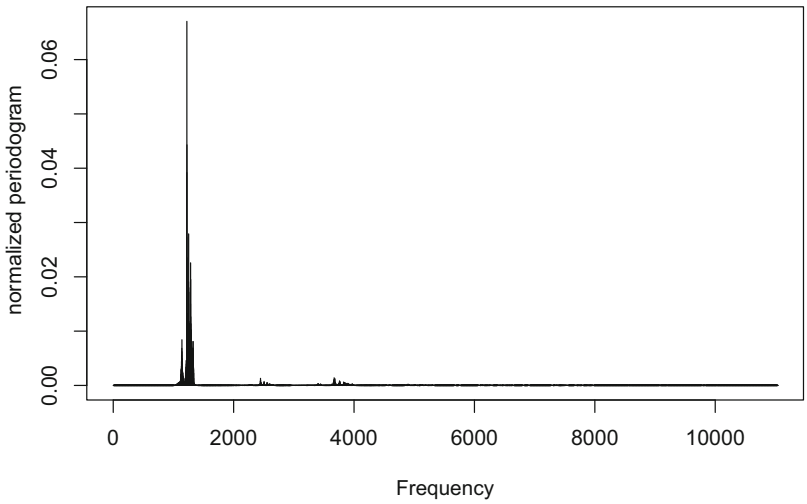


Fig. 10.2 Frequency spectrum with `periodogram()` of `tuneR`. The frequency spectrum returned by `periodogram()` is a power spectral density, that is, a frequency spectrum squared and scaled by its sum

The data of the spectrum are saved in the two first slots of the `S4` object returned so that the plot can be reconstructed manually with this code that uses the function `unlist()` to convert the *S*4 slot which is a list into a vector:

```
plot(p@freq, unlist(p@spec), type="h")
```

10.1.2 Functions of the Package `seewave`

The main function of `seewave` to compute and draw a frequency spectrum is the function `spec()`. The simplest way to use it is (Fig. 10.3):

```
fspec <- spec(peewit)
```

By default, the function `spec()` takes the complete data series as input, which can be of almost any numeric and audio format, and plots it with a frequency kHz scale on the x-axis and an relative amplitude scale scaled between 0 and 1 on the y-axis. The frequency scales range from 0, meaning that the spectrum includes the DC component, to the $N - 1$ frequency, which is the frequency preceding the Nyquist

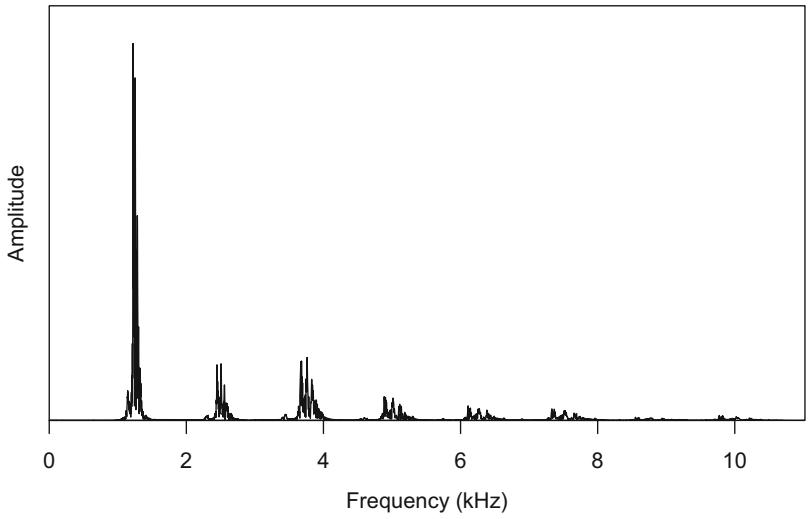


Fig. 10.3 Frequency spectrum with `spec()` of `seewave`

frequency f_N. The type of plot used by default is a line (`type="l"`) to follow most common way to publish frequency spectra. However, as we have seen in Sect. 9, the frequency spectrum is by essence a discrete function such that the most correct way to plot a spectrum is a barplot (Fig. 9.5). This can be changed easily with:

```
spec(peewit, type="h")
```

To even better visualize the discrete properties of the frequency spectrum, one can use the option `type="o"` that combines a line and points:

```
spec(peewit, type="o")
```

The spectrum can be plotted and saved into an object at the same time. The value returned is a numeric matrix with two columns, the first column being the frequency in kHz and the second column being the amplitude. The number of rows is the length of the input data N divided by 2 due to the symmetry of the DFT:

```
class(fspec)
[1] "matrix"
```

(continued)

```
dim(fspec)
[1] 7780    2
head(fspec)
                x            y
[1,] 0.000000000 2.458994e-04
[2,] 0.001417095 8.840225e-05
[3,] 0.002834190 1.444195e-04
[4,] 0.004251285 1.197009e-04
[5,] 0.005668381 2.190808e-05
[6,] 0.007085476 9.511212e-05
tail(fspec)
                x            y
[7775,] 11.01650 8.806292e-05
[7776,] 11.01791 2.343500e-05
[7777,] 11.01933 1.086190e-04
[7778,] 11.02075 4.949982e-05
[7779,] 11.02217 7.759361e-05
[7780,] 11.02358 6.917961e-05
```

The numeric output of `spec()` can be used to build manually the graphical output:

```
plot(fspec, type="l", xlab="Frequency (kHz)", ylab="Amplitude")
```

The function `cutspec()` offers a facility to manipulate the object returned by `spec()` such that we can apply a selection of the spectrum according to frequency. The following action cuts the frequency spectrum `fspec` between 2 and 4 kHz:

```
fspec.cut <- cutspec(fspec, flim=c(2,4))
dim(fspec.cut)
[1] 1412    2
head(fspec.cut)
            x            y
[1,] 2.000938 0.0015616509
[2,] 2.002355 0.0012655268
[3,] 2.003773 0.0004317273
[4,] 2.005190 0.0008492436
[5,] 2.006607 0.0011620947
[6,] 2.008024 0.0007394728
tail(fspec.cut)
                x           y
[1407,] 3.993374 0.014846632
[1408,] 3.994791 0.009850293
[1409,] 3.996208 0.005660242
```

(continued)

```
[1410,] 3.997625 0.008755836
[1411,] 3.999042 0.011353654
[1412,] 4.000460 0.011816028
```

The function `spec()` includes a long list of arguments that can be divided in two broad categories: (1) the arguments to control the analysis (Fourier arguments) and (2) the arguments to manipulate the graphical output (graphical arguments). We detail these categories successively in the two next sections.

10.1.2.1 Fourier Arguments

We have seen that the format of the input is rather easy to manage as `spec()` handles most of audio-related `R` object classes. However, `spec()` takes by default the complete input object leading to a very high-frequency resolution; here for the case of `peewit`, which is made of 15561 samples, we end up with a resolution $\Delta_f = 22050 \div 15561 = 1.4$ Hz. Such a high-frequency resolution is often not required, if not irrelevant. In addition, computing the DFT of the whole sound might be time consuming and not be appropriate if there is frequency modulation. It can be therefore useful to compute the DFT for a section of the data only, something that the arguments `from` and `to` allow by specifying where the analysis should start and end. They must be provided in s, such that the following code computes the DFT of `peewit` between the time positions 0.3 and 0.4 s:

```
spec(peewit, from=0.3, to=0.4)
```

Another way is to compute the DFT locally using a so-called window that selects a specific number of samples at a particular time position (Fig. 10.4). For instance, we could wish to get the frequency spectrum of `peewit` in the middle of the signal. To do so, we use the argument `at` to specify the time position and the argument `wl` (for **w**indow **l**ength) to set the length of the window in number of samples. This argument, which we will often encounter with frequency related functions, is usually set with a power of 2 value facilitating the computation of the DFT. Classical values are $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$. Table 10.1 provides the frequency resolution in relation with window length and sampling frequency, and Fig. 10.5 illustrates the effect of DFT size on the shape and resolution of the frequency spectrum.

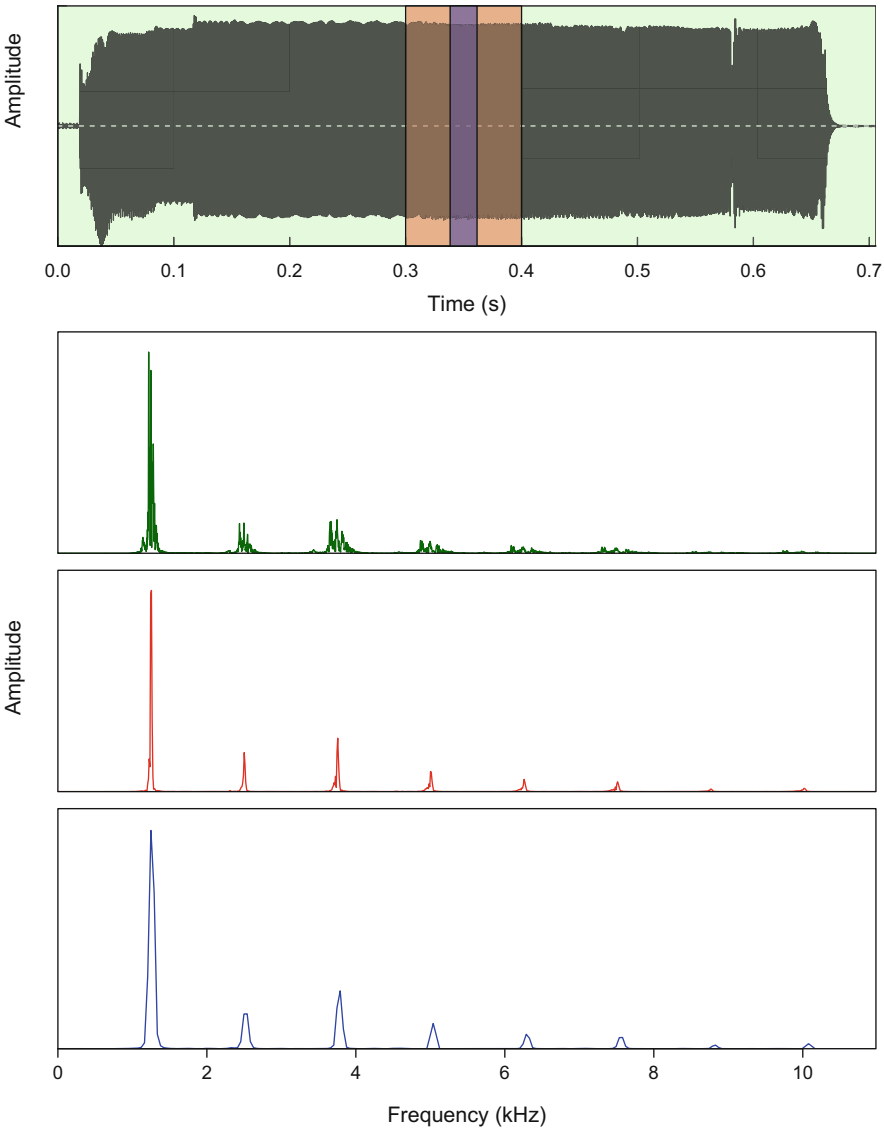


Fig. 10.4 Size of the frequency spectrum—1. The frequency spectrum is computed with `spec()` over the complete `peewit` dataset (top), on a section between 0.3 and 0.4 s (middle) and on a 512 sample window selected in the middle of the sound (bottom)

Table 10.1 Frequency and time resolution

			Sampling frequency f_s (Hz)			
			11,025	22,050	44,100	98,000
DFT size wl (samples)	128	Δ_f (Hz)	86.13	172.27	344.53	750
		Δ_t (ms)	11.61	5.8	2.9	1.33
	256	Δ_f (Hz)	43.07	86.13	172.27	375
		Δ_t (ms)	23.22	11.61	5.8	2.67
	512	Δ_f (Hz)	21.53	43.07	86.13	187.5
		Δ_t (ms)	46.44	23.22	11.61	5.33
	1024	Δ_f (Hz)	10.77	21.53	43.07	93.75
		Δ_t (ms)	92.88	46.44	23.22	10.67

The time and frequency resolution, respectively, Δ_f and Δ_t, are given in relation with the sampling frequency (f_s) and the size of the DFT wl. One of the most common settings is highlighted with a gray background

In the following code, we find the time center of `peewit`:

```
center <- round(duration(peewit)/2, 2)
center
[1] 0.35
```

and we compute the frequency spectrum for a $2^9 = 512$ sample window around the central position of `peewit` with:

```
spec(peewit, wl=512, at=center)
```

The frequency resolution steps down to $\Delta_f = 22050 \div 512 = 43.06$ Hz. Increasing the window size increases the frequency resolution, but the decomposition is less accurate in terms of time as more signal is selected. Inversely, reducing the window size is more specific in terms of time (position) but the frequency resolution decreases. This trade-off is an example of the uncertainty or Heisenberg principle that stipulates that there is a limit in the precision of pairs of parameters, here the time and frequency parameters (see Sect. 11.1.2). This issue can be partly solved by computing a mean spectrum through the use of a sliding window. This solution, available with the function `meanspec()`, is detailed in Chap. 11.

We have seen that frequency leakage can be limited by multiplying the data by a window function or taper $w[n]$ (see Sect. 9.6). This operation is achieved by default by `spec()` with a Hanning window, but the type of the window can be changed with the argument `wn` (for **w**indow **n**ame), here with a Bartlett window:

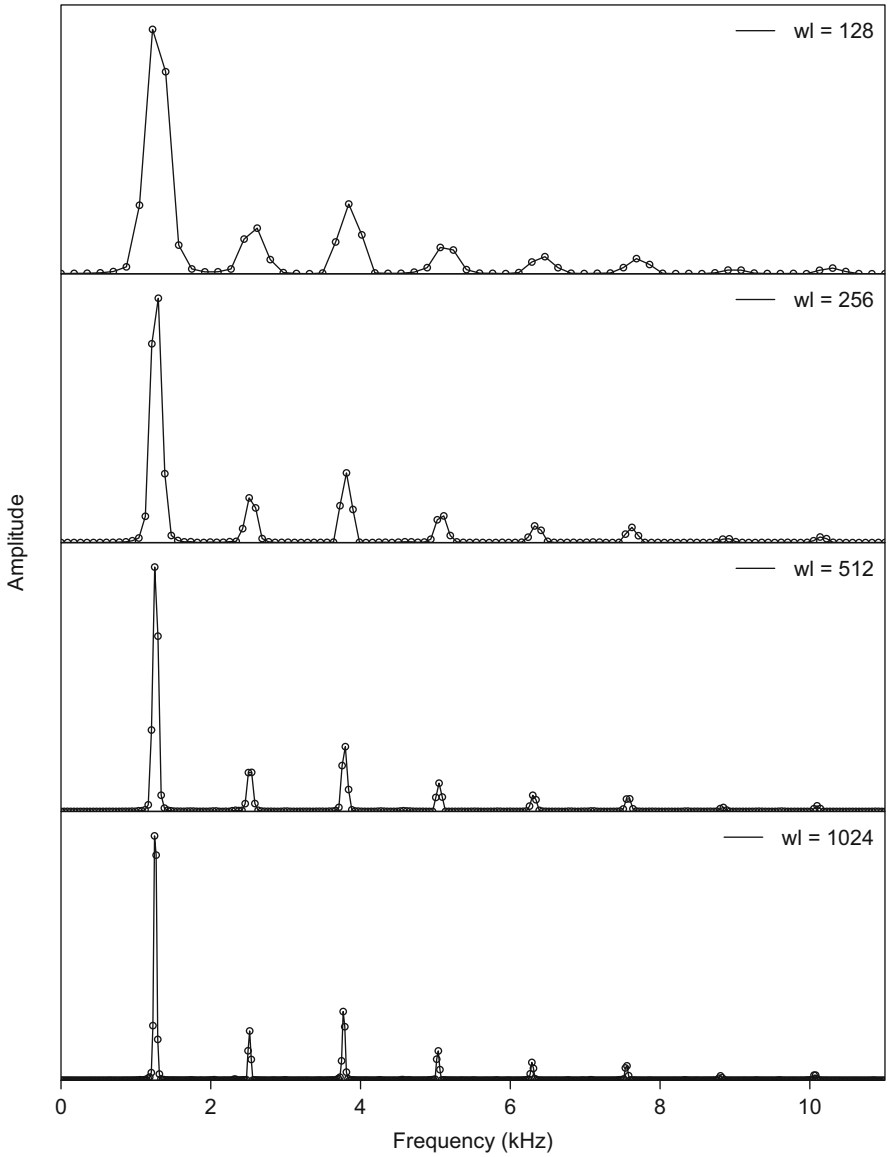


Fig. 10.5 Size of the frequency spectrum—2. The frequency spectrum is computed with `spec()` at the center of `peewit` dataset with different DFT sizes (128, 256, 512, 1024). The spectrum is displayed with a line and points to highlight the frequency resolution

```
spec(peewit, wn="bartlett")
```

The power spectral density can also be computed by turning the argument `PSD` to `TRUE`:

```
spec(peewit, PSD=TRUE)
```

The different scaling and normalization of the frequency spectrum data are available with the following arguments:

- `scaled=TRUE`: returns a spectrum scaled to the length of the DFT,
- `norm=TRUE`: returns a spectrum normalized between 0 and 1,
- `PMF=TRUE`: returns a **p**robability **m**ass **f**unction, that is, a spectrum normalized by the sum of the frequency amplitude.

If `norm=FALSE`, `scaled=FALSE` (default), and `PMF=FALSE` (default), `spec()` returns the raw data of the DFT. In this case, the function used to taper the window changes the overall amplitude of the signal. It is necessary to apply a correction factor to obtain values in the frequency domain that fits with values in time domain (see the Parseval's theorem in Sect. 9.3). There are two possible corrections, the amplitude and energy correction defined as:

$$\text{Amplitude correction} = \frac{1}{\bar{w}}$$

and

$$\text{Energy correction} = \sqrt{\frac{1}{\bar{w}^2}}$$

where $\bar{w}$ is the mean of the window function. These corrections can be applied with the argument `correction` set to either `"none"` (default), `"amplitude"`, or `"energy"`.

Here is a test of these different options without plotting the results (`plot=FALSE`) and by checking the ranges of the amplitude values extracted from the second column of each returned object:

```
fspec.norm   <- spec(peewit, norm=TRUE, plot=FALSE)
range(fspec.norm[,2])   # normalised in [0,1]
[1] 2.103256e-06 1.000000e+00
fspec.pmf    <- spec(peewit, PMF=TRUE, plot=FALSE)
range(fspec.pmf[,2])    # probability mass function
```

(continued)

```
[1] 2.922088e-08 1.389317e-02
fspec.scaled <- spec(peewit, scaled=TRUE, norm=FALSE,
                     plot=FALSE)
range(fspec.scaled[,2]) # scaled by DFT length
[1] 4.952858e-03 2.354853e+03
fspec.raw    <- spec(peewit, norm=FALSE, plot=FALSE)
range(fspec.raw[,2])    # raw data without correction
[1] 7.707142e+01 3.664386e+07
fspec.raw.a  <- spec(peewit, norm=FALSE,
                             correction="amplitude", plot=FALSE)
range(fspec.raw.a[,2])  # raw data with amplitude correction
[1] 1.541527e+02 7.329244e+07
fspec.raw.e  <- spec(peewit, norm=FALSE,
                             correction="energy", plot=FALSE)
range(fspec.raw.e[,2])  # raw data with energy correction
[1] 9.849008e+01 4.682744e+07
```

Knowing how to use all these arguments, the results of `periodogram()` can be reproduced with the following `spec()` options:

```
spec <- spec(peewit, PSD=TRUE, type="h", PMF=TRUE, las=0)
```

We have so far dealt with linear amplitude spectra but `spec()` can produce dB spectra as `periodogram()` thanks to the argument `dB`. This argument can take five different values: `"max0"` for regular dB maximized to 0 and `"A"`, `"B"`, `"C"`, and `"D"` for dB(A), dB(B), dB(C), and dB(D), respectively. For example, to plot a frequency spectrum with a dB scale maximized to 0 (Fig. 10.6):

```
spec(peewit, at=center, dB="max0")
```

10.1.2.2 Graphical Arguments

The argument `plot` of `spec()` can accept different values: a `FALSE` logical value to cancel plotting but to print values, a numeric value of 1 to have a vertical plot (frequency as *x*-axis, amplitude as *y*-axis), or a numeric value of 2 to have an horizontal plot (amplitude as *x*-axis, frequency as *y*-axis).

There are classical graphical arguments, namely, `col` to control for the color of the spectrum line, `cex` for the size of points if `type="p"` or `type="o"`, and `flab` and `alab` to change the labels of the **a**mplitude and **f**requency labels. The arguments `flim` and `alim` can be used to change the **a**mplitude and **f**requency limits to zoom in. In addition, because the function is built on `plot()`, most of

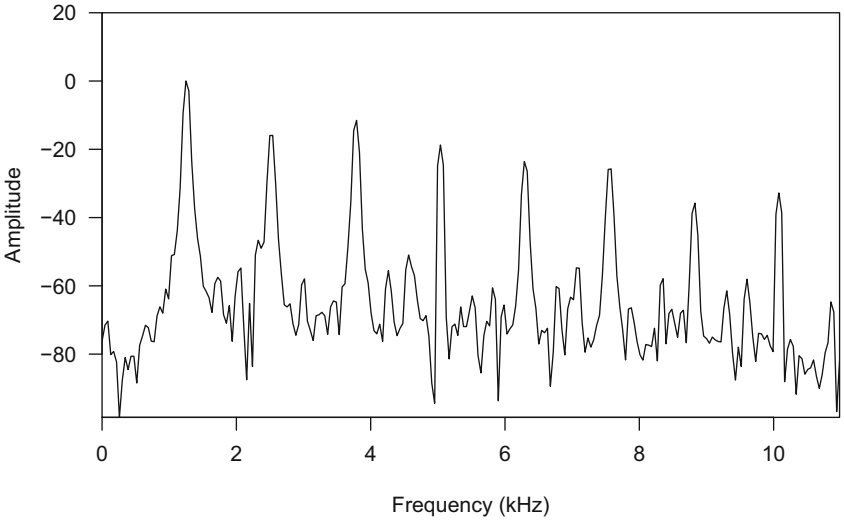


Fig. 10.6 dB frequency spectrum. Frequency spectrum computed at the center of `peewit` with a window of 512 samples. The amplitude scale is expressed in dB in reference to a maximum value set to 0

usual graphical parameters of the function `par()` can be used. In the following example, color, graphic orientation, labels, and axes limits are changed as well as line type (`lty`), line thickness (`lwd`), axis tick length (`tcl`), *y* axis scale (turned to log with `log="y"`), and title (`main`) (Fig. 10.7):

```
spec(peewit,
     plot=2,                          # plot orientation
     flab="Frequency [kHz]",          # frequency axis label
     alab="Amplitude [no unit]",      # amplitude axis label
     flim=c(2, 8), alim=c(0, 0.3),    # x-y zoom in
     col="blue", lty=2, lwd=0.75,     # line color, type and width
     tcl=0.5,                         # internal axis ticks
     log="y",                         # y-axis scale
     main="Frequency spectra")        # main title
```

10.1.2.3 Decoration

The function `spec()` can be considered as a high-level plot function such that any low-level graphical function can be used to decorate or annotate a frequency spectrum (see Sect. 3.3.9.3). The next code changes a lot of graphical parameters, not in an optimal visual aspect, but shows how much the appearance can be changed. A part of the code used the function `fpeaks()` that detects frequency peaks as explained in Sect. 10.1.6 (Fig. 10.8):

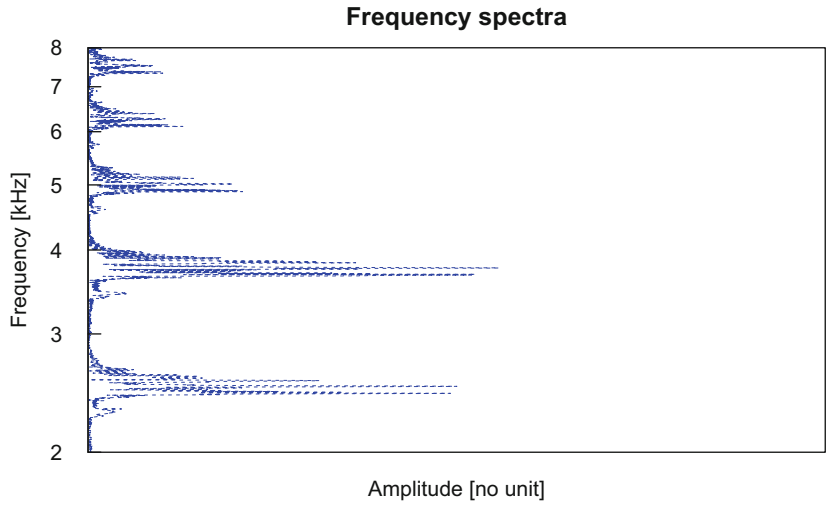


Fig. 10.7 High-level plot modifications of the frequency spectrum. The main graphical parameters of `spec()` were used to change the appearance of the frequency spectrum, including its orientation

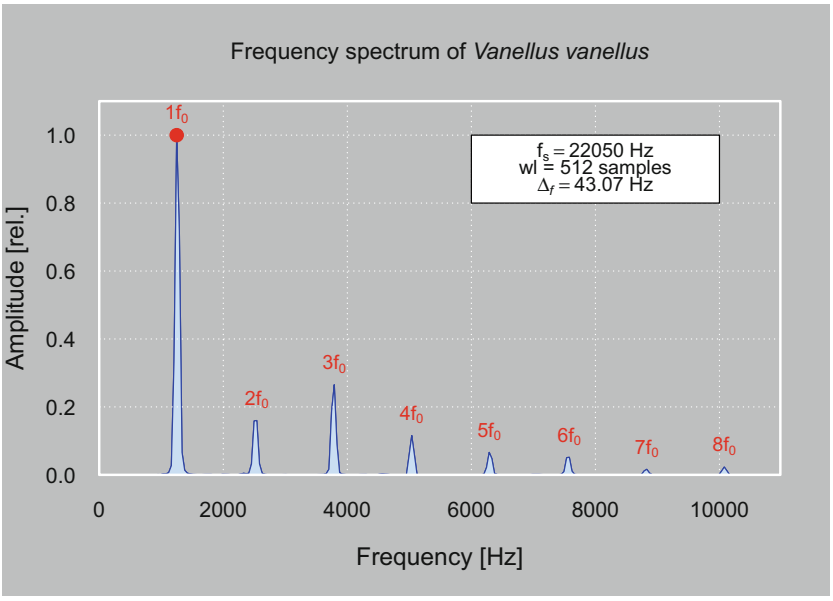


Fig. 10.8 Decoration of the frequency spectrum. This plot results from the use of low-level plot functions—`par()`, `polygon()`, `axis()`, `grid()`, `title()`, `points()`, `rect()`, `rect()`, `box()`—to change the visual output of `spec()`

```
# change the main colors and text size
par(bg="grey", fg="white", cex.lab=1.25, cex.axis=1.1, tck=0)
# plot the frequency spectrum (center of peewit)
center <- round(duration(peewit)/2, 2)
fspec <- spec(peewit, at=center, col="blue",
     xaxt="n", flab="Frequency [Hz]",
     yaxt="s", alab="Amplitude [rel.]")
# add a color surface under the spectrum line
polygon(x=c(fspec[,1], rev(fspec[,1])),
        y=c(fspec[,2], rep(0, nrow(fspec))),
        col=colours()[600], border="blue")
# add a frequency axis in Hz (not in kHz)
freq.axis <- seq(0,10,by=2)
axis(side=1, at=freq.axis, labels=freq.axis*1000)
# add x-y white grid
grid(col="white")
# add a title
title(main=expression(paste("Frequency spectrum of ",
                            italic("Vanellus vanellus"))))
# get the x-y coordinates of the main frequency peaks
freq.peaks <- fpeaks(fspec, amp=c(0.01,0.01), plot=FALSE)
# plot a label over each main frequency peak
for(i in 1:nrow(freq.peaks)) {
      text(freq.peaks[i,1], freq.peaks[i,2],
      labels=substitute(i*f[0], list(i=i)),
      col=2, pos=3)
      }
# add a point on the dominant frequency
points(x=freq.peaks[1,1],
       y=freq.peaks[1,2], pch=19, cex=1.5, col="red")
# add a white rectangle for text inset
rect(xleft=6, ybottom=0.8,
     xright=10, ytop=1, col="white", border="black")
# text inset
text(x=8, y=0.95,
     labels=expression(paste(f[s] == 22050, " Hz")), col="black")
text(x=8, y=0.90,
     labels="wl = 512 samples", col="black")
text(x=8, y=0.85,
     labels=expression(paste(Delta[italic(f)]==43.07, " Hz")),
     col="black")
# draw a complete bow around the graph
box(lwd=2)
```

10.1.2.4 Multifrequency Spectrum Plot

It may be necessary to compute the DFT at different positions along the signal and to plot all the resulting frequency spectra on a single plot. This task can be accomplished by first preparing an empty plot with `type="n"` and then adding with a `for` loop the results of successive frequency spectra (Fig. 10.9):

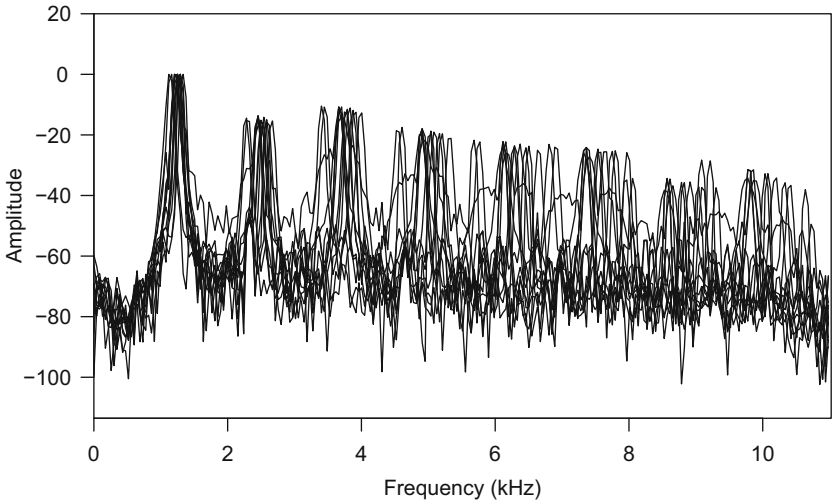


Fig. 10.9 Multifrequency spectrum plot. Thirteen frequency spectra computed regularly along `peewit` are plotted on a single graph

```
# time positions of the spectra
pos <- seq(from=0.05, to=duration(peewit)-0.05, by=0.05)
pos
 [1] 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55
[12] 0.60 0.65
# number of spectra
l <- length(pos)
l
[1] 13
# empty plot
spec(peewit, dB="max0", type="n")
# for loop around l to plot all the spectra
for(i in 1:l) {
    lines(spec(peewit, at=pos[i], dB="max0", plot=FALSE))
    }
```

The box DIY 10.1 details another option with the package `ggplot2`.

DIY 10.1 — How to plot two frequency spectra with the `ggplot2` style

Here is a solution to plot two frequency spectra on the same plot with `ggplot2` functions. We first need to load `ggplot2`:

```
library(ggplot2)
```

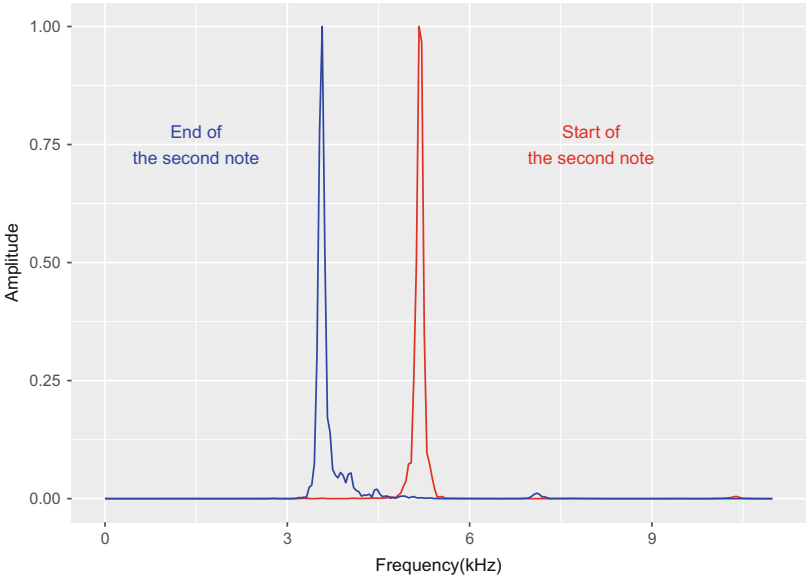
The two spectra are computed in the middle of the first and second notes of `tico`. The values of `spec()` need to be coerced into a `data.frame` to comply with `ggplot2` rules, and then the spectra are simply plotted as line geoms using the columns `x` and `y` as aesthetics.

```
# spectrum at the start of the second tico note
fspec.start <- as.data.frame(spec(tico, at=0.63, plot=FALSE))
# spectrum at the end of the second tico note
fspec.end <- as.data.frame(spec(tico, at=0.77, plot=FALSE))
# first layer of geoms to the projection of fspec.note1
p <- ggplot(data=fspec.start, mapping=aes(x,y)) +
     geom_line(col="red")
# second layer of geoms to add fspec.end
```

```
p <- p + geom_line(mapping=aes(x,y), data=fspec.end,
     col="blue")
# axes labels
p <- p + xlab("Frequency(kHz)") + ylab("Amplitude")
# annotations
p <- p + annotate("text", x = 8, y = 0.75,
                  label = "Start of\nthe second note",
                  col="red")
p <- p + annotate("text", x = 1.5, y = 0.75,
                  label = "End of\nthe second note",
                  col="blue")
# final display
p
```

(continued)

DIY 10.1 (continued)



See Wickham (2009) for more details regarding `ggplot2` coding.

10.1.2.5 Binned Frequency Spectrum

We have seen that the frequency resolution of the DFT (Δ_f) is set by the sampling frequency f_s and the number of samples of the signal to be decomposed (N for a complete sound, or wl for a section of sound specified by a DFT window), such that $\Delta_f = f_s \div N$ (Table 10.1). This resolution can be lowered after the computation of the DFT by defining new frequency intervals. Such data binning leads to a frequency spectrum having the shape of a barplot showing variation of amplitude over predefined frequency bands. Such modifications and graphical output can be run with the `seewave` function `fbands()`. The function does not take a sound as an input but a two-column matrix describing a frequency spectrum, that is, the value of the function `spec()` (or `meanspec()`; see Sect. 11.14). The properties of the frequency intervals, or bands, can be set using one of the two following options (Fig. 10.10):

1. a number of bands with equal size can be set by giving a single numeric value to the argument `bands`. For instance, choosing `bands=10` slices the spectrum into 10 equal bands,

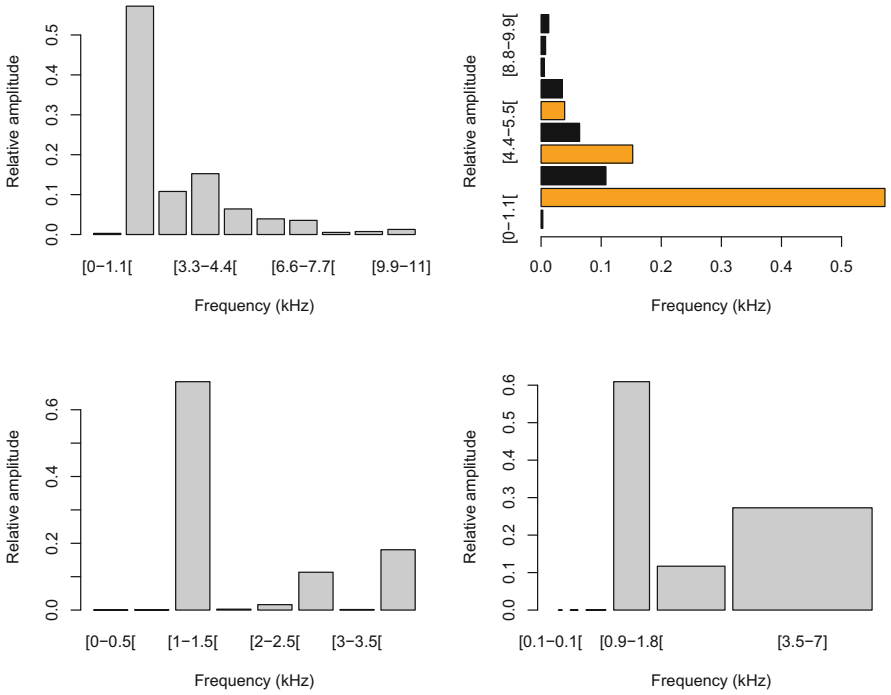


Fig. 10.10 Frequency band plot. Four displays of the function `fbands()`: ten regular frequency bands in a usual vertical orientation (top-left), ten regular frequency bands with color and orientation modifications (top-right), eight regular frequency bands defined by hand (bottom-left), and eight frequency bands defined following music octaves (bottom-right)

2. the frequency limits of the bands can be specified manually by feeding the argument `bands` with a numeric vector of length superior to 1. The limits, expressed in kHz, can follow a regular or an irregular numeric series. For instance, attributing the vector `c(0,2,4,11)` generates the bands [0, 2[, [2, 4[, [4, 11] kHz. The last frequency limit should not exceed half the sampling frequency used to obtain the spectrum.

The examples of the Fig. 10.10 are obtained with the following script:

```
# data
center <- round(duration(peewit)/2, 2)
fspec <- spec(peewit, at=center, plot=FALSE)
# plot top-left with 10 regular frequency bands
fbands(fspec, bands=10)
# plot top-right with 10 regular frequency bands
# and some graphical modifications
```

(continued)

```
col <- rep("black", 10)
col[c(2,4,6)] <- "orange"
fbands(fspec, bands=10,plot=2, horiz=TRUE, col=col)
# plot bottom-left with regular manually set frequency bands
fbands(fspec, bands=seq(0, 4, by=0.5))
# plot bottom-right with 7 octave frequency bands
# and adapted bar width
oct <- round(octaves(x=440, below=3, above=4)/1000, 1)
fbands(fspec, bands=oct, width=TRUE)
```

The frequency spectrum is a cardinal tool of data visualization. It is of prime interest to understand and explore the frequency content of a sound. However, it is obviously necessary to go a step beyond to extract qualitative and quantitative data. In the next sections, we will discover how to describe a frequency spectrum by (1) measuring the peaks, (2) interpreting the profile or shape, and (3) measuring several features using descriptive statistics.

10.1.3 Identification of Peaks

10.1.3.1 Major Peaks

The frequency spectrum is used to assess the frequency content of a time series; it is therefore crucial to be able to identify the peaks of the spectrum, that is, what are the frequencies where most of the sound energy lies in? Peak measurements can be taken either manually or automatically with the same advantages and disadvantages listed when considering the manual and automatic measurement of time features (see Chap. 8).

Manual measurements can be processed by setting the argument `identify` of `spec()` to `TRUE`. The frequency spectrum is then displayed and the console prints the following message `'Choose points on the spectrum'` to invite the user to identify any points, and in particular peaks, of the frequency spectrum. The coordinates of the chosen points can be saved in a new object, here named `res`. In the following code, we facilitate the measurements by setting the type of plot to `"o"` to more accurately visualize the frequency samples:

```
res <- spec(peewit, at=center, identify=TRUE, type="o")
```

The object is structured as a list of two items, the first item `$freq` contains the frequency coordinates and the second item `$amp` stores the amplitude coordinates.

Using this manual procedure, we found the following eight frequency peaks for the frequency spectrum computed in the middle of `peewit`:

```
res
$freq
[1]  1.248926  2.540918  3.789844  5.038770  6.287695
[6]  7.579688  8.828613 10.077539

$amp
[1] 1.00000000 0.15982195 0.26507001 0.11569401 0.06619829
[6] 0.05161196 0.01628750 0.02295913
```

The consideration of hundreds, even thousands, of sounds makes the automatic measurement of peaks inescapable. The function `fpeaks()` of `seewave` can estimate the localization of spectrum peaks and return both frequency and relative amplitude coordinates of these peaks. The basic use of `fpeaks()` is to give the value of the function `spec()` (or `meanspec()`, see Sect. 11.14) and to let the function find, plot, and return the peaks:

```
fp <- fpeaks(fspec)
```

The value of `fpeaks()` is a two-column numeric matrix with the first column being the frequency coordinates and the second column the amplitude coordinates. In this case, `fpeaks()` find all the peaks, that is, all the frequency samples that are preceded and followed by a smaller value. For `peewit`, we end up with a total of 56 peaks (Fig. 10.11):

```
class(fp)
[1] "matrix"
dim(fp)
[1] 56  2
head(fp)
           freq          amp
[1,] 0.08613281 3.045488e-04
[2,] 0.17226563 1.093189e-04
[3,] 0.34453125 8.897070e-05
[4,] 0.43066406 9.301099e-05
[5,] 0.64599609 2.639685e-04
[6,] 0.86132812 4.936136e-04
tail(fp)
           freq          amp
[51,]  9.905273 1.883691e-04
[52,] 10.077539 2.295913e-02
```

(continued)

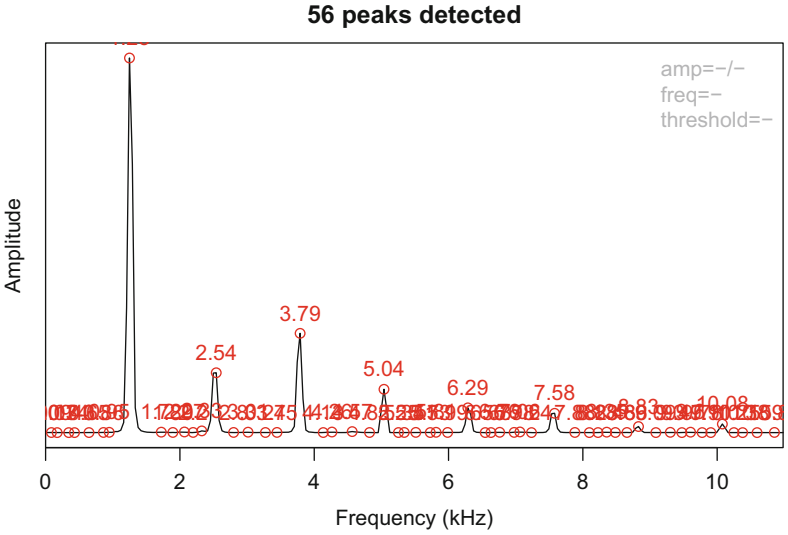


Fig. 10.11 Peak detection of frequency spectrum. Plot output of the basic use of the function `fpeaks`: all peaks, here 56, even if tenuous, are detected

```
[53,] 10.249805 1.639613e-04
[54,] 10.379004 9.483274e-05
[55,] 10.594336 8.213308e-05
[56,] 10.852734 5.814204e-04
```

There are four arguments in `fpeaks()` to apply a selection on this raw, and meaningless, peak detection: `amp`, `freq`, `nmax`, and `threshold` (Fig. 10.12).

The argument `amp` considers the difference between the amplitude of a peak with the amplitude of the frequency samples just before and after the peak. In other words, the argument `amp` considers the amplitude of the left and right slopes of each detected peak. Peaks with important slopes can be selected using two thresholds, one for the left slope and another one for the right slope. The argument `amp` requires a numeric vector of length 2, such that for a spectrum computed on a linear amplitude scale normalized to 1, specifying `amp=c(0.01, 0.01)` removes all peaks with left and right slope amplitudes lower than 0.01. Peaks are often asymmetric such that different values can be set for left and right slopes, such as `amp=c(0.01, 0.2)` for peaks with a more pronounced slope on the left than on the right. A value of 0 means that no threshold is applied: `amp=c(0,0.01)` selects peaks based on right slopes only. The values given to `amp` should be in agreement with the amplitude scale. For instance, `amp` should receive negative values when handling a spectrum computed with the option `dB="max0"`: `amp=c(-20,20)` selects peaks with slopes higher than −20 dB.

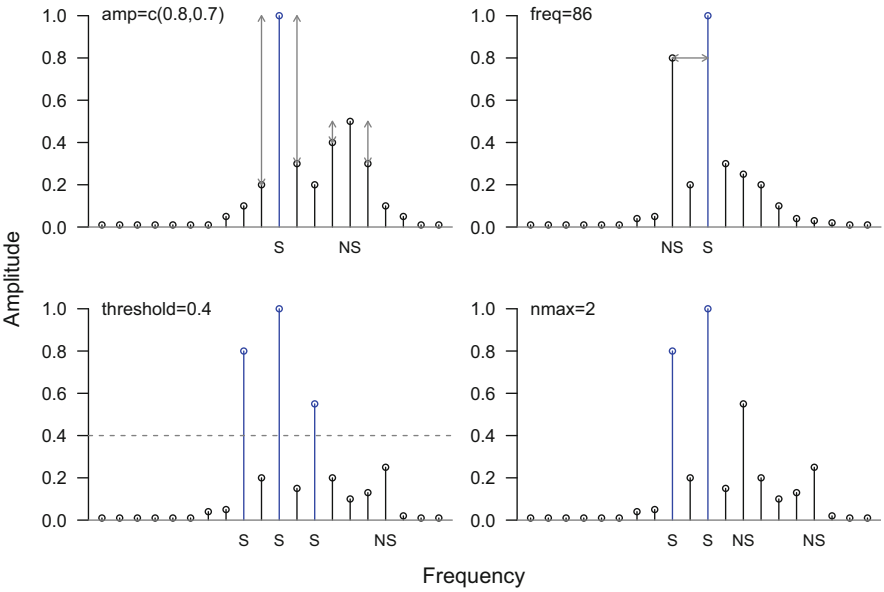


Fig. 10.12 Parameters for frequency spectrum peak detection. The function `fpeaks()` has four arguments to help in selecting the peaks of a frequency spectrum. The argument `amp` is an amplitude threshold working on the slopes of the peaks (top-left), the argument `freq` acts as a frequency threshold (top-right), the argument `threshold` is an overall amplitude threshold (bottom-left), and the argument `nmax` selects the most prominent n peaks (bottom-right). The illustration is based on schematized frequency spectra with frequency resolution of $\Delta_f = 43$ Hz. *S* selected peak, *NS* nonselected peak

The argument `freq` is a frequency threshold parameter set in Hz. If the frequency difference of two successive peaks is less than this threshold, then only the peak of highest amplitude is kept. Successive peaks with a small frequency differences can be then eliminated. As an example, if we have two successive peaks at 1200 and 1210 Hz and with an amplitude of 0.5 and 0.25, respectively (linear amplitude normalized to 1), setting `freq=50` results in selecting the first peak only.

The argument `threshold` works a simple amplitude threshold that retains all the peaks that have an amplitude above a reference value and discards all the peaks that have an amplitude below this reference. This argument can be useful when low-amplitude peaks due to background noise need to be removed. The use is straightforward as the argument `threshold` requires a single numeric value chosen within the limits of the amplitude scale.

The argument `nmax` looks for the n most prominent peaks, that is, the n peaks with the most important slopes, which are not necessarily the peaks of highest energy. The use of this argument is trivial: `nmax=4` returns the coordinates of the most significant 4 peaks. This argument overrides the arguments `amp` and `freq`.

We can try these arguments on `peewit` frequency spectrum (Fig. 10.13):

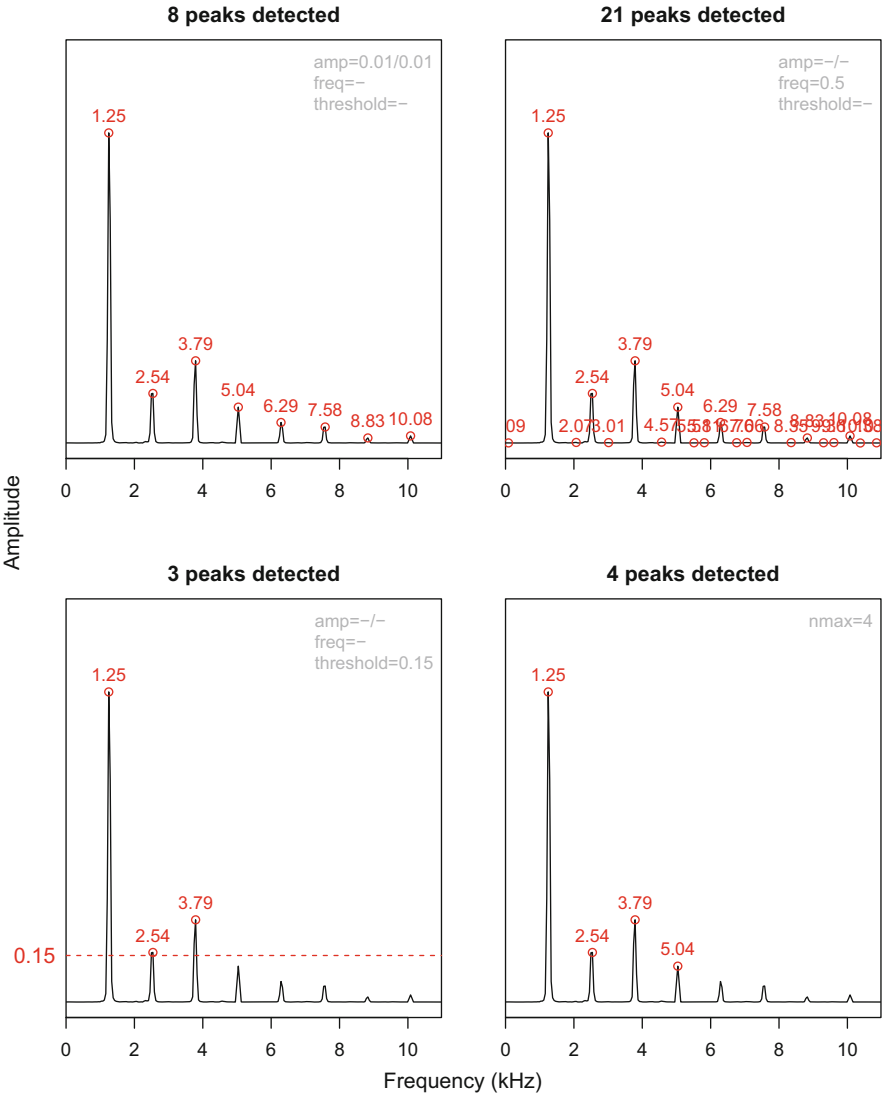


Fig. 10.13 Example of frequency spectrum peak detection. Frequency peak detection is here tested on the a frequency spectrum computed at the center of the dataset `peewit`. Each setting (arguments `amp`, `freq`, `threshold`, and `nmax`) returns a different number of peaks detected

```
fpeaks(fspec, amp=c(0.01,0.01))
fpeaks(fspec, freq=500)
fpeaks(fspec, threshold=0.15)
fpeaks(fspec, nmax=4)
```

Similar results can be obtained with a dB amplitude scale by preliminary computing the spectrum in dB with a 0 value as a maximum:

```
fspec.dB <- spec(peewit, at=center, dB="max0")
```

and then apply `fpeaks()`:

```
fpeaks(fspec.dB, amp=c(-40,40))
fpeaks(fspec.dB, threshold=-16)
fpeaks(fspec.dB, freq=500)
fpeaks(fspec.dB, nmax=4)
```

Some arguments can be used together sharping the selection of the peaks. The following is an example with a spectrum computed on `tico` (Fig. 10.14):

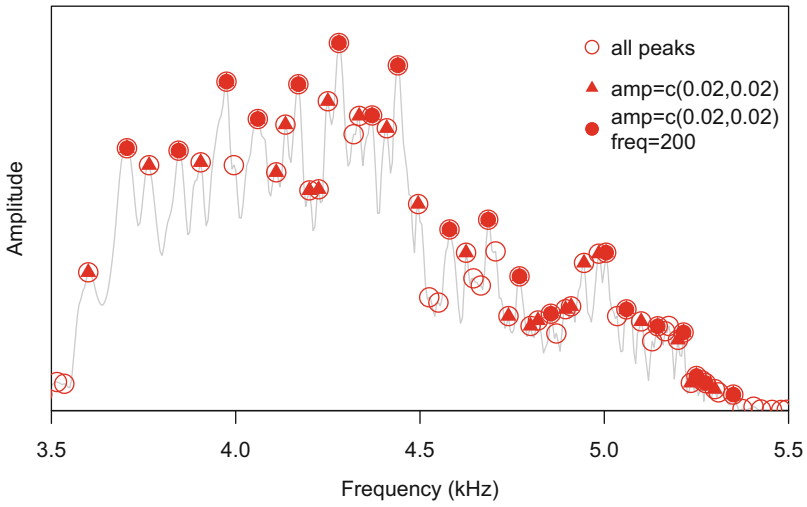


Fig. 10.14 Example of frequency spectrum peak detection with combined parameters. The figure shows peak detection on a spectrum computed for the second note of `tico` without any selection (circle), using the argument `amp` only (triangle), and the arguments `amp` and `freq` together (disk). A frequency zoom in was operated between 3.5 and 5.5 kHz

```
fspec.note2 <- spec(tico,                  # data
                    from=0.6, to=0.8,      # time selection
                    flim=c(3.5,5.5),       # frequency zoom in
                    col="grey")            # color of the line
res1 <- fpeaks(fspec.note2, plot=FALSE)    # full peak detection
res2 <- fpeaks(fspec.note2,                # detection with 'amp'
               amp=c(0.02,0.02),
               plot=FALSE)
res3 <- fpeaks(fspec.note2,                # detection
               amp=c(0.02,0.02),           # with 'amp' and 'freq'
               freq=200, plot=FALSE)
points(res1, pch=1,  col=2, cex=2)         # results as a circle
points(res2, pch=17, col=2, cex=1.25)      # results as a triangle
points(res3, pch=19, col=2, cex=1.5)       # results as a disc
legend("topright",                         # legend
       legend=c("all peaks",
                "amp=c(0.02,0.02)",
                "amp=c(0.02,0.02)\nfreq=200"),
       pch=c(1,17,19), col=2,
       pt.cex=c(1.4,1,1.3), bty="n")
```

10.1.3.2 Local Peaks

The main goal of the function `fpeaks()` is to retrieve peaks of important amplitude wherever they are along the frequency scale. However, it can be interesting to identify the peaks over specific frequency bands. This corresponds to the identification of local or regional peaks, a region being defined by lower and upper frequency limits. The function `localpeaks()` of `seewave` looks for such local peaks by dividing the frequency spectrum in bands defined by the user and by looking for the major—or dominant—peak within each band or region. The function `localpeaks()` uses the function `fbands()` (see Sect. 10.1.2.5) to split the frequency spectrum in successive region. By default, `localpeaks()` divides the frequency bands in ten equal regions, but the band limits can be specified manually with the argument `bands`. The following code was used to produce the Fig. 10.15:

```
layout(matrix(1:4, nc=2, byrow=TRUE))
par(oma=c(3,2.5,0,0), mar=c(2,2,1,1), las=1)
localpeaks(fspec)
localpeaks(fspec.dB, bands=seq(0,11.025,by=0.5))
localpeaks(fspec.dB, bands=c(0,0.5,1,1.5,3,4,11.025))
localpeaks(fspec.dB, bands=octaves(440, below=1, above=5)/1000)
## legend
mtext("Frequency", side=1, outer=TRUE, line=1.5)
mtext("Amplitude", side=2, outer=TRUE, line=1, las=0)
```

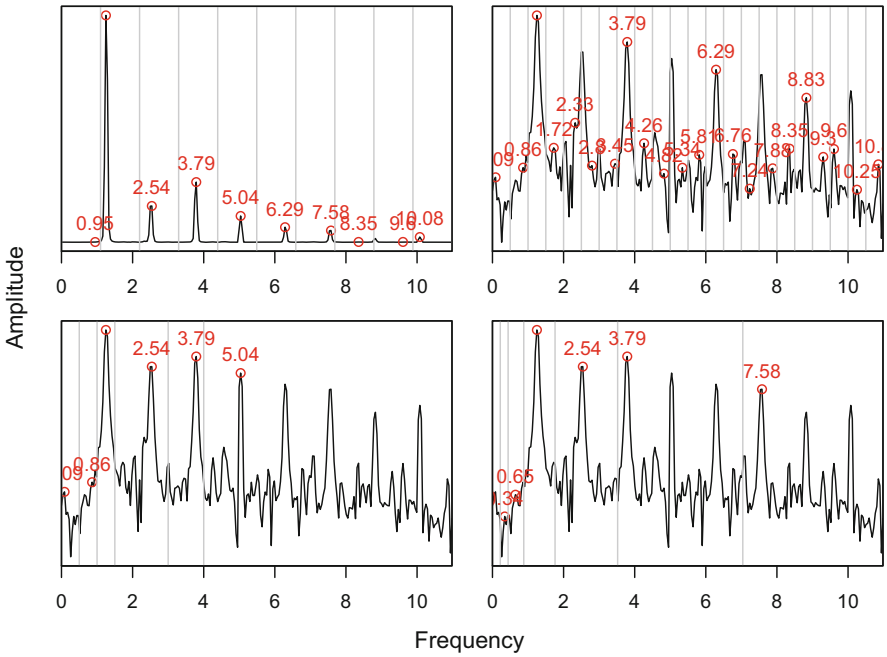


Fig. 10.15 Local peak detection on the frequency spectrum. The peak of maximum energy is identified for specific frequency regions defined with the argument `bands` of the function `localpeaks()`. Detection over ten regular frequency regions (top-left), over 500 Hz wide regions (top-right), seven irregular regions (bottom-left), and octave-based regions (bottom-right)

10.1.3.3 Dominant Frequency

The frequency of the highest energy, that is, the peak of the frequency spectrum with the highest amplitude value, is the dominant frequency. The dominant frequency is one of the most important features of sound. This feature is rather easy to identify as it corresponds to the maximum value of the frequency spectrum. If we consider the frequency spectrum as a statistical distribution, then the dominant frequency is the mode. It could be returned by hand in kHz with:

```
fspec[which.max(fspec[,2]),]
       x        y
1.248926 1.000000
```

We can also use the function `fpeaks()` with `nmax=1`:

```
fpeaks(fspec, nmax=1, plot=FALSE)
          [,1] [,2]
[1,] 1.248926    1
```

`dfreq()` is another `seewave` function that directly returns the dominant frequency. It is nonetheless mandatory to be precise where to compute the frequency spectrum (argument `at`) and to specify the properties the analysis windows (arguments `wl` and `wn`). These two latter arguments have the usual default values of 512 and `"hanning"` such we only need to set `at`:

```
res <- dfreq(peewit, at=center, plot=FALSE)
res
              x        y
[1,] 0.0000000       NA
[2,] 0.3500000 1.248926
[3,] 0.7057143       NA
```

In this case, the value returned is a numeric matrix with three lines, the single line of interest is the second one. This specific format is due to the fact that `dfreq()` was initially developed to compute the dominant frequency at different time step using a sliding window. This process and all other arguments of `dfreq()` are introduced in Sect. 13.1.1.

10.1.3.4 Fundamental Frequency

If the dominant frequency is easy to identify, the fundamental frequency is much more difficult to determine because its amplitude can be low due to the action of one or several resonators that amplify overtones. The detection of the fundamental frequency cannot therefore be based on a simple analysis of the frequency spectrum. A solution is to identify the first rahmonic peak of the cepstrum (see Sect. 9.8).

The `seewave` function `fund()` can estimate the fundamental frequency based on the detection of the cepstrum peak with the lowest quefrency. The use of `fund()` is similar to the one of `dfreq()` as it is also primarily based on short-time function with a sliding window. There is, however, one more argument to consider: `fmax` expressed in Hz is the upper limit where the function should look for the fundamental frequency. This mandatory argument greatly helps in finding the right peak. For instance, in the following, we estimate the fundamental frequency in the middle of `peewit`, and we stipulate that the fundamental frequency cannot have a value higher than 2000 Hz:

```
fund(peewit, at=center, fmax=2000, plot=FALSE)
        x        y
[1,] 0.35 1.297059
```

The result is based on the analysis of the cepstrum of `peewit` depicted in Fig. 10.26.

In this case, as clearly shown in Fig. 10.3, the fundamental frequency is also the dominant frequency. However this is not the case when resonating systems amplify overtones (see Sect. 10.1.4). This is particularly the case of the sheep (*Ovis aries*) that produces an harmonic series where the third harmonic is the dominant frequency. A typical sheep bleat is available in the `seewave` dataset `sheep` sampled at $f_s = 8000$ Hz:

```
data(sheep)
sheep

Wave Object
Number of Samples:      19764
Duration (seconds):     2.47
Samplingrate (Hertz):   8000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

We can have a look at both spectrum and cepstrum of `sheep` computed over the same time window localized at 1 s and made of 512 samples (default value of `wl`) (Fig. 10.16):

```
par(mfrow=c(2,1))
spec(sheep, at=1)
ceps(sheep, at=1)
```

```
fund(sheep, f=8000, fmax=300, at=1, plot=FALSE)
     x         y
[1,] 1 0.1538462
```

However, it is important to note that detection of the fundamental `fund()` works only for an harmonic series, following the definition of the fundamental frequency which can be viewed as the first harmonic of an harmonic series.

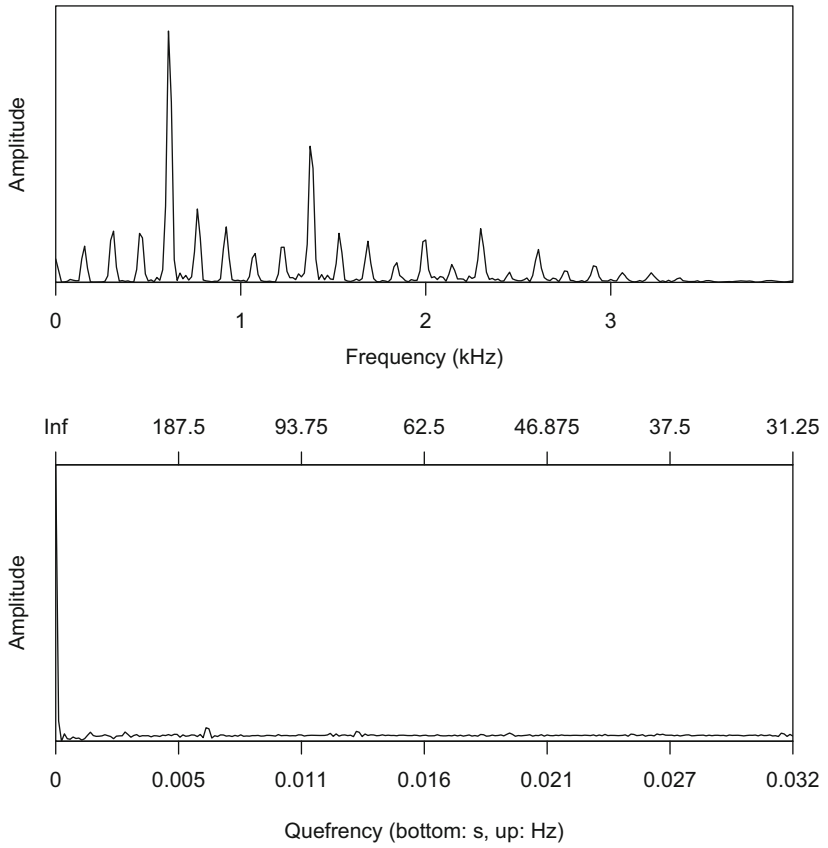


Fig. 10.16 Frequency spectrum and quefrency cepstrum of a sheep bleat. The plots were obtained with `spec()` and `ceps()`, respectively

10.1.4 Profile Analysis

The shape of the frequency spectrum embeds important information that should be scrutinized carefully for an appropriate sound description of sound frequency content. Different factors, such as resonance, amplitude modulations, frequency modulations, or two-voice systems, can make the interpretation of the peaks of a frequency spectrum risky.

It is important to remember that the value of the Fourier coefficients, that is, the energy of each component of the frequency spectra, increases when the signal deviates from that of a single sinusoidal wave of infinite duration (Bradbury and Vehrencamp 1998). Non-sinusoidal amplitude or frequency modulations and aperiodicity increase the number of significant frequency components. For instance, an infinite pure tone is characterized by a single frequency component when an instantaneous pulse is represented by a complete set of frequency components with equal energy drawing a flat frequency spectrum.

There is no `R` function to give a diagnostic on the spectrum shape and composition but the following sections try to illustrate cases of frequency spectrum shapes commonly encountered.

10.1.4.1 Harmonic Series

When the frequency spectrum shows a series of P frequency peaks, it is necessary to identify if the series follows an harmonic series and to localize the fundamental and dominant frequency. Figure 10.17 shows an harmonic series with the fundamental frequency being the dominant frequency as well (Fig. 10.17, top-left) and the same harmonics series filtered with a resonating system that changes the profile of the spectrum by enhancing the energy of harmonics and reducing the energy of the fundamental frequency such that the fundamental and the dominant frequency peaks are clearly distinct (Fig. 10.17, top-right). In some cases, the fundamental frequency may totally vanish due to the action of a resonator.

Inharmonicity might be obvious with clear unrelated frequency peaks (Fig. 10.17, bottom-left); however, it is not always trivial to diagnose harmonicity (resp. inharmonicity). A first solution could be to check that the ratios between the overtones and the lowest frequency band follow an integer series that is as follows:

$$f_{i-1} = i \times f_0$$

with $i = \{1, 2, \ldots, P\}$.

However, this suggests that the lowest frequency band is the fundamental frequency, a requirement not always met as just mentioned. Another solution could be to estimate the difference between successive frequency bands, in other words the first derivative of the frequency of the frequency peaks (in Hz). These successive differences should be similar so that the sum of the differences of these differences should be null. This means that the sum D of the second derivative of the frequency f_i of the P frequency peaks should be null, something that can be written as:

$$D \begin{cases} = 0 & \text{if the frequency of the frequency peaks follows an harmonic series} \\ \neq 0 & \text{if the frequency of the frequency peaks does not follow an harmonic series} \end{cases}$$

with:

$$\begin{aligned} D &= \sum_{i=1}^{P-2} (f_{i+2} - f_{i+1}) - (f_{i+1} - f_i) \\ &= \sum_{i=1}^{P-2} (f_{i+2} + f_i - 2f_{i+1}) \end{aligned}$$

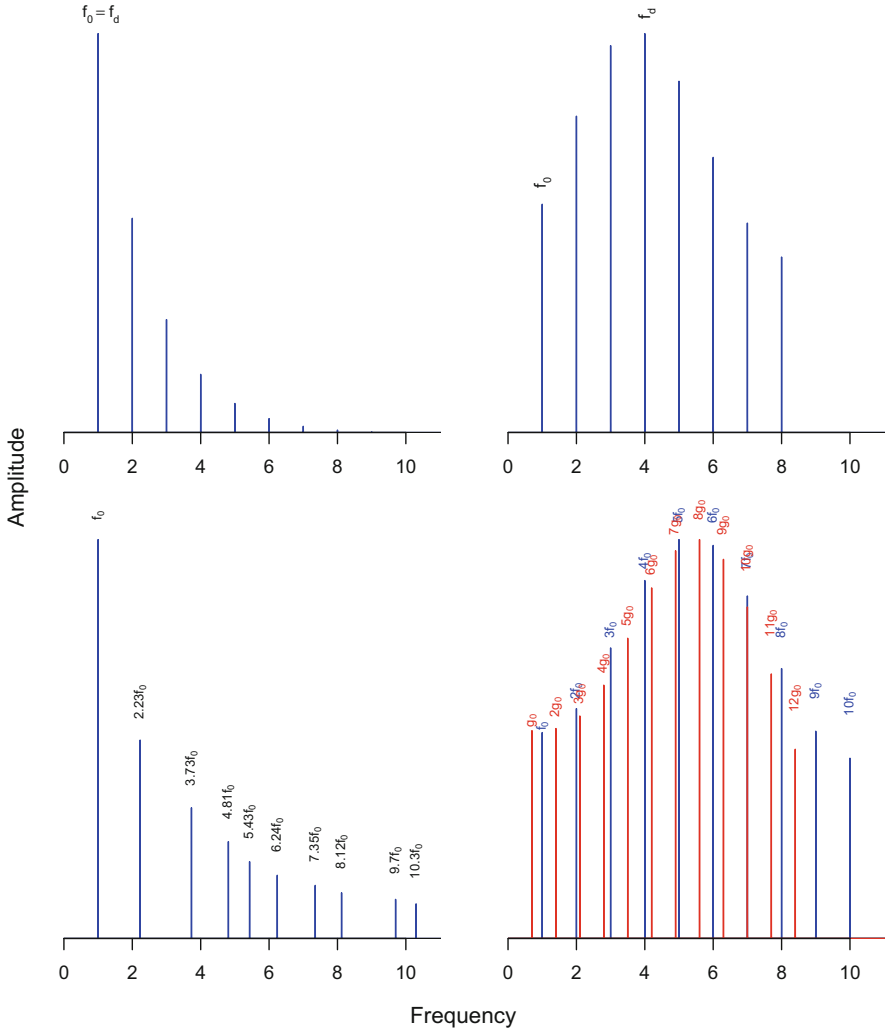


Fig. 10.17 Frequency spectrum of periodic signals—part 1. Pure harmonic series with a dominant fundamental frequency (top-left), harmonic series with a dominant frequency different from the fundamental frequency (top-right), inharmonic series (bottom-left) and two harmonics series mixed (bottom-right). f_d: dominant frequency. f_0 and g_0: fundamental frequencies

This test that can be run quite easily with the functions `fpeaks()` and `diff()`, the latter computing the *n*th derivative of a numeric vector. The following examples constitute a test for `peewit` and `sheep`. For `peewit`:

```
fspec <- spec(peewit, at=0.4, plot=FALSE) # frequency spectrum
p <- fpeaks(fspec, nmax=8, plot=FALSE)    # frequency peaks
d <- diff(p[,1], differences=2)           # second derivative
sum(d)                                    # = 0 harmonic series
[1] -4.440892e-16
```

and for `sheep`:

```
fspec <- spec(sheep, at=1.25, plot=FALSE) # frequency spectrum
p <- fpeaks(fspec, threshold=0.02,        # frequency peaks
            plot=FALSE)
d <- diff(p[,1], differences=2)           # second derivative
sum(d)                                    # = 0 harmonic series
[1] 0
```

Note that the test is valid only if all the harmonics are all apparent in the frequency spectrum and properly detected by `fpeaks()`.

Eventually, the occurrence of a two-voice system as observed in humans, birds, fish, or some insects can produce very complex spectra (Fig. 10.17, bottom-left). It is in that case very important to identify independently these two series, harmonic series or not, and to estimate how much they overlap.

10.1.4.2 Sideband Series Due to Amplitude Modulations

The amplitude of the signal has an important effect on the frequency spectrum. First, the DC component of the signal, that is, the average of the amplitude signal, corresponds to the first Fourier coefficient, that is, to the angular frequency ω_0. A signal with a non-null DC component shows a non-null 0 frequency peak (Fig. 10.18, top-left). Such frequency component can be discarded by removing the offset of the original signal, before the computation of the Fourier transform, using the function `rmoffset()` of `seewave` following the syntax (see Sect. 6.5.1):

```
tico <- rmoffset(tico, output="Wave")
```

The frequency spectrum of a pure tone signal affected by a sinusoidal amplitude modulation (AM) shows sidebands around the carrier frequency (Fig. 10.18, top-right, middle left). The frequency and the amplitude of these sidebands are determined by the frequency f_{am} and the depth m, also called the modulation

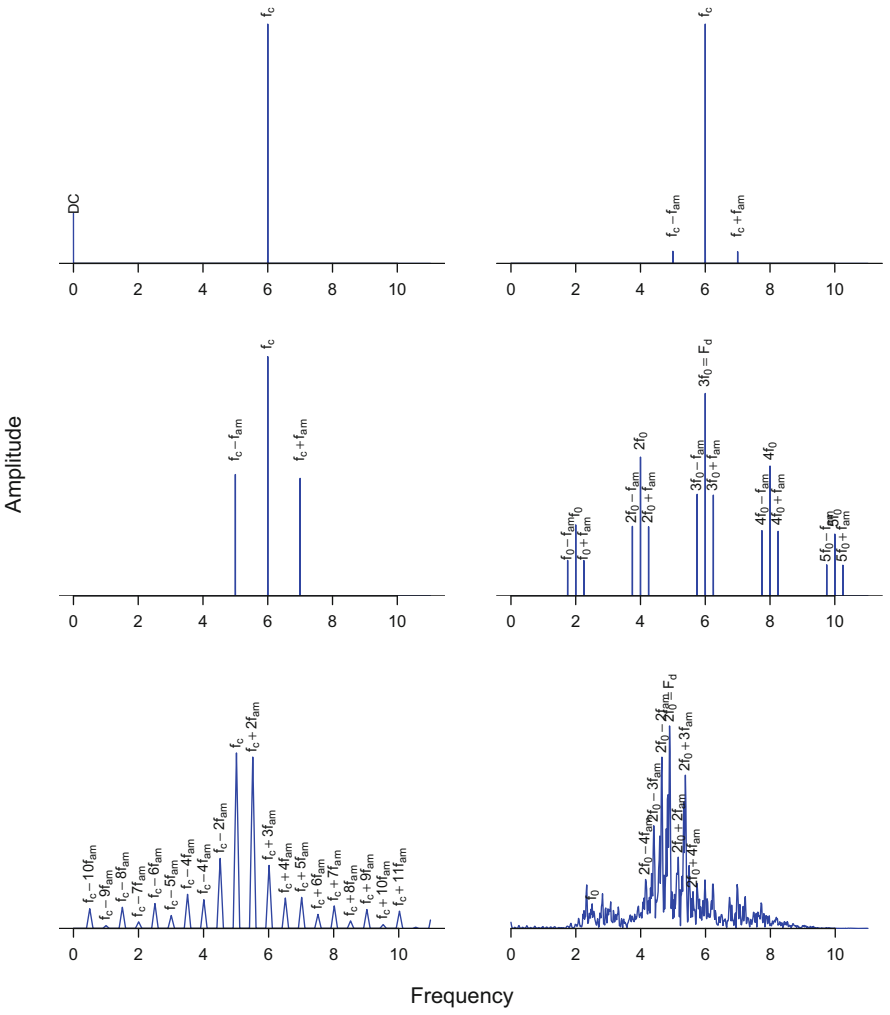


Fig. 10.18 Frequency spectrum of periodic signals—part 2. Pure sine wave with a DC component (top-left), pure sine wave with a sinusoidal amplitude modulation beating at f_{am} and with low (m=10%) modulation index (top-right), pure sine wave with a sinusoidal amplitude modulation beating at f_{am} with a maximum (m=100%) modulation index (middle-left), a harmonic series with a sinusoidal amplitude modulation beating at f_{am} with a maximum (100%) modulation index (middle-right), squared pure sine wave repeated at the frequency f_{am} (bottom-left), spectrum of `orni` which can be considered as a AM signal with periodic pauses. DC: direct current. f_c: carrier frequency. f_0: fundamental frequency

index, of the amplitude modulation (see Sect. 2.2.8). The frequency of the lower sideband is $f_c - f_{am}$, and the frequency of the upper sideband is $f_c + f_{am}$ such that the bandwidth delimited by the two sidebands is $B = 2f_{am}$. The amplitude of each sideband is $a_c m \div 2$ where a_c is the amplitude of the carrier frequency. If the signal is not a pure tone but is made of several frequencies, following or not a harmonic series, then the sidebands will pop up around each frequency peak (Fig. 10.18, middle right).

However, sound emanating from natural systems is rarely modulated by a perfect sinusoidal function but is more commonly made of the succession of short items that do not have a sine shape and that might be spaced by a short silence or pause. Such amplitude modulated signals with periodic pauses have singular frequency spectra with a series of sidebands around the carrier frequency (Fig. 10.18, bottom left). The frequency distance between the sidebands corresponds to the inverse of the time period of the items, that is, the inverse of the sum of the item duration and the following pause duration. In the case depicted in Fig. 10.18 (bottom left), the items last 0.001 s and are followed by a 0.001 s of pause, that is, a period of 0.002 s. The space between the sidebands is therefore $1 \div 0.002 = 500$ Hz.

Knowing this, we can try to interpret to frequency spectrum of the cicada *Cicada orni*. The analysis of the amplitude modulations revealed that the sound is modulated in amplitude by a slow modulation beating at around 237 Hz and a fast modulation beating at approximately 2347 Hz corresponding to the fundamental frequency (see Sect. 8.4). The dominant frequency, not yet estimated, can be assessed by focusing, for instance, on the second echeme and by using `fpeaks()`:

```
fspec <- spec(orni, from=0.1826, to=0.2421, plot=FALSE)
fdom <- fpeaks(fspec, nmax=1, plot=FALSE)[,1]
fdom
[1] 4.890669
```

This dominant frequency at 4.89 Hz seems to be related to the fundamental frequency by a factor of 2 such that it can be interpreted as the first harmonic of the fundamental frequency. The dominant frequency is the first harmonic suggesting the action of a resonator, most probably the thin and hollow abdomen of the calling male. The slow amplitude modulation is made by the more or less regular repetition of pulses. This amplitude modulation is not exactly sinusoidal but should be regarded as a waveform repeated with a period of approximately 0.0042 s. This generates sidebands around the dominant frequency spaced by $1 \div 0.0042 \approx 238$ Hz. To summarize, the song of *Cicada orni* is a harmonic series with a resonance around the first harmonic modulated by a slow non-sinusoidal amplitude modulation.

10.1.4.3 Sideband Series Due to Frequency Modulations

Frequency modulation (FM) can drastically change the shape of a frequency spectrum. A pure tone signal with a carrier frequency f_c shows a single sharp frequency peak at f_c. If this carrier frequency increases or decreases over a range of frequency, that is, if the pure tone is linearly modulated by a frequency deviation or excursion Δf_c, then the frequency spectrum will still show a single frequency peak but over a frequency bandwidth $B = \Delta f_c$. For instance, the frequency spectrum computed over a pure tone increasing from 5000 to 7000 Hz has a peak starting at 5000 Hz, maximizing at 6000 Hz and ending at 7000 Hz, that is, with a frequency bandwidth $\Delta f = 2000$ Hz (Fig. 10.19, top-left).

The frequency spectrum of a pure tone signal modulated in frequency following a regular sinusoidal way shows sidebands around the carrier frequency (Fig. 10.19, top-right, middle) (Chowning 1973). The frequency and the amplitude of these sidebands are determined by the frequency f_{fm} and the modulation index β (also named I) of the FM (see Sect. 2.2.8). The modulation index is the ratio of the peak frequency deviation Δf_c, to the frequency of the FM f_{fm}:

$$\beta = \frac{\Delta f_c}{f_{fm}}$$

For $\beta > 0$, sidebands occur above and below the carrier frequency f_c. The frequency of the lower sidebands is $f_c - kf_{fm}$ and the frequency of the upper sidebands is $f_c + kf_{fm}$, where k is a positive integer. The number of sidebands on each side of the carrier frequency, k, and the amplitude of these sidebands are related to the modulation index β. The number of sidebands k increases with β as energy of the carrier frequency is transferred to the sidebands.

Following Carson's rule, the maximum number of sidebands on each side of the carrier frequency can be roughly estimated as $\max k = \beta + 2$ for $\beta \geq 1$ and the bandwidth can be estimated using the following empirical formulae (Carson 1922):

$$\Delta f \begin{cases} = 2 \times f_{fm} & \text{for } 0 < \beta < 0.5 \\ = 2 \times (2\Delta f_c + f_{fm}) & \text{for } 0.5 \leq \beta \leq 100 \\ = 2 \times (\Delta f_c + f_{fm}) & \text{for } \beta > 100 \end{cases}$$

The amplitude of FM sidebands can be computed by invoking Bessel function of the first kind. The Bessel function of nth-order $J_n(\beta)$ is applied to the FM modulation index β (Chowning 1973) following:

$$J_n(\beta) = \sum_{p=0}^{\infty} \frac{(-1)^p}{p!(n+p)!} \left(\frac{\beta}{2}\right)^{2p+n}$$

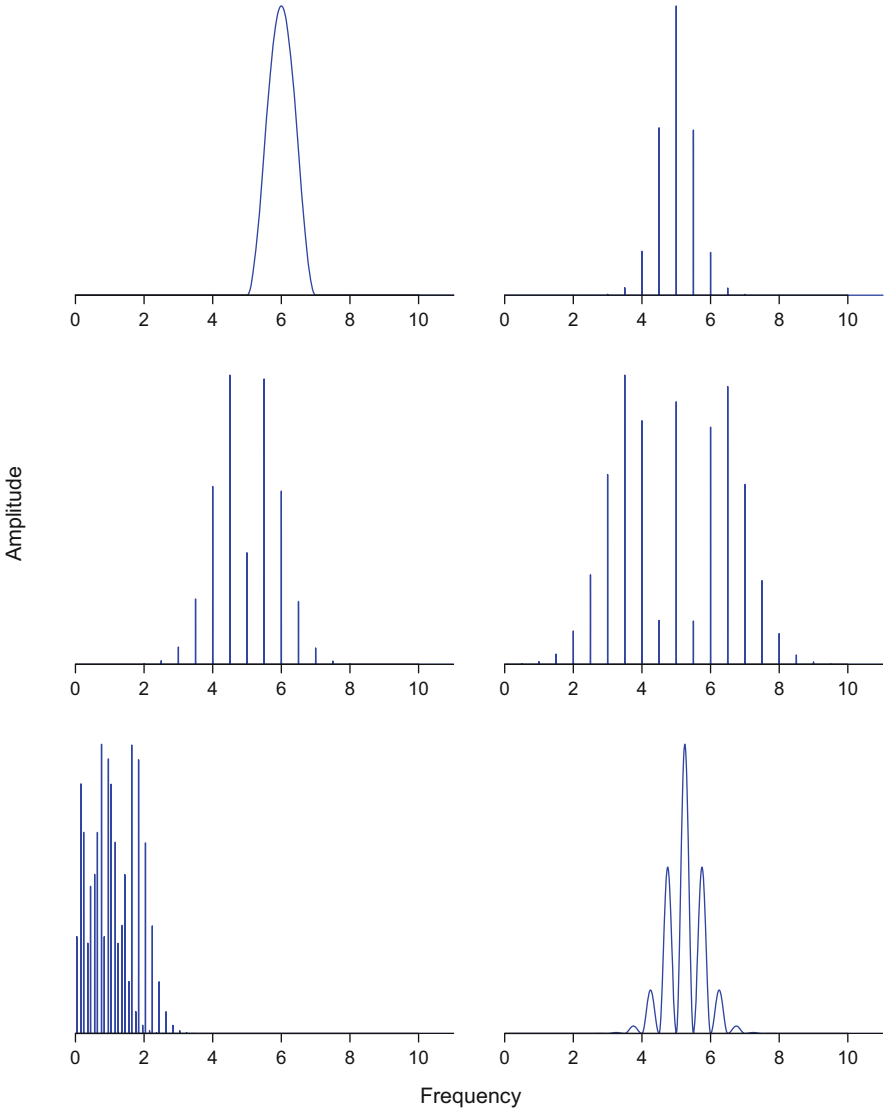


Fig. 10.19 Frequency spectrum of periodic signals—part 3. 5 kHz pure sine wave linearly increasing in frequency up to 7 kHz (top-left), 5 kHz pure sine wave affected by a sinusoidal frequency modulation with $f_{fm} = 0.5$ kHz and $\beta = 1$ (top-right), 5 kHz pure sine wave affected by a sinusoidal frequency modulation with $f_{fm} = 0.5$ kHz and $\beta = 2$ (middle-left), 5 kHz pure sine wave affected by a sinusoidal frequency modulation with $f_{fm} = 0.5$ kHz and $\beta = 4$ (middle-right), 0.44 kHz pure sine wave affected by a sinusoidal frequency modulation with $f_{fm} = 0.2$ kHz and $\beta = 8$ generating sidebands reflected around 0 (bottom-left), 5 kHz pure sine wave increasing in frequency from 5 to 5.5 kHz affected by an additional sinusoidal frequency modulation with $f_{fm} = 0.5$ kHz and $\beta = 1$ (bottom-right)

where n is a null or positive integer. $J_0(\beta)$ is the amplitude of the carrier frequency, $J_1(\beta)$ is the amplitude of the first sidebands just above and below the carrier frequency, $J_2(\beta)$ is the amplitude of the second sidebands around the carrier frequency, and so forth.

We can now try to estimate the sidebands of a given pure tone. Imagine that we have a 5000 Hz pure tone modulated by a FM with a frequency of $f_{fm} = 500$ Hz over a peak frequency deviation of $\Delta f_c = 500$ Hz. The modulation index is:

$$\beta = \frac{500}{500} = 1$$

We can estimate that there will be $\beta + 2 = 3$ sidebands on each side of the carrier frequency, that is, a total of 6 sidebands over a frequency bandwidth of $\Delta f = 2 \times (2 \times 500 + 500) = 3000$ Hz. These sidebands will appear at an interval of 500 Hz around the carrier frequency, that is, at 3500, 4000, 4500, 5500, 6000, and 6500 Hz. The amplitude of these sidebands are obtained by computing the first three Bessel function $J_{0 \leq n \leq 3}(\beta)$ of β with the `R` base function `besselJ()`:

```
beta <- 1
n <- 0:(beta+2)
amp <- besselJ(beta, n)
amp
[1] 0.76519769 0.44005059 0.11490348 0.01956335
```

The amplitude of all frequency peaks and the upper and lower sidebands together with the carrier frequency are simply obtained with:

```
amp <- c(rev(amp), amp[-1])
amp
[1] 0.01956335 0.11490348 0.44005059 0.76519769 0.44005059
[6] 0.11490348 0.01956335
```

The expected frequency spectrum can be plot easily with (Fig. 10.20):

```
freq <- seq(3500 ,6500, by=500)
plot(freq, amp, type="h", col="blue",
     xlab="Frequency", ylab="Amplitude")
```

For values of $\beta > 2.5$, the Bessel function can return negative values that would not correspond to positive frequency coefficients returned by the DFT:

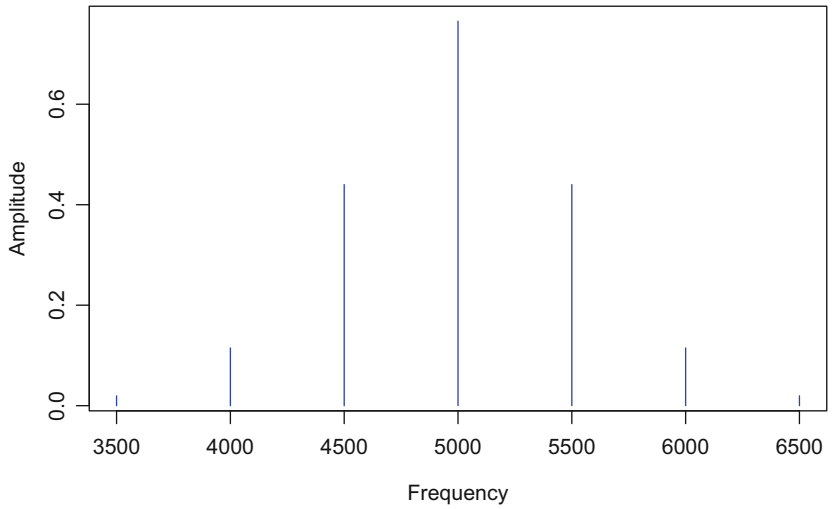


Fig. 10.20 Theoretical frequency spectrum of a FM signal. The spectrum is obtained by applying Carson's rule and Bessel functions to estimate the number, the frequency position, and the relative amplitude of a pure tone sound with a carrier frequency at 5000 Hz and a frequency modulation with a frequency of 500 Hz and a frequency peak deviation of 500 Hz equivalent to a modulation index $\beta = 1$

```
beta <- 4
n <- 0:(beta+2)
amp <- besselJ(beta, n)
amp <- c(rev(amp), amp[-1])
amp
 [1]  0.04908758  0.13208666  0.28112906  0.43017147
 [5]  0.36412815 -0.06604333 -0.39714981 -0.06604333
 [9]  0.36412815  0.43017147  0.28112906  0.13208666
[13]  0.04908758
```

When the frequency bandwidth of the FM is large and/or when the carrier frequency is low, the frequency of the sidebands can be either null or negative. For instance, a 440 Hz pure-tone sound with a FM with $f_{fm} = 200$ and $\Delta f_c = 1600$, that is, $\beta = 1600 \div 200 = 8$, will have $8 + 2 = 10$ lower sidebands at the following frequencies in Hz:

```
fc <- 440
f.fm <- 200
delta.fm <- 1600
```

(continued)

```
beta <- delta.fm / f.fm
lower.sidebands <- fc-(1:round(beta+2))*f.fm
lower.sidebands
 [1]   240    40  -160  -360  -560  -760  -960 -1160 -1360
[10] -1560
upper.sidebands <- fc+(1:round(beta+2))*f.fm
upper.sidebands
 [1]  640  840 1040 1240 1440 1640 1840 2040 2240 2440
```

The negative sidebands reflect around the 0 Hz value and are combined with the sidebands on the positive side of the frequency axis. In the above example, the frequency spectrum shows peaks at the following positions:

```
sort(c(abs(lower.sidebands), fc, upper.sidebands))
 [1]   40  160  240  360  440  560  640  760  840  960 1040
[12] 1160 1240 1360 1440 1560 1640 1840 2040 2240 2440
```

This generates a frequency spectrum with harmonics and inharmonics combined (Fig. 10.19, bottom-left).

Linear and sinusoidal FMs can be combined giving a frequency spectrum with sidebands along a large peak due to the carrier frequency (Fig. 10.19, bottom-right).

10.1.4.4 Aperiodic or Brief Signals

The sounds we explored in the previous sections were all periodics or long requesting, in most cases, few frequency components to be described properly by the Fourier transform. However, aperiodic or brief sounds need more frequency components for a correct description. Figure 10.21 shows that the frequency spectrum widens for a sound with a duration from 0.1 to 0.001 s. An instantaneous sound of 0.0001 s is transferred in the frequency domain by a totally flat spectrum (Fig. 10.21, bottom-right). It is therefore essential to consider the duration of the sound to be studied. Computing a frequency spectrum on a too short sound is meaningless and should be avoided. Other solutions, such as zero crossing (see Sect. 13.1.4.2), should be considered.

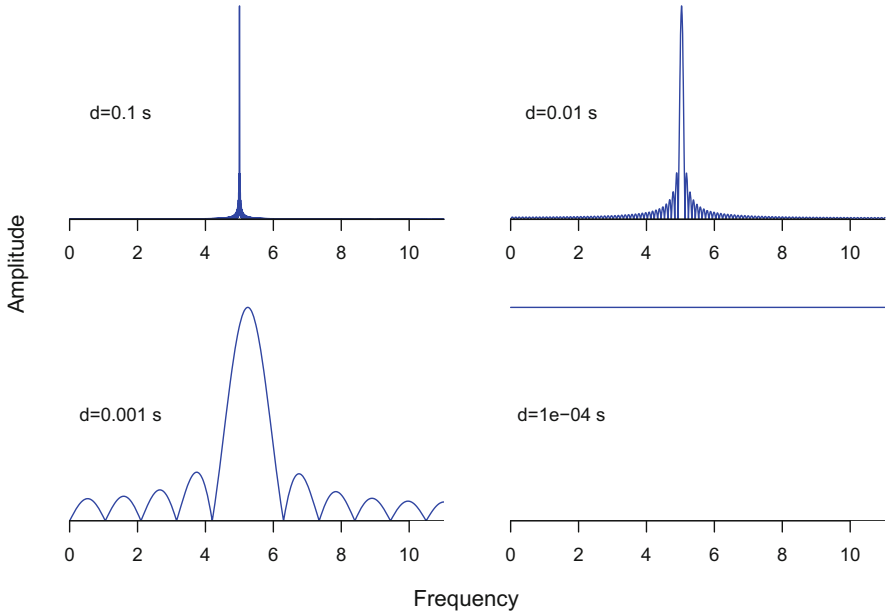


Fig. 10.21 Frequency spectrum shape of brief signals. Frequency spectrum of a pure sine wave with a duration of 0.1, 0.01, 0.001, and 0.0001 s showing the appearance of side lobes that increase in importance up to a totally flat spectrum profile

10.1.5 Symbolic Analysis

10.1.5.1 Symbolic Discretization

A solution to qualify a numeric series $x(n)$ is to translate it into a symbolic series $S(n)$ by converting the numeric values into a finite number of symbols through a discretization process (Cazelles 2004). The idea is to assess the state of the numeric series by classifying each sample value as depending on the shape of the series profile. This assessment can follow three or five predefined states. In the three-state option, each sample value can be considered as an increase (I), a decrease (D), or a flatness (F) state in the series.

The translation from numbers to symbols is ruled out by the following conditions:

$$\begin{aligned} x_n < x_{n+1} &\Rightarrow \text{increase} \Rightarrow S_{n+1} = I \\ x_n > x_{n+1} &\Rightarrow \text{decrease} \Rightarrow S_{n+1} = D \\ x_n = x_{n+1} &\Rightarrow \text{flat} \Rightarrow S_{n+1} = F \end{aligned}$$

where $\{x_n,\ x_{n+1}\}$ are two consecutive sample values of the numeric series $x(n)$ of length N and S_{n+1} is the $n+1$ symbol of the symbol series $S(n)$ of length $N-1$.

In the five-state option, the rules are longer (Cazelles 2004):

$$\begin{aligned}
x_n < x_{n+1} < x_{n+2} &\Rightarrow \text{increase} \Rightarrow S_{n+1} = I \\
x_n < x_{n+2} \leq x_{n+1} &\Rightarrow \text{peak} \Rightarrow S_{n+1} = P \\
x_{n+1} < x_n \leq x_{n+2} &\Rightarrow \text{trough} \Rightarrow S_{n+1} = T \\
x_{n+1} < x_{n+2} \leq x_n &\Rightarrow \text{trough} \Rightarrow S_{n+1} = T \\
x_{n+2} < x_n \leq x_{n+1} &\Rightarrow \text{peak} \Rightarrow S_{n+1} = P \\
x_{n+2} < x_{n+1} \leq x_n &\Rightarrow \text{decrease} \Rightarrow S_{n+1} = D \\
x_n = x_{n+1} = x_{n+2} &\Rightarrow \text{flat} \Rightarrow S_{n+1} = F
\end{aligned}$$

where $\{x_n, x_{n+1}, x_{n+2}\}$ are three consecutive sample values of the numeric series $x(n)$ of length N and S_{n+1} is the $n + 1$ symbol of the symbol series $S(n)$. The length of $S(n)$ is $N - 2$ because the first and the last sample values have a single neighbor so that no state can be assessed for these values. In this case, a plateau—that is, an elevated flat region such as {0, 1, 1, 1, 0}—is considered as a "flat peak" symbolized with the succession $\{P, F, P\}$ with possibly several F framed by a pair of P. However, a plateau could also be seen as an increase, a flat region, and a decrease and therefore encoded as $\{I, F, D\}$. In that case, the set of rules changes to:

$$\begin{aligned}
x_n < x_{n+2} < x_{n+1} &\Rightarrow \text{peak} \Rightarrow S_{n+1} = P \\
x_{n+1} < x_n < x_{n+2} &\Rightarrow \text{trough} \Rightarrow S_{n+1} = T \\
x_n \leq x_{n+1} \leq x_{n+2} &\Rightarrow \text{increase} \Rightarrow S_{n+1} = I \\
x_{n+2} \leq x_{n+1} \leq x_n &\Rightarrow \text{decrease} \Rightarrow S_{n+1} = D \\
x_n = x_{n+1} = x_{n+2} &\Rightarrow \text{flat} \Rightarrow S_{n+1} = F
\end{aligned}$$

As an example, the following numeric series of length 14:

$$x(n) = \{0, 1, 2, 1, 0, 1, 1, 1, 0.5, 0, 0, 0, 1, 0\}$$

is translated into the three-state symbol series of length 13:

$$S(n) = \{I, I, D, D, I, F, F, D, D, F, F, I, D\}$$

or into the five-state symbol series of length 12 for a plateau encoded as $\{P, F, P\}$:

$$S(n) = \{I, P, D, T, \mathbf{P}, \mathbf{F}, \mathbf{P}, D, D, F, I, P\}$$

or into the five-state symbol series of length 12 for a plateau encoded as $\{I, F, D\}$:

$$S(n) = \{I, P, D, T, \mathbf{I}, \mathbf{F}, \mathbf{D}, D, D, F, I, P\}$$

This symbolic discretization is available in the function `discrets()` of `seewave` where the argument `symb` controls the number of symbols to use, either

3 or 5 (default), and the argument `plateau` sets the way a plateau is encoded (1 for the $\{P, F, P\}$ option and 2 for $\{I, F, D\}$ option). Here, the following code repeats the example given just above:

```
x <- c(0, 1, 2, 1, 0, 1, 1, 1, 0.5, 0, 0, 0, 1, 0)
discrets(x, symb=3)
[1] "IIDDIFFDDFFID"
discrets(x, symb=5)
[1] "IPDTPFPDDFIP"
discrets(x, symb=5, plateau=2)
[1] "IPDTIFDDDFIP"
```

Turning the argument `collapse` to `FALSE` separates the symbols of the returned character vector:

```
discrets(x, symb=3, collapse=FALSE)
 [1] "I" "I" "D" "D" "I" "F" "F" "D" "D" "F" "F" "I" "D"
```

The frequency spectrum is a numeric series that can be transformed into a symbolic series using the discretization introduced just above. For instance, the five-state discretization of the frequency spectrum of `peewit` is obtained with `discrets()` and displayed over the spectrum with the low-level plot function `text()`. The first and last frequencies are discarded for the plotting operation to align the numeric and symbolic series (Fig. 10.22):

```
center <- round(duration(peewit)/2,2)
fspec <- spec(peewit, at=center)
res <- discrets(fspec[,2], collapse=FALSE)
head(res)
[1] "I" "P" "T" "P" "D" "T"
text(fspec[-c(1,nrow(fspec)),], labels=res, col="red")
```

This symbolic analysis can help in understanding, describing, and comparing frequency spectra. It also can be useful to compute the entropy of the symbol series using the classical definition of Shannon-Wiener entropy H:

$$H = -\sum_{i=1}^{N-2} (p_i \log p_i)$$

where p_i are the elements of the probability mass function (PMF) of the symbolic series $S(n)$.

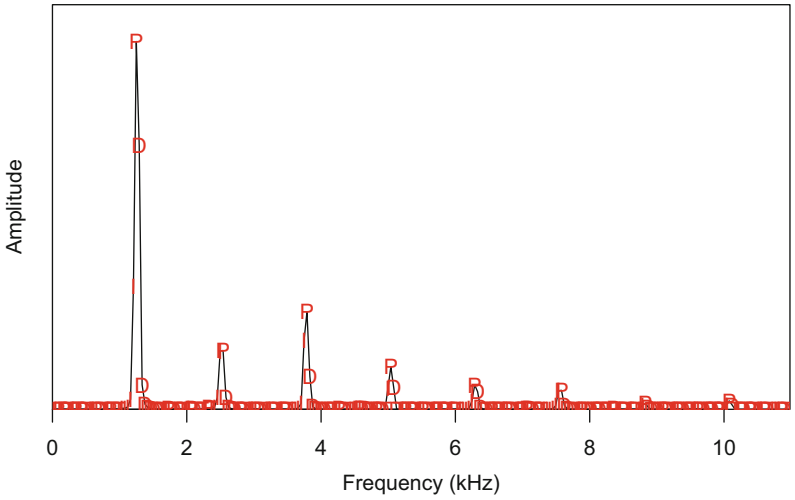


Fig. 10.22 Symbolic analysis. The symbolic analysis consists in translating each amplitude values into a letter according to the shape of the numeric series, here a frequency spectrum of `peewit`

The function `symba()` uses the function `discrets()` and returns the symbolic series (`$s1`) of an object with as well the relative frequency (proportion) of each symbol (`$freq`) and the entropy of the series (`$h1`):

```
symba(fspec[,2], collapse=FALSE)
$s1
  [1] "I" "P" "T" "P" "D" "T" "I" "P" "T" "P" "D" "T" "I"
 [14] "I" "P" "D" "D" "T" "I" "P" "T" "P" "T" "I" "I" "I"
 [27] "I" "I" "P" "D" "D" "D" "D" "D" "D" "D" "D" "T" "I"
 [40] "P" "D" "D" "T" "P" "T" "I" "I" "P" "D" "T" "P" "T"
 [53] "I" "P" "T" "I" "I" "I" "P" "D" "D" "D" "D" "T" "P"
 [66] "D" "T" "I" "I" "P" "D" "D" "T" "I" "I" "P" "D" "T"
 [79] "I" "P" "D" "T" "I" "I" "I" "I" "I" "P" "D" "D" "D"
 [92] "D" "D" "D" "T" "P" "T" "I" "P" "D" "D" "T" "I" "I"
[105] "I" "P" "D" "D" "D" "D" "T" "P" "D" "D" "T" "I" "P"
[118] "D" "D" "T" "I" "P" "T" "P" "D" "T" "I" "P" "D" "D"
[131] "T" "I" "P" "T" "P" "D" "T" "I" "P" "T" "I" "I" "I"
[144] "I" "I" "P" "D" "D" "D" "D" "T" "P" "T" "P" "T" "I"
[157] "P" "D" "D" "T" "I" "P" "T" "P" "D" "D" "T" "P" "T"
[170] "I" "I" "I" "I" "I" "I" "P" "D" "D" "D" "D" "T" "I"
[183] "P" "D" "D" "D" "T" "P" "D" "T" "P" "T" "I" "P" "T"
[196] "I" "P" "D" "T" "I" "P" "T" "I" "I" "P" "D" "D" "D"
[209] "D" "T" "P" "D" "D" "T" "I" "P" "D" "D" "T" "P" "T"
[222] "I" "P" "D" "D" "T" "P" "D" "T" "P" "D" "T" "I" "P"
[235] "D" "T" "I" "P" "D" "T" "P" "D" "T" "I" "I" "P" "D"
[248] "T" "I" "I" "I" "P" "D" "T"
```

(continued)

```
$freq1
s1
        D         I         P         T
0.3070866 0.2519685 0.2204724 0.2204724

$h1
[1] 1.376582
```

Rounding the values of the frequency spectrum can change importantly the results as the number of flat parts (F) may increase significantly:

```
symba(round(fspec[,2],2), collapse=FALSE)
$s1
  [1] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
 [14] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "I" "I"
 [27] "I" "I" "P" "D" "D" "D" "P" "D" "F" "F" "F" "F" "F"
 [40] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
 [53] "F" "F" "F" "I" "I" "P" "P" "D" "D" "F" "F" "F" "F"
 [66] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
 [79] "F" "F" "F" "F" "F" "F" "I" "I" "I" "P" "D" "D" "D"
 [92] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[105] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "I" "I" "P"
[118] "D" "D" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[131] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[144] "I" "I" "P" "D" "D" "F" "F" "F" "F" "F" "F" "F" "F"
[157] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[170] "F" "F" "F" "I" "I" "P" "P" "D" "D" "F" "F" "F" "F"
[183] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[196] "F" "F" "F" "F" "F" "F" "F" "I" "I" "P" "D" "D" "F"
[209] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[222] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "I" "I" "P"
[235] "D" "D" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
[248] "F" "F" "F" "F" "F" "F" "F"

$freq1
s1
         D          F          I          P
0.07480315 0.80708661 0.07480315 0.04330709

$h1
[1] 0.6968517
```

10.1.5.2 Symbolic Aggregate Approximation

Symbolic aggregate approximation, abbreviated SAX, is another mean to convert a numerical series into a symbolic series (Lin et al. 2003). SAX has been used for search of similarity in animal behavior for insect duet analysis (Aldersley et al. 2016) and in soundscape ecology (Kasten et al. 2012). SAX proceeds to a reduction of dimensionality through another process named piecewise aggregate approximation (PAA) so that a long numeric series can be translated into a short series of symbols also known as SAX word. PAA mainly consists in a normalization and in a reduction of dimension by defining segments (syn. frames, windows). The three main steps of PAA are (see DIY box 10.2 for a manual encoding):

1. The original numeric series $s[n]$ of length N is Z-normalized, so that each numeric value is transformed into the following:

$$z[n] = \frac{s[n] - \mu}{\sigma}$$

   with μ is the arithmetic mean, and σ the standard deviation of the original numeric series $s[n]$,
2. The numeric series $z[n]$ is segmented into $w \leq N$ equal sized segments,
3. The mean of each segment is computed.

DIY 10.2 — How to code the piecewise aggregate approximation (PAA)

Here is a way to encode the PAA with a test on a numeric series of length $N = 120$ extracted from the time series `sunspots` of the package `datasets` that provides the monthly mean relative sunspot numbers from 1749 to 1983. This numeric series, corresponding to 10 years of observations, is divided in $w = 8$ segments of length $N \div w = 16$:

```
N <- 120                  # length of the numeric series
s <- sunspots[1:N]        # first 120 data from 'sunspots'
z <- (s-mean(s))/sd(s)    # step 1: Z normalisation
w <- 8                    # step 2: length of the segments
pos <- seq(0, N, by=N/w) # position limits of each segment
pos
[1]   0  15  30  45  60  75  90 105 120
PAA <- numeric(w)         # step 3
for(i in 1:w){            # computation of segment average
  PAA[i] <- mean(z[(pos[i]+1):pos[i+1]])
}
PAA                       # result
[1]  1.3936404  1.1162006  0.2093451 -0.2558151 -0.9866827
[6] -1.0520168 -0.6505802  0.2259087
```

The next procedure, which is the core of SAX, is to translate the averaged values obtained with the PAA into a finite number of symbols (letters). This conversion is achieved by attributing with equiprobability each value of the PAA sequence to a

letter in reference to a Gaussian distribution of mean 0 and standard deviation 1. The process assumes that the distribution of PAA follows a Gaussian distribution, a condition usually met as normalized time series tends to a Gaussian distribution. The Gaussian distribution is divided into α quantiles, where α is the number of symbols, or PAA alphabet size, set by the user (Fig. 10.23).

The package `jmotif` is dedicated to SAX (Senin 2015); however, `seewave` contains also a function, named `SAX()`, that can transform any numeric series into a letter series given a certain alphabet size (argument `alphabet_size`) and a PAA number of segments (argument `PAA_number`).

A basic example of `SAX()` is given here for a simple numeric series:

```
x <- c(-1, 0, 1, 2, 1, 2, 3, 4,
       2, 1, 0, -1, -1, -1, 0, 0)
SAX(x, alphabet_size=3, PAA_number=4)
[1] "b" "c" "b" "a"
```

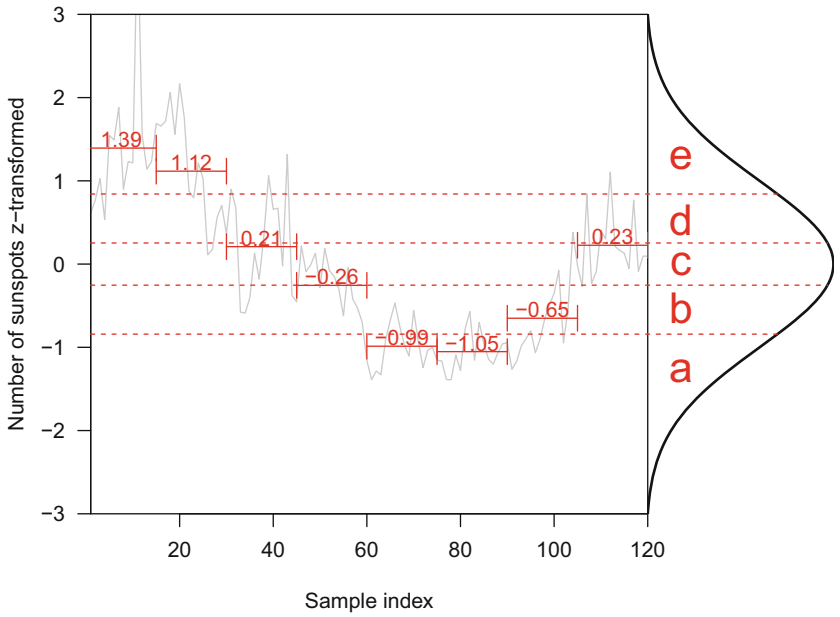


Fig. 10.23 SAX principle. The figure shows how the *Z*-transformed data are converted into letters in reference to a Gaussian distribution. The data come from the example given in the DIY box 10.2. The SAX series of symbols, or word, would be here *eecbaabc*. They correspond to monthly number of sun spots from 1750 to 1760. Inspired from Lin et al. (2003)

`SAX()` therefore returns a series of letters that can be collapsed into different formats using the argument `collapse`:

```
SAX(x, alphabet_size=3, PAA_number=4, collapse="")
[1] "bcba"
SAX(x, alphabet_size=3, PAA_number=4, collapse="-")
[1] "b-c-b-a"
```

`SAX()` can be used to characterize a spectrum. The five-letter SAX code of the frequency spectrum of `peewit` divided in ten segments by the PAA is:

```
fspec <- spec(peewit, at=duration(peewit)/2, plot=FALSE)
SAX(fspec[,2], alphabet_size=5, PAA_number=10)
 [1] "c" "e" "c" "c" "c" "c" "c" "c" "c" "c"
```

To relax the constraints of normality, the function `SAX()` offers the possibility to directly work on the quantiles or any particular breakpoints of the original data distribution rather than on the quantiles of a Gaussian distribution. This is achieved with the argument `breakpoints`:

```
SAX(fspec[,2], alphabet_size=5, PAA_number=10,
    breakpoints="quantiles")
 [1] "c" "e" "e" "e" "e" "e" "e" "d" "d" "e"
SAX(fspec[,2], alphabet_size=5, PAA_number=10,
    breakpoints=c(0, 0.1, 0.9, 1))
 [1] "b" "b" "b" "b" "b" "b" "b" "b" "b" "b"
```

If the input is a `Wave` object, then `SAX()` computes a soundscape spectrum (see Sect. 11.15) and applies the SAX transformation to this spectrum following the procedure described in Kasten et al. (2012).

10.1.6 Parametrization

We have seen that the frequency spectrum can be parametrized by determining the frequency and the relative amplitude of the profile peaks. However other features can be obtained to characterize the frequency spectrum. In the two following sections, we detail quality and statistical parameters.

10.1.6.1 Quality Factor Q

As detailed in Bennet-Clark (1999), the quality factor, Q, describes the properties of damped resonant or oscillatory systems. In other words, the Q factor is a dimensionless measurement of tuning sharpness or of the sharpness of a resonant frequency f_r. The measurement can be processed either in the time or in the frequency domain after a DFT computation. In the latter case, the computation consists in dividing the resonant frequency f_r by the frequency bandwidth found at x dB below the same resonant frequency f_r:

$$Q_{-x\,\mathrm{dB}} = \frac{f_r}{\Delta_{-x\,\mathrm{dB}} f}$$

Usually, the frequency bandwidth is set at $x = -3$ dB, but other dB levels, like -10 dB, can be used depending on the application. Because sharpness increases as $\Delta_{-x\,\mathrm{dB}} f$ decreases, the factor Q increases with sharpness. When reporting Q values, it is of course necessary to keep the same $-x$dB value for each computation but it is also essential to specify the value of the resonant frequency because the Q of resonating systems with similar sharpness but different frequencies differ. For instance, different Q values are obtained for systems resonating, respectively, at 4000 and 5000 Hz with a similar bandwidth of 300 Hz:

```
fr <- c(4000, 5000)  # resonant frequency
bandwidth <- 300     # frequency bandwidth
Q <- fr/bandwidth    # Q
Q
[1] 13.33333 16.66667
```

At the opposite, two resonating systems with different resonant frequency and different sharpness might return similar Q values. In the following, the systems have, respectively, a resonant frequency of 4000 and 5000 Hz with a frequency bandwidth of 300 and 375 Hz.

```
fr <- c(4000,5000)        # resonant frequency
bandwidth <- c(300,375)   # frequency bandwidth
Q <- fr/bandwidth         # Q
Q
[1] 13.33333 13.33333
```

It is therefore highly advised to undertake a comparison between Q factors only for systems resonating within the same frequency range. If a system resonates at different frequencies, due to the combined action of different resonators, the Q factor can be computed for each resonant frequency. However, in most cases, only the resonating frequency with the maximal amplitude is considered.

The function `Q()` of `seewave` computes and displays the resonance quality factor of a frequency spectrum considering the dominant frequency only. In other words, this function estimates the frequency sharpness by (1) determining the dominant frequency of a dB frequency spectrum, (2) estimating the frequencies found at a lower fixed dB level (by default -3 dB), and (3) computing the ratio between f_r and the difference between the $-x$ dB frequencies. As the function `Q()` requires a frequency spectrum computed along a dB scale, the computation of Q requires two lines of code.

To test the function `Q()`, we need to refer to a sound driven by a resonator. The Italian tree cricket, *Oecanthus pellucens*, produces sound by wing stridulation, that is, by rubbing the plectrum of one wing against the file of the other (Fig. 10.1). A peculiar area of the wing, named the "harp", acts as a resonator amplifying the carrier frequency generated by the plectrum and the file. The energy of the Italian tree cricket stridulation is mainly concentrated around 2000 Hz. The data `pellucens` attached to `seewave` is made of the stridulations of an individual recorded in the west of France with a sampling frequency f_s of 11,025 Hz:

```
data(pellucens)
pellucens

Wave Object
Number of Samples:      36476
Duration (seconds):     3.31
Samplingrate (Hertz):   11025
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The following code estimates and plots the Q factor for a 1024 sample frequency spectrum computed at 1 s of `pellucens`, in the middle of the first stridulation (Fig. 10.24):

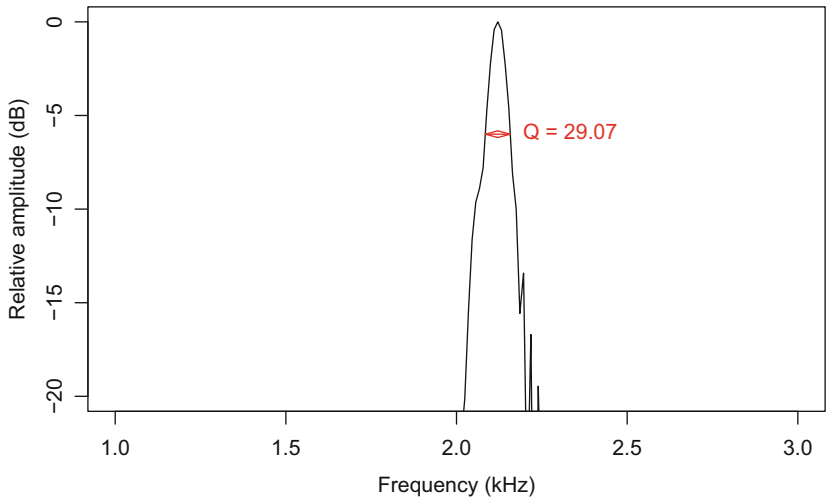


Fig. 10.24 Resonance quality factor Q. The $Q_{-6\,\mathrm{dB}}$ factor of `pellucens` was computed with a dB frequency spectrum over 1024 samples at the position 1s. Specifying axis limits allows to zoom in around the frequency peak where Q is computed

```
fspec <- spec(pellucens, at=1, wl=1024, dB="max0", plot=FALSE)
Q(fspec, level=-6, xlim=c(1, 3), ylim=c(-20,0))
```

```
$Q
       x
29.0701

$dfreq
        x
2.121026

$fmin
        x
2.084448

$fmax
       x
2.15741

$bdw
          x
0.07296244
```

The value of `Q()` returns not only the quality factor (`$Q`) but as well the dominant frequency (`$dfreq`), the minimum frequency (`$fmin`), the maximum frequency (`$fmax`), and the frequency bandwidth (`$bdw`) at the $-x$ dB level.

The size of the DFT can modify the Q values as increasing the DFT increases the frequency resolution of the spectrum and may therefore widen the spectrum if there are amplitude and/or frequency modulations occurring within the length of the DFT. The following code proves that varying DFT length from 64 to 1024 samples changes the Q value of the `pellucens` frequency spectrum:

```
wl <- 2^(6:10)
wl
[1]   64  128  256  512 1024
n <- length(wl)
res <- numeric(n)
for(i in 1:n) {
    fspec <- spec(pellucens, at=1, wl=wl[i],
                  plot=FALSE, dB="max0")
    res[i] <- Q(fspec, plot=FALSE)$Q
}
res
[1]  8.893406 15.818978 28.741961 38.074497 43.003486
```

Eventually, the window function applied to the DFT can also change the shape of the frequency spectrum and hence the value of the Q factor. The next test returns the value of Q with six different Fourier windows:

```
wn <- c("bartlett", "blackman", "flattop",
        "hamming", "hanning", "rectangle")
n <- length(wn)
res <- numeric(n)
for(i in 1:n) {
    fspec <- spec(pellucens, at=1, wl=512, wn=wn[i],
                  plot=FALSE, dB="max0")
    res[i] <- Q(fspec, plot=FALSE)$Q
}
res
[1] 39.03372 36.40836 30.46242 38.45022 38.07450 44.69399
```

The description of the shape of the spectrum is as well crucial to understand if the dominant frequency results or not of a resonating system. As detailed above (see Sect. 10.1.4), the width of a frequency band can result of other processes than resonance as of FM or aperiodic signals.

10.1.6.2 Roughness and Rugosity

The roughness and rugosity can be both considered as a measure of the noise occurring in the frequency spectrum. The roughness, obtained with the function `roughness()` of `seewave`, corresponds to the total curvature of a curve. The spectral roughness is the integrated squared second derivative of the spectrum $s(f)$ (Ramsay and Silverman 2005):

$$\text{roughness} = \sum_{i=1}^{n} \ddot{s}(f)^2$$

Closely related, the rugosity defined in `seewave` under the function `rugo()` is based on the first derivative. The formula used was taken from Mezquida (2009, p.826) slightly modified to fit with the classical definition of the root-mean-square:

$$\text{rugosity} = \sqrt{\left(\frac{1}{n}\sum_{i=1}^{n} \dot{s}(f)^2\right)}$$

An example of the use of `roughness()` and `rugo()` is given in the DIY box 10.3.

DIY 10.3 — How to compute several spectral features on several sounds

The following code shows how to compute six spectral features on four sounds (`tico`, `orni`, `peewit`, and `sheep`). The main trick consists first in assigning the four frequency spectra to a single `R list`:

```
data <- list(tico.s=spec(tico, at=1.1, plot=FALSE),
             orni.s=spec(orni, at=0.36, plot=FALSE),
             peewit.s=spec(peewit, at=0.35, plot=FALSE),
             sheep.s=spec(sheep, at=1.25, plot=FALSE))
l <- length(data)
```

Then a homemade function is created to compute successively six features (roughness, rugosity, spectral flatness, spectral entropy or evenness, (Gini-)Simpson entropy, and Rényi entropy of order 2):

```
parameters <- function(x){
    return(c(roughness(x), rugo(x), sfm(x),
             sh(x), sh(x, alpha="simpson"),
             sh(x, alpha=2)))
}
```

(continued)

DIY 10.3 (continued)

Eventually, the results are obtained by writing a `for` loop:

```
# preparation of the data.frame
# where the results will be stored
num <- numeric(l)
res <- data.frame(roughness=num, rugo=num,
                  sfm=num, shannon=num,
                  simpson=num, renyi=num,
                  row.names=names(data))
# use of the new function on each spectrum
for(i in 1:l) res[i,] <- parameters(data[[i]])
res
          roughness      rugo         sfm   shannon
tico.s    0.6010492 0.4880328 0.005607601 0.4696611
orni.s    3.6233291 0.4900904 0.293239059 0.8433729
peewit.s  1.8082739 0.4891119 0.036838594 0.4962191
sheep.s   2.3560042 0.1816416 0.244637921 0.7565844
            simpson     renyi
tico.s    0.8734768 0.3728157
orni.s    0.9844235 0.7505607
peewit.s  0.8742644 0.3739418
sheep.s   0.9650854 0.6050030
```

10.1.6.3 Flatness and Evenness (Entropy)

The spectral flatness measure, also known as the Wiener entropy, is calculated by the function `sfm()` of `seewave` as the ratio between the geometric mean and the arithmetic mean of the n frequency bins f_i of the frequency spectrum:

$$\text{sfm} = n \times \frac{\sqrt[n]{\prod_{i=1}^{n} f_i}}{\sum_{i=1}^{n} f_i}$$

Evenness, or equitability, is the entropy divided by its maximum entropy. Spectral evenness is computed with the `seewave` function `sh()`. This function computes by default the Shannon evenness but can also return the (Gini-)Simpson and the Rényi entropies by setting adequately the argument `alpha`. The Shannon spectral evenness obeys to the following equation:

$$S_f = -\frac{\sum_{i=1}^{n} f_i \log f_i}{\log(n)}$$

S corresponds to the acoustic index H_f (see Sect. 16.1).

The Simpson, or Gini-Simpson, spectral entropy is computed according to:

$$GS = 1 - \sum_{i=1}^{n} f_i^2$$

The Rényi spectral entropy of order alpha is obtained with:

$$R = \frac{1}{1-\alpha} \times \log\left(\sum_{i=1}^{n} f_i^{\alpha}\right)$$

with $\alpha >= 0$ and $\alpha \neq 1$.

Examples of `sfm()` and `sh()` are provided in the DIY box 10.3.

10.1.6.4 Statistic Parameters

The frequency spectrum is a histogram in which cell breakpoints are determined by the sampling frequency of the original sound and the frequency resolution of the Fourier transform ($\Delta_f = f_s \div wl$). The frequency spectrum can be therefore considered as a probability function that can be parametrized with usual summary statistics of central tendency (mean, median, mode, centroid), dispersion (standard deviation, standard error, quartiles), and shape (skewness, kurtosis, flatness, entropy). These parameters can be obtained with the function `specprop()` of `seewave`.[1] The values returned by `specprop()` are:

- `$mean`: mean frequency
- `$sd`: standard deviation of the mean
- `$sem`: standard error of the mean
- `$median`: median frequency
- `$mode`: mode frequency corresponding to the dominant frequency
- `$Q25`: first quartile
- `$Q75`: third quartile
- `$IQR`: interquartile range, with $IQR = Q75 - Q25$
- `$cent`: centroid, computed according to $C = \sum_{i=1}^{N}(f_i \times ai)$ with a_i the relative amplitude of the N frequency f_i.
- `$skewness`: skewness, as defined in Sect. 7.1, is a measure of asymmetry computed with $\gamma_1 = \mu_3 \div \mu_2^{3/2}$ where μ_2 and μ_3 are, respectively, the second and the third central moments. A value of $\gamma_1 < 0$ indicates that the spectrum is skewed to left, $\gamma_1 = 0$ indicates that the spectrum is symmetric, and $\gamma_1 > 0$ indicates that the spectrum is skewed to right. The asymmetry increases with $|\gamma_1|$.

[1] A derived version of `specprop()` is available in the package `warbleR` under the name `specan()`.

- `$kurtosis`: kurtosis, as defined in Sect. 7.1, is a measure of peakedness computed with $\beta_2 = \mu_4 \div \mu_2^2$ where μ_2 and μ_4 are, respectively, the second and the fourth central moments. A value of $\beta_2 < 3$ is obtained when the spectrum is platykurtic, meaning that it has fewer items at the center and at the tails than the normal curve but has more items in the shoulders; $\beta_2 = 3$ is obtained when the spectrum shows a normal shape; $\beta_2 > 3$ is obtained when the frequency spectrum is leptokurtic, meaning that it has more items near the center and at the tails, with fewer items in the shoulders relative to normal distribution with the same mean and variance.
- `$sfm`: spectral flatness measure, as computed by the function `sfm()`
- `$sh`: spectral evenness as computed by the function `sh()`
- `$prec`: frequency precision or resolution Δ_f of the spectrum

Some of these parameters can be visualized using the argument `plot`. Unusually, this argument waits either a logical value (`TRUE` or `FALSE`) or a numeric value (1 or 2). A value of 1 returns a usual frequency spectrum and a value of 2 returns the cumulative distribution function of the frequency spectrum. The function has to be fed with a frequency spectrum, not with a sound. Here is an example with a frequency spectrum computed for a segment of `orni` (Fig. 10.25):

```
fspec <- spec(orni, at=0.36, wl=512, plot=FALSE)
specprop(fspec, plot=2)
```

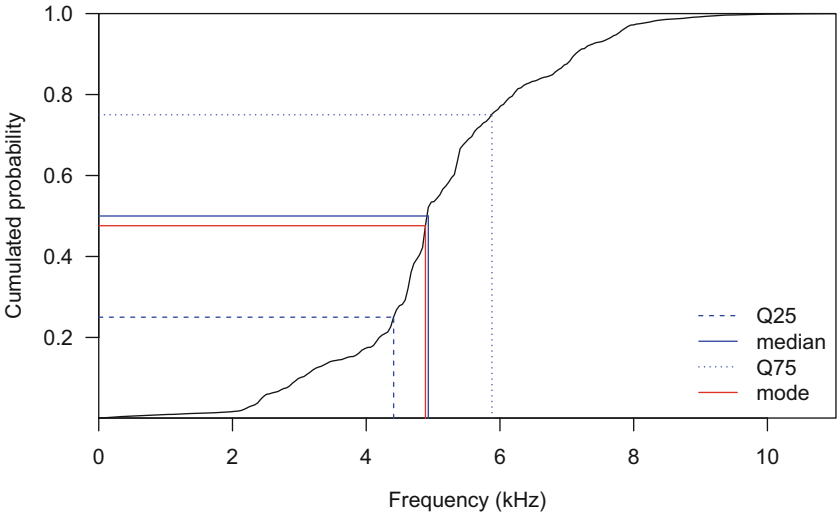


Fig. 10.25 Statistic parameters of the frequency spectrum. The frequency spectrum of a segment of `orni` is here displayed as cumulative distribution function by setting `plot=2`

10.2 Quefrency Cepstrum

In `seewave`, the function `ceps()` computes and optionally plots the quefrency cepstrum. The function works basically as `spec()` does with arguments `at` and `wl` to compute the cepstrum at a particular time position over a specific width. The arguments `from` and `to` can also be called to limit the analysis between two time limits. By default, the cepstrum is computed discarding the phase component of the original wave; however, this can be changed by setting the argument `phase` to `TRUE`. The following code computes and displays the cepstrum in the middle of `peewit` over 512 samples (default value of `wl`) (Fig. 10.26):

```
center <-  round(duration(peewit)/2, 2)
```

Particular points of the cepstrum can be manually identified with either the argument `tidentify` or the argument `fidentify` set to `TRUE` to get results in the time or frequency domain, respectively. In addition, the function `fpeaks()` dedicated to frequency peaks of a spectrum can be recycled to determine the quefrency peaks of a cepstrum in a similar way than with a frequency spectrum. It is first necessary to compute the cepstrum and then to give it as an input to `fpeaks()`. In the following case, the number of decimal places is increased to

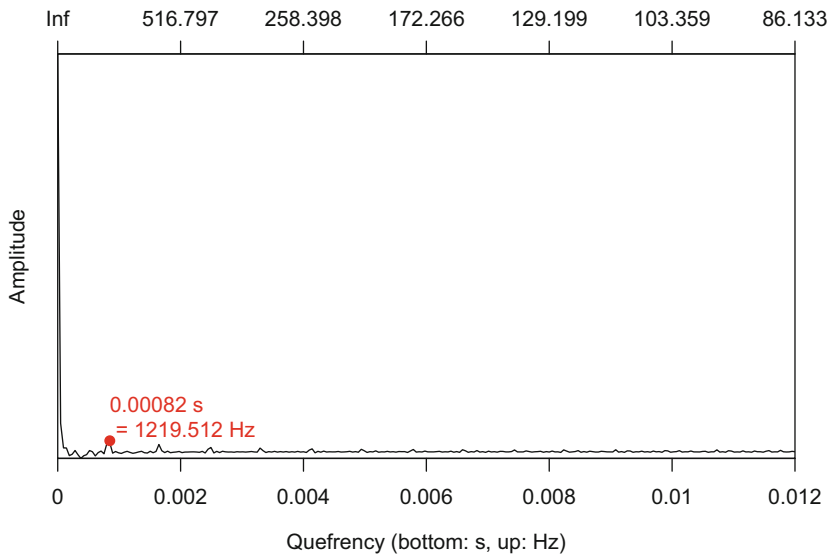


Fig. 10.26 Quefrency cepstrum. The first rahmonic, or quefrency peak, was estimated by using the argument `tidentify` and then highlighted with `points`. The graph has two *x* scale, one at the bottom expressed in time (s) and the other (top) expressed in Hz

4, the default value being 2, to see relevant values on the graphic (Fig. 10.26). Then, the first column of the result should be interpreted as quefrency in s so that an estimation of the peak in Hz is obtained by computing the inverse of the result:

```
qceps <- ceps(peewit, at=center,  # cepstrum
              plot=FALSE)
res <- fpeaks(qceps, nmax=1,      # peaks detection
              plot=FALSE)
res                               # results in quefrency in s
             [,1]     [,2]
[1,] 0.0008195278 116.3069
res[,1] <- 1/res[,1]              # results in quefrency in Hz
res
          [,1]     [,2]
[1,] 1220.215 116.3069
```

Both methods indicate that the fundamental frequency at the center of `peewit` is at 1220 Hz.

Finally, the function `ceps()` includes a list of basic graphic arguments to control for the color (`col`), size (`cex`), labels (`qlab`, `alab`), and axis limits (`qlim`, `alim`).

10.3 Phase Portrait

As defined in Sect. 2.2.4, phase (φ) is the horizontal translation of a cyclic function in respect with time. For a sine sound, phase is therefore a temporal translation of $sin(t)$ in respect with t. Phase is often used to diagnose the state of an active system, in particular to assess whether the output of the system shows irregularity and unpredictability suggesting nonlinear phenomenon due to either random input or deterministic chaos (Lauterborn and Parlitz 1988; Kantz and Schreiber 2003). Nonlinearity has been reported several times in animal sounds, such as fish and mammal vocalizations (Tokuda 2017).

The detection of nonlinear acoustics can be conducted by producing phase-space graphics. A first solution consists in displaying the progression of the signal $s(t)$ through time in a (x, y) or (x, y, z) space as a function of its first, second, and possibly third derivatives (see examples in Rice et al. (2011)). A second solution, more classical and known under the name of so-called phase portrait, consists in displaying the original signal with a delayed version of itself, that is, to compare each state of $s(t)$ and $s(t + \tau)$ with τ a fixed delay (see examples in Fitch et al. (2002)). Both types of graphics show a periodic structure for linear signals and an aperiodic structure for nonlinear signals as illustrated in Fig. 10.27. These graphics are available with the `seewave` functions `phaseplot()` and `phaseplot2()`, respectively.

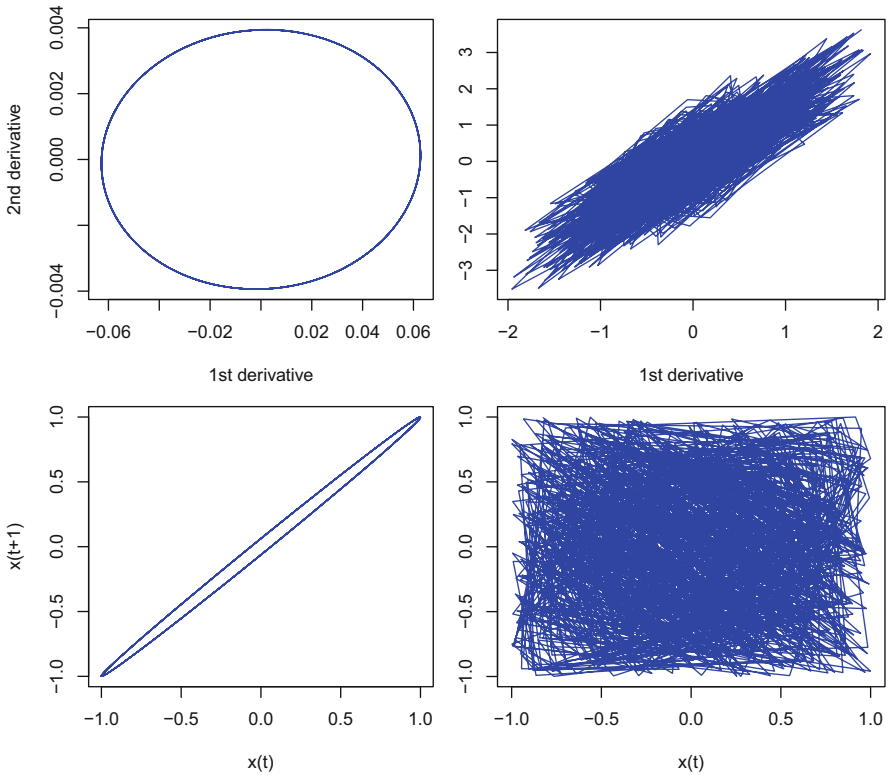


Fig. 10.27 Phase-space plots of pure tone and noise. The figure shows the phase-space plots obtained with `phaseplot()` (top) and `phaseplot2()` (bottom) applied to a pure tone (left) and to noise (right). Pure tone has a periodic shape when noise has an unstructured an aperiodic shape

Gilbert et al. (2014) drew a parallel between the production of brassy sounds in musical instruments and the sound emitted by African elephants (*Loxodonta africana*). The authors recorded the sound produced by a 3 m hose pipe fitted with a trombone mouthpiece and compared it with the trumpet calls of a 20-year-old female elephant. The sound of the hose pipe could be significantly distorted when produced at high amplitude. This generated a "brassy" sound which properties were mainly due to nonlinear propagation of the wave in the pipe. Similarly, the elephant female was able to produce "brassy" trumpet calls probably involving nonlinear propagation as well. Samples of these sounds (see Gilbert et al. 2014, figures 2 and 3) are available in the file `"Loxodonta_africana.wav"` imported into `R` with:

```
elephant <- readWave("sample/Loxodonta_africana.wav")
elephant

Wave Object
Number of Samples:      366706
Duration (seconds):     8.32
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

We need to edit the file by selecting the frequencies between 0 and 6000 Hz where most of the sound energy lays. To achieved this selection, we use the function `fir()`, a frequency filter which principle and use are detailed in Sect. 14.6.

```
output <- "Wave"
elephant <- fir(elephant, from=0, to=6000, output=output)
```

We then select the sounds of interest using the edition function `cutw()`. Each sound lasts 0.3 s to make the phase plots comparable.

```
x <- 0.3
# linear pipe sound
pipe1 <- cutw(elephant, from=0.30, to=0.30+x, output=output)
# non linear pipe sound
pipe2 <- cutw(elephant, from=4.55, to=4.55+x, output=output)
# linear elephant call
elephant1 <- cutw(elephant, from=6.20, to=6.20+x, output=output)
# non linear elephant call
elephant2 <- cutw(elephant, from=7.80, to=7.80+x, output=output)
```

The four sounds are then stored in a single list:

```
sounds <- list(pipe1, pipe2, elephant1, elephant2)
```

The function `phaseplot()` of `seewave` displays either a 2D or 3D phase portrait. Its use is rather easy, the most important argument being the input (first argument `wave`) and the third argument (`dim`) that controls the number of derivatives to compute and to plot (either 2 or 3). The 3D plot is produced thanks to the library `rgl` (Adler and Murdoch 2016). The following code was used to produce the Fig. 10.28:

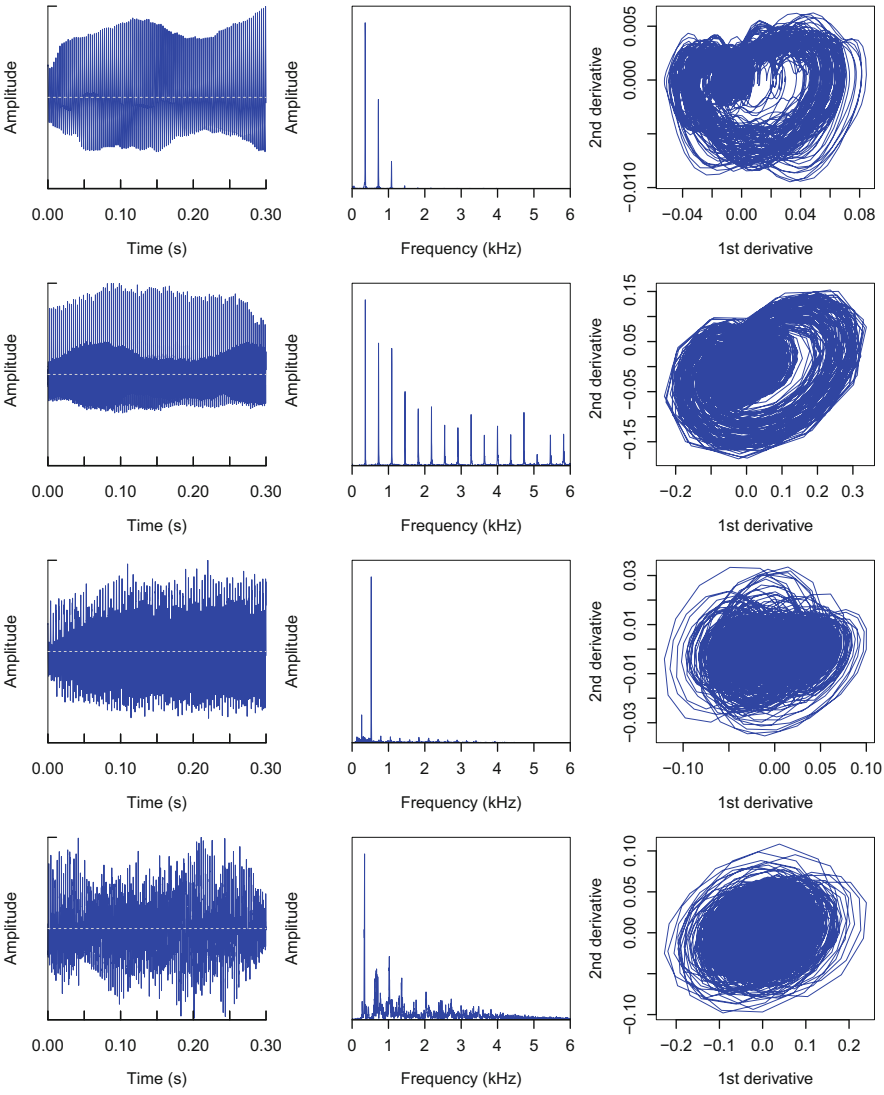


Fig. 10.28 Phase portrait of pipe and elephant sounds. Oscillogram, frequency spectrum, and phase portrait of (from top to bottom) a pipe sound (line 1), a "brassy" pipe sound (line 2), an elephant trumpet call (line 3), and a "brassy" elephant trumpet call (line 4)

```
flim <- c(0,6)         # 0-6 kHz frequency selection
col <- "blue"          # color line
par(mfrow=c(4,3),      # 4*3 figure plate organisation
    mar=c(4.5,4,1,1), # margins
    lwd=0.5)           # line width for all graphics
# 'for' loop to plot successively
# the oscillogram, the frequency spectrum and the phase portrait
# of each sound stored in the list 'sounds'
for(i in 1:length(sounds)){
       oscillo(sounds[[i]], colwave=col, cexlab=0.7)
       spec(sounds[[i]], flim=flim, col=col)
       phaseplot(sounds[[i]], dim=2, col=col)
}
```

The complexity of the phase portrait, that is, the nonlinearity of the sound, increases from the non-"brassy" pipe sound to the "brassy" elephant trumpet call.

Chapter 11
Spectrographic Visualization

So far, we considered separately the three main dimensions of sound, namely, amplitude (see Chap. 7), time (see Chap. 8), and frequency (see Chap. 10). However, amplitude and frequency are rarely invariant with time: sound can be affected by independent amplitude and/or frequency modulations. The time dynamics of sound are neither accessible on an oscillogram nor on a frequency spectrum or a cepstrum such that a new representation of sound should be used to visualize the variations of amplitude and frequency according to time. In this chapter, we will discover a first time $\times$ frequency $\times$ amplitude solution, the short-time Fourier transform, used to produce the spectrogram.

11.1 Short-Time Fourier Transform

11.1.1 Principle

The principle of the short-time Fourier transform (or short-term Fourier transform, abbreviated STFT or STDFT for its discrete version) is rather simple: instead of computing the discrete Fourier transform (DFT) on the complete sound, the DFT is computed on successive sections or windows of the sound. The windows all have the same duration so that the process can also be viewed as sliding a single window along the sound. The DFT is computed through the FFT algorithm at each slide or jump. The short-time Fourier transform operates therefore a kind of time discretization.

A good way to understand how STDFT works is to use the interactive function `dynspec()`. The function computes a STDFT and displays the successive DFTs. The user can navigate from the beginning to the end of the signal with a sliding control button. At each jump of the Fourier window, the frequency spectrum is plotted enlightening the spectral dynamics of the sound. An example with the dataset

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_11

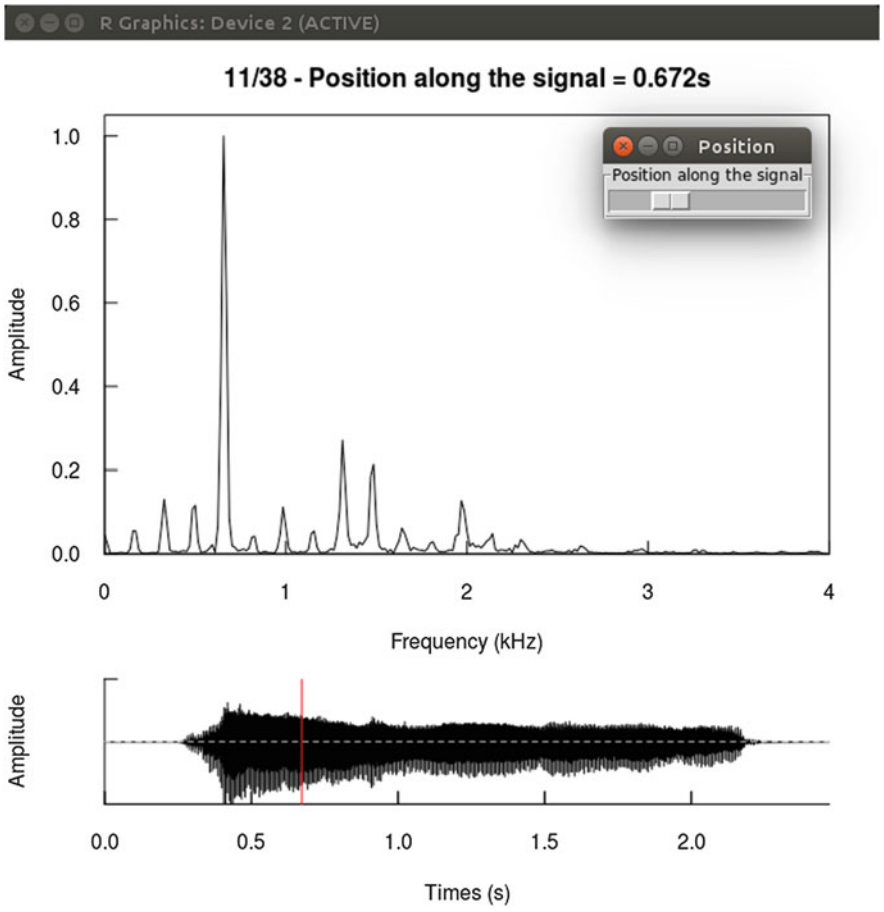


Fig. 11.1 Illustration of the short-time discrete Fourier transform. The function `dynspec()` can be used to better understand the principle of the short-time discrete Fourier transform. A series of frequency spectra are computed along the signal, here the dataset `sheep`, for a given Fourier window. The screenshot here shows the frequency spectrum computed for the eleventh window located at 0.672 s along the sound. The Fourier window has a length of 512 samples and is tapered by a Hanning window (default values of the arguments `wl` and `wn` respectively). Moving along the signal is made possible, thanks to the small control pop-up window entitled "Position." Operating system: Ubuntu

`sheep` is provided with the following code which starts by the loading of the library `rpanel` used to build the control button (Fig. 11.1):

```
library(rpanel)
dynspec(sheep, osc=TRUE)
```

Mathematically, the STFT consists in multiplying the signal $s(t)$ with a function $w(t)$ which is not null for only a short period of time, a process that we already described when introducing the window functions that limit spectral leakage (see Sect. 9.6). The multiplication is applied for each sliding step or jump of the window. The window function can be a rectangle, a Bartlett, a blackman, a flattop, a Hanning(Hann), or a Hamming window.

Knowing that the Fourier transform is written as:

$$F(\omega) = \int_{-\infty}^{\infty} s(t) e^{-i\omega t} \, \mathrm{dt}$$

the equation of the STFT is:

$$STFT\{s(t)\} = F(\tau, \omega) = \int_{-\infty}^{\infty} s(t) w(t - \tau) e^{-i\omega t} \, \mathrm{dt}$$

where τ is the time index used to slide the function window $w(t)$ along the signal. For a finite sound made of N samples taking successive values $s[n]$, the time index is m, and the discrete Fourier transform (DFT) is obtained with:

$$F(\omega) = \sum_{n=0}^{\infty} s[n] e^{-i\omega_k n}$$

such that the short-time discrete Fourier transform (STDFT) is expressed as:

$$STDFT\{s[n]\} = F(m, \omega) = \sum_{0}^{\infty} s[n] w[n - m] e^{-i\omega_k n}$$

The STDFT is by essence a collection of successive frequency spectra that are grouped into a single matrix which dimensions are determined by the length N of the sound and the time index m that corresponds to the size of the sliding window. The matrix can be expressed by referring to the Fourier coefficients a_{kj}, with K the number of frequencies ω and J the number of Fourier windows computed along the signal:

$$\begin{array}{c} \\ \omega_1 \\ \vdots \\ \omega_k \\ \vdots \\ \omega_K \end{array} \begin{array}{c} \begin{array}{ccccc} n_1 & \dots & n_j & \dots & n_J \end{array} \\ \begin{pmatrix} a_{11} & \dots & a_{1j} & \dots & a_{1J} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{k1} & \dots & a_{kj} & \dots & a_{kJ} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{K1} & \dots & a_{Kj} & \dots & a_{KJ} \end{pmatrix} \end{array}$$

Because of the mirror property of the Fourier transform, K is simply obtained by dividing the size of the window m by 2. J is the floor of the ratio of signal size N over window size m in number of samples. For instance, a STDFT computed with a window of 512 samples for a signal lasting 1.5 s sampled at $f_s = 44{,}100$ Hz returns a matrix containing $J = \lfloor N \div m \rfloor = \lfloor (44{,}100 \times 1.5) \div 512 \rfloor = 129$ columns and $K = m \div 2 = 512 \div 2 = 256$ lines.

11.1.2 The Uncertainty Principle

11.1.2.1 Time and Frequency Resolutions

The STDFT produces a two-dimensional object based on a time and frequency discretization through windowing. The returned object is hence made of atoms or cells. Each atom of the STDFT has a time-frequency localization represented as a "Heisenberg box" located in the time-frequency plane (Mallat 2009). The Heisenberg box is a rectangle with a time width (σ_t) and a frequency height (σ_f) where the signal is assumed to be locally stationary, that is, where amplitude and frequency modulations are considered to be null (Fig. 11.2). If the assumptions are not met, other solutions should be investigated as the wavelet transform (Staszewski and Robertson 2007). The size and the shape of the Heisenberg box are heavily constrained by a trade-off between the time resolution and the frequency

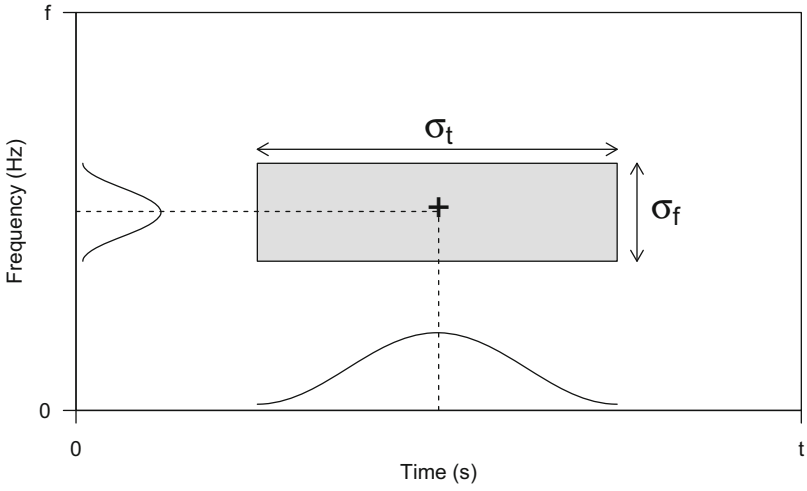


Fig. 11.2 Heisenberg box. The principle of the short-time discrete Fourier transform is based on a division of the time-frequency plane into an array of atoms. A unity atom is named a Heisenberg box represented as a quadrilateral with a width σ_t and a height σ_f. The window function applied on the frequency domain applies as well on the frequency domain. Inspired from Mallat (2009)

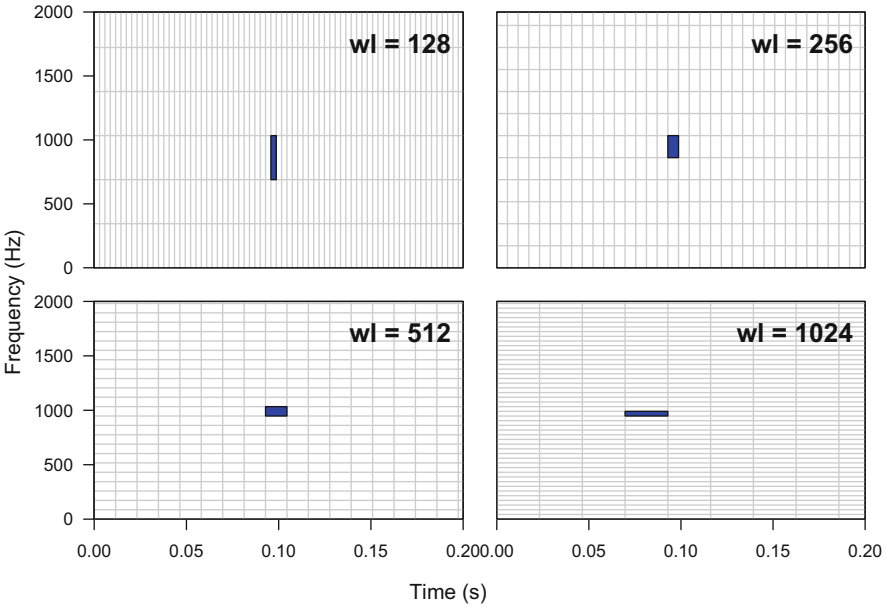


Fig. 11.3 Short-time discrete Fourier transform: atom shape. The figure shows the shape of the atoms (or Heisenberg boxes) for different window sizes. Four time width σ_t are considered: 128, 256, 512 and 1024 samples for a 0.2 s sound sampled at 44,100 Hz. A zoom is operated along the frequency y-axis from 0 to 2000 Hz. To facilitate the comparison, one central atom is highlighted in blue

resolution in relation with the Heisenberg uncertainty principle already evoked in Sect. 10.1.2.1.

Figure 11.3 illustrates the time-frequency discretization operated by the STDFT on a signal sampled at $f_s = 44{,}100$ with four different DFT windows. The choice of the window length (or duration) σ_t is crucial: a DFT window made of $\sigma_t = 128$ samples corresponding to a duration of 0.0029 s returns a STDFT with a good time resolution but a poor frequency resolution when a long DFT window made of $\sigma_t = 1024$ samples corresponding to a duration of 0.0232 s returns a STDFT with a poor time resolution but a good frequency resolution. In the specific case of the Fig. 11.3, the choice of an intermediate window length ($\sigma_t = 512$ samples $= 0.00116$ s) seems to provide a good compromise between time and frequency resolutions.

The time and frequency resolutions both depends on the sampling frequency f_s and are related following:

$$\Delta_t = \frac{1}{\Delta_f} = \frac{\sigma_t}{f_s}$$

11.1.2.2 Increasing the Time Resolution with Window Overlap

A first way to reduce the effects of the uncertainty principle is to try to increase the time resolution without reducing the frequency resolution. This is be achieved by applying an overlap over the successive windows (see Sect. 5.2.3.1). The time and frequency resolutions are now obtained with:

$$\Delta_t = \frac{1}{\Delta_f} = \frac{\sigma_t}{f_s} \times \frac{100 - \text{overlap}}{100}$$

where overlap is the window overlap expressed in percentage (%).

Figure 11.4 illustrates the Heisenberg box grid obtained for a fixed time window width ($\sigma_t = 512$ samples) but with different degrees of overlap. The box width σ_t remains the same, but the time resolution Δ_t increases from $\Delta_t = 512 \div 44{,}100 = 0.0116\,\text{s}$ for a null overlap to $\Delta_t = 512 \div (8 \times 44{,}100) = (512 \div 44{,}100) \times ((100 - 87.25) \div 100) = 0.00145\,\text{s}$ for a 87.5% overlap. The overlap solution is quite attractive; however, it increases the number of DFT to compute by a factor of $100 \div (100 - \text{overlap})$. For instance, setting an overlap of 87.5% induces the

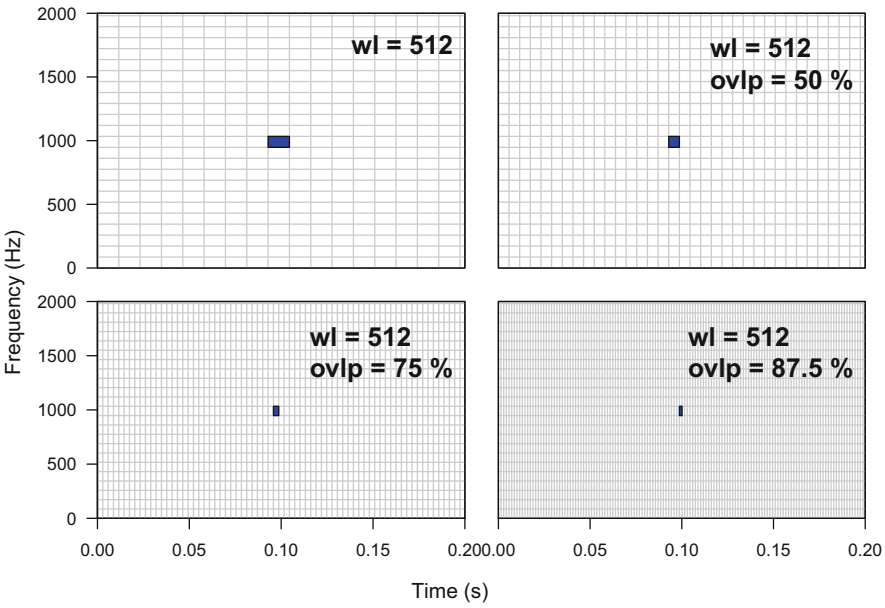


Fig. 11.4 Short-time Fourier discrete transform: atom shape with overlapping. The figure shows the shape of the atoms (or Heisenberg boxes) obtained with a window made of 512 samples. Four overlaps between successive windows are considered: 0%, 50%, 75%, and 87.5% for a 0.2 s sound sampled at 44,100 Hz. A zoom is operated along the frequency y-axis from 0 to 2000 Hz. To facilitate the comparison, one central atom is highlighted in blue

computation of eight times more DFTs:

```
ovlp <- c(25, 50, 75, 87.5)  # overlap in %
100/(100-ovlp)               # number of FTs scaling factor
[1] 1.333333 2.000000 4.000000 8.000000
```

11.1.2.3 Increasing the Frequency Resolution with Zero-Padding

A second trick to cope with the uncertainty principle is to increase the frequency resolution without losing time resolution. The idea is to artificially widen the Heisenberg box by adding P zeros after the box, a process named zero-padding, but keeping constant the time resolution of the original box, such that we have:

$$\Delta_f = \frac{f_s}{\sigma_t + P}$$

and yet:

$$\Delta_t = \frac{\sigma_t}{f_s}$$

Figure 11.5 illustrates the Heisenberg box grid obtained for a fixed time window width ($\sigma_t = 512$ samples) but with the different zero-paddings. The box width σ_t remains the same, but the frequency resolution Δ_f increases from $\Delta_f = 44{,}100 \div 512 = 86.13$ Hz for no zero-padding to $\Delta_f = 44{,}100 \div (512 + 128) = 68.90$ Hz for a zero-padding with $P = 128$.

11.2 Computation and Display of the Spectrogram

The spectrogram is the square of the STDFT, it is therefore made of a collection of power spectra densities (PSD). The equation of the STDFT is:

$$spectrogram\{s[n]\} = |F(m, \omega)|^2$$

The spectrogram (sometimes named sonagram) is a visualization tool which is commonly used, in particular in bioacoustics. The matrix of the Fourier coefficients is displayed as an image with time along the x-axis, frequency along the y- axis, and amplitude encoded as a gradient of gray or color levels. The spectrogram can also be projected as a waterfall display or a 3D object.

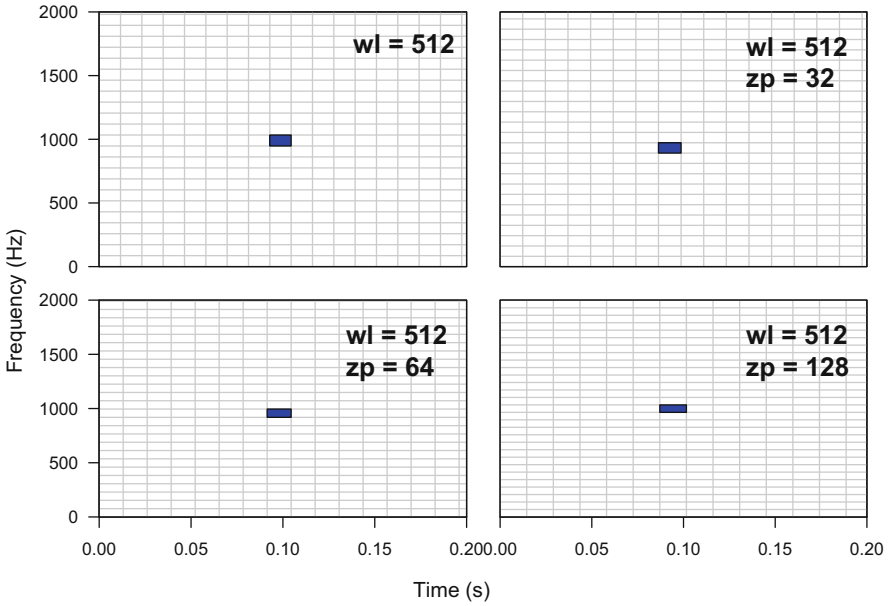


Fig. 11.5 Short-time Fourier transform: atom shape with zero-padding. The figure shows the shape of the atoms (or Heisenberg boxes) obtained with a window made of 512 samples. Four zero-padding settings are considered: 0, 32, 64, and 128 for a 0.2 s sound sampled at 44,100 Hz. A zoom is operated along the frequency y-axis from 0 to 2000 Hz. To facilitate the comparison, one central atom is highlighted in blue

The spectrogram can be obtained with functions of the packages `signal`, `tuneR`, `seewave`, `phonTools`, `monitoR`, and `warbleR`. The visual aspect of the spectrogram depends on mathematical settings that are common to all spectrographic functions and on graphical settings, such as the control of the colors, which are specific to each spectrographic function. A correspondence between the mathematical and graphical settings (arguments) of the different spectrographic functions is given in Table 11.2.

The mathematical settings of the spectrogram refer to the computation of the STDFT in link with the uncertainty principle detailed just above. We can list five main parameters that influence the spectrogram:

1. the sampling frequency, f_s, expressed in Hz,
2. the Fourier window function or taper, labeled with a name ("rectangle," "bartlett," "blackman," etc.),
3. the Fourier window length (or width or duration) σ_t, expressed either in number of samples or in s,
4. the amount of overlap (or hopp) between successive Fourier windows, expressed either in number of samples or percentage,
5. the zero-padding, expressed in number of zeros added.

Each parameter has an impact on the spectrogram. A high sampling frequency is the requirement for a good sound digitization but also implies to use a wide Fourier window to reach an acceptable frequency resolution. A STDFT processed on two sounds sampled, respectively, at 96,000 and 44,100 Hz with a similar Fourier window length $\sigma_t = 512$ has a frequency resolution $\Delta_f = 96{,}000 \div 512 = 187.5$ and $\Delta_f = 44{,}100 \div 512 = 86.13$ Hz. Downsampling the original sound may then be a solution to increase Δ_f.

The Fourier window function has important effects on the shape of the frequency spectra and so on the STDFT has illustrated in Fig. 9.13.

As we have seen in the previous section, the Fourier window length, overlap, and zero-padding can also change the time-frequency resolution of the spectrogram. The choice of these parameters is therefore not straightforward and should be carefully considered. Table 11.1 provides a few examples of time and frequency resolution obtained for different combinations of f_s, σ_t, and overlap. As an example, the spectrogram of a few seconds sound sampled at 44,100 Hz is usually well constructed with a Hanning (Hann) window of 512 samples with an overlap of 75%.

For the following sections, we will use a synthetic signal with amplitude and frequency modulations. The sound, which has a duration of 1 s with a sampling frequency of 44,100 Hz, was generated as described in Sect. 18.6.3.2 and saved in a file named `synth-face.wav`:

Table 11.1 Time and frequency resolution of the STFT

f_s	σ_t	$ovlp$	Δt	Δf
22,050	512	0	0.0232	43.07
22,050	512	50	0.0116	43.07
22,050	512	75	0.0058	43.07
22,050	512	88	0.0029	43.07
22,050	1024	0	0.0464	21.53
22,050	1024	50	0.0232	21.53
22,050	1024	75	0.0116	21.53
22,050	1024	88	0.0058	21.53
44,100	512	0	0.0116	86.13
44,100	512	50	0.0058	86.13
44,100	512	75	0.0029	86.13
44,100	512	88	0.0015	86.13
44,100	1024	0	0.0232	43.07
44,100	1024	50	0.0116	43.07
44,100	1024	75	0.0058	43.07
44,100	1024	88	0.0029	43.07

The table shows different time (Δt in s) and frequency (Δf in Hz) resolutions in respect to sampling frequency (f_s in Hz), Fourier window length (wl in number of samples), and Fourier window overlap ($ovlp$ in %)

Table 11.2 Correspondence between the main arguments of spectrographic functions found in several packages

Package	signal	tuneR	tuneR	seewave	phonTools	monitoR	warbleR	soundgen
Function	specgram(), plot()	periodogram()	powspec()	spectro()	spectrogram()	viewSpec()	lspec()	spectrogram()
Input	x	object	x	wave	sound	clip	X	x
Sampling frequency	Fs	–	sr	f	fs	samp.rate	–	samplingRate
Window function	window	–	–	wn	window	wn	–	wn
Window size	n	width	wintime	wl	windowlength	wl	wl	windowLength
Overlap	overlap	overlap	steptime	ovlp	timestop	ovlp	ovlp	overlap
Zero-padding	–	–	–	zp	padding	zp	–	zp
Color	col	–	–	palette	colors	spec.col	pal	colorTheme
Time limits	xlim	from, to, units	–	tlim	–	start.time, units	–	–
Frequency limits	ylim	–	–	flim	maxfreq	frq.lim	flim	–
Amplitude limits	–	–	–	collevels	nlevels	collevels	collevels	–

```
face <- readWave("sample/synth-face.wav")
face

Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    32
```

11.3 Function of the Package signal

The package signal, which we have not met yet, includes a function named specgram() to compute and plot the spectrogram of a numeric vector x. The function, the size, and the overlap of the Fourier window are set with the arguments window, n, and overlap, respectively. The plot is based on a call of the high-level plot function image(). To obtain a spectrogram of face, we first need to call the library signal:

```
library(signal)
```

We then apply specgram() on the left channel of face which is a Wave object. We also specify the sampling frequency, the size of the window, and the overlap between successive windows in number of samples. The overlap is here set to 75% of the window size, that is, in number of samples $\texttt{overlap} = 512 \times 0.75 = 384$:

```
f <- face@samp.rate
spg <- specgram(face@left, Fs=f,
         n=512, overlap=512*0.75)
```

The object returned by specgram() is an object of class specgram which consists of a list containing three items: $S a (256, 341) matrix with the FFT results as complex numbers, $f the frequencies, and $t the time:

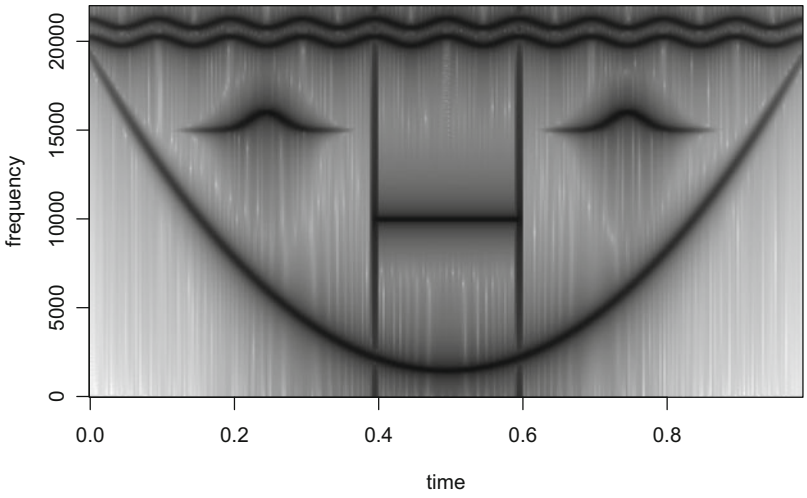


Fig. 11.6 Spectrogram with `specgram()` of `signal`. The spectrogram is computed and displayed with the function `specgram()` of the package `signal`. Fourier window size = 512 samples, overlap = 75% = 383 samples, Hanning window

```
str(spg)
List of 3
 $ S: cplx [1:256, 1:341] 120.13+0i -67.21-2.4i -9.66+1.3i ...
 $ f: num [1:256] 0 86.1 172.3 258.4 344.5 ...
 $ t: num [1:341] 2.27e-05 2.93e-03 5.83e-03 8.73e-03 1.16e-02 ...
 - attr(*, "class")= chr "specgram"
```

The matrix, which is the spectrogram, can be plotted directly with `specgram()` or by calling the generic function `plot()`. The matrix of the STDFT results stored in `$S` are converted into dB. The main arguments of `plot()`, such as `col`, `xlim`, or `ylim`, can be used to tune the graphical output (Fig. 11.6):

```
plot(spg, col=gray((512:0)/512))
```

11.4 Functions of the Package `tuneR`

The function `periodogram()` of `tuneR`, which was already mentioned when treating the frequency spectrum (see Sect. 10.1.1), can be used to compute the spectrogram. The function can calculate a series of power spectrum densities using

the arguments width and overlap which correspond to the size and the overlap of the Fourier window. The spectrogram of face is obtained with:

```
p <- periodogram(face, width=512, overlap=512*0.75)
```

The function does not display anything but returns a Wspec object. The object is quite large, and printing it in the console could take a lot of space. We can first get the name of the different slots with:

```
slotNames(p)
 [1] "freq"      "spec"      "kernel"    "df"
 [5] "taper"     "width"     "overlap"   "normalize"
 [9] "starts"    "stereo"    "samp.rate" "variance"
[13] "energy"
```

There are slots informing about the properties of the spectrogram (freq, spec, kernel, df, variance), the settings used to compute it (taper, width, overlap, normalize), and the sound used as an input (starts, stereo, samp.rate). The most important slot is the slot spec which contains the successive power spectrum densities (PSDs) organized in a list. Here p@spec contains 342 PSDs:

```
length(p@spec)
[1] 342
```

The plot function associated to periodogram() plots one of the frequency spectra of the spectrogram. For instance, the following code plots the first PSD only:

```
plot(p)
```

and the next code plots the 10th PSD:

```
plot(p, which=10)
```

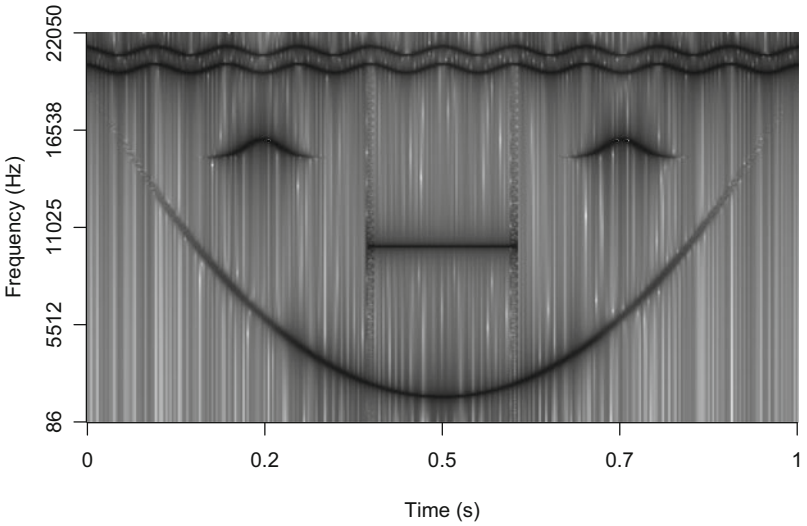


Fig. 11.7 Spectrogram with `periodogram()` of `tuneR`. The spectrogram is computed with the function `periodogram()` of the package `tuneR` and displayed with the function `image()`. Fourier window size = 512 samples, overlap = 75%, split cosine bell window

To get a display of the spectrogram, we need to extract the slot `spec`, to convert it into a matrix, to scale the data in [0, 1], to convert the data in dB, and to use the function 2D-plot function `image()` to plot the transpose of the matrix. Labeling the axes is a bit tricky, in particular for the time axis (Fig. 11.7):

```
f <- face@samp.rate
p.data <- matrix(unlist(p@spec), nc=length(p@spec)) # data
p.data <- p.data/max(p.data)                        # scaling
p.data <- 10*log10(p.data)                          # dB scale
image(t(p.data), col=gray((512:0)/512),             # image plot
      xlab = "Time (s)", ylab = "Frequency (Hz)",   # axes labels
      axes=FALSE)                                   # no units
# manual construction of the axes
frequency <- round(p@freq[seq(1, length(p@freq), length=5)])
time <- round(p@starts[seq(1, length(p@starts), length=5)]/f, 1)
axis(side=1, at=seq(0, 1,length=5), labels=time)
axis(side=2, at=seq(0, 1,length=5), labels=frequency)
```

`tuneR` has another function, `powspec()` based on `specgram()` of `signal`, that can compute the spectrogram as well. The function can take any numeric vector as input, and the Fourier window parameters, the length and the overlap, are encoded in the arguments `wintime` and `steptime` expressed in s such that a window of

512 samples should be set to $512 \div f_s$ and a steptime for a 75% overlapping window to $0.25 \times 512 \div f_s$:

```
f <- face@samp.rate
pspectrum <- powspec(face@left, sr=f,
                     wintime=512/f, steptime=0.25*512/f)
```

The data are returned in a numeric (256, 341) matrix containing linear values:

```
str(pspectrum)
 num [1:256, 1:341] 5.06e+10 5.05e+10 5.05e+10 5.05e+10 5.05e+10 ...
```

The value of `powspec()` is therefore on a linear scale without information on neither the frequency nor the time scale. A dB display can be manually obtained with the following code (Fig. 11.8):

```
pspectrum <- pspectrum/max(pspectrum)                # scaling
pspectrum <- 10*log10(pspectrum)                     # dB scale
image(t(pspectrum), col=gray((512:0)/512),           # plot
```

(continued)

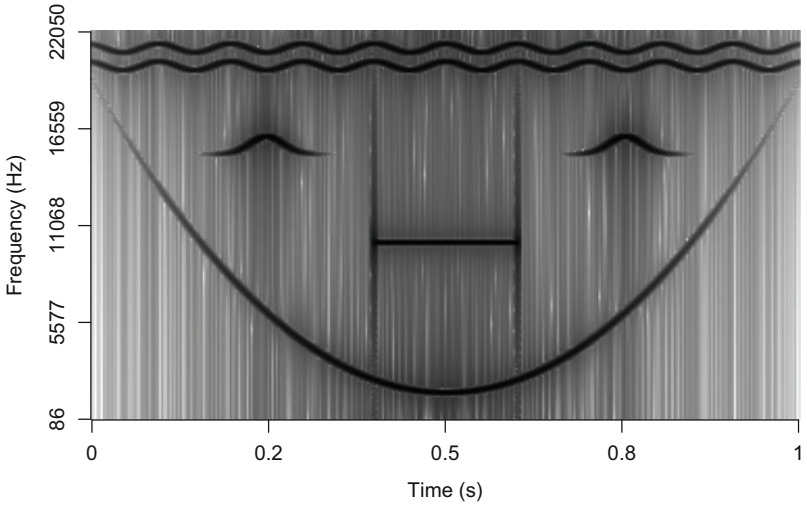


Fig. 11.8 Spectrogram with `powspec()` of `tuneR`. The spectrogram is computed with the function `powspec()` of the package `tuneR` and displayed with the function `image()`. Fourier window size = 512 samples, overlap = 75%, Hamming window

```
      xlab = "Time (s)", ylab = "Frequency (Hz)",  # axes labels
      axes=FALSE)                                  # no units
# manual display of the axes
time <- round(seq(0, duration(face), length=5), 1)
frequency <- round(seq(f/512, f/2, length=5))
axis(side=1, at=seq(0, 1,length=5), labels=time)
axis(side=2, at=seq(0, 1,length=5), labels=frequency)
```

11.5 Function of the Package `phonTools`

`phonTools` has its own spectrographic function as well, named `spectrogram()`. The function is tuned to speech with time expressed in ms, frequency in Hz, a default value of the upper frequency limit set to 5000 Hz, and a preemphasis filter (see Sect. 14.1). We can adapt it to show the face sound as we do with other spectrographic functions. The window length is set in ms with the argument `windowlength` and the overlap is set in ms as well with `timestep` (Fig. 11.9). We specifically refer to the package `phonTools` with the syntax `::` following:

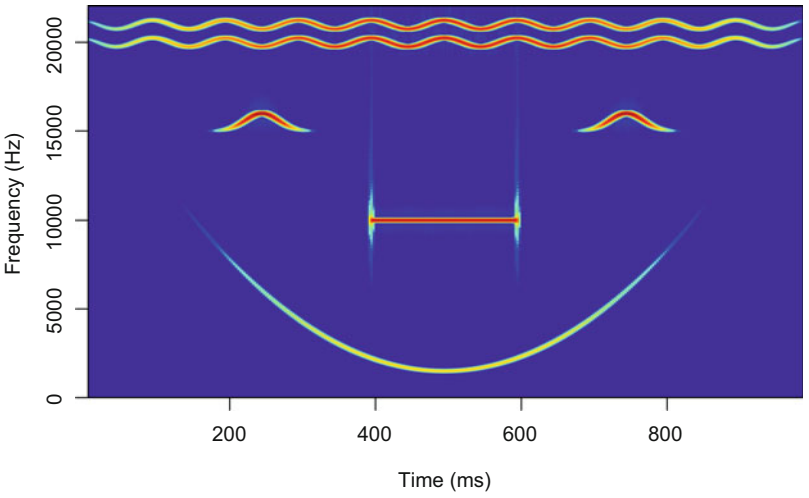


Fig. 11.9 Spectrogram with `spectrogram()` of `phonTools`. Fourier window size = 512 samples, overlap = 75%, Hamming window

```
f <- face@samp.rate
phonTools::spectrogram(face@left, fs=f,
            windowlength=1000*512/f,
            timestep=0.25*(512/f)*1000,
            maxfreq=f/2,
            window="hamming")
```

11.6 Function of the Package soundgen

The package soundgen includes as well function to plot a spectogram. This function, named spectrogram() as the one of phonTools, mainly derives from the function spectro() of seewave (see Sect. 11.7), with a few different graphical parameters as the control of contrast and brightness. Here is a basic use of this function; we refer to the package soundgen with ::, and we coerce the sound data into a numeric object with as.numeric(). The window length is given in ms and the overlap in % (Fig. 11.10):

```
f <- face@samp.rate
soundgen::spectrogram(as.numeric(face@left), samplingRate=f,
                      windowLength=1000*512/f,
                      overlap=75, wn="hamming")
```

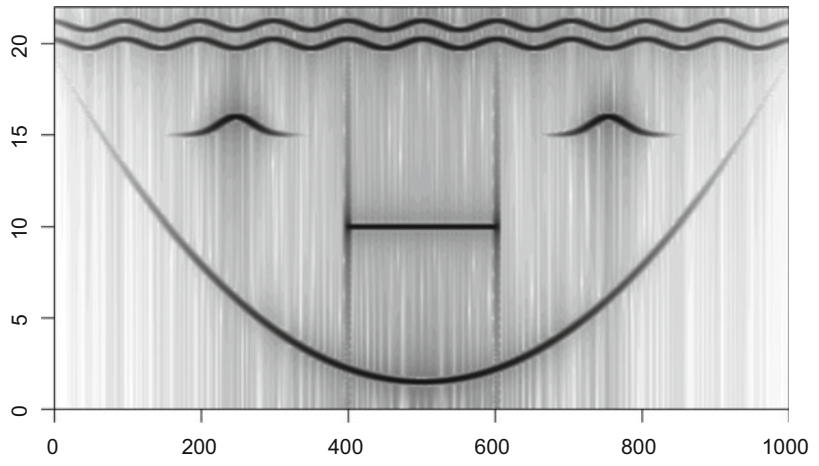


Fig. 11.10 Spectrogram with spectrogram() of soundgen. Fourier window size = 512 samples, overlap = 75%, Hamming window

11.7 Functions of the Package `seewave`

The package `seewave` includes three main functions related to the spectrogram: (1) the function `spectro()` that computes and display the spectrogram, (2) the function `stft.ext()` that computes the short-time discrete Fourier transform out of `R` by calling `.C` programs, and (3) the function `istft()` that computes the inverse short-time discrete Fourier transform.

11.7.1 2D Spectrogram

11.7.1.1 Setting the Scene

The main graphical output of `spectro()` is based on the base function `filled.contour()` and operates a $[-30, 0]$ selection along the dB scale enhancing the visual quality as testified by the Fig. 11.11.

```
spectro(face, ovlp=75)
```

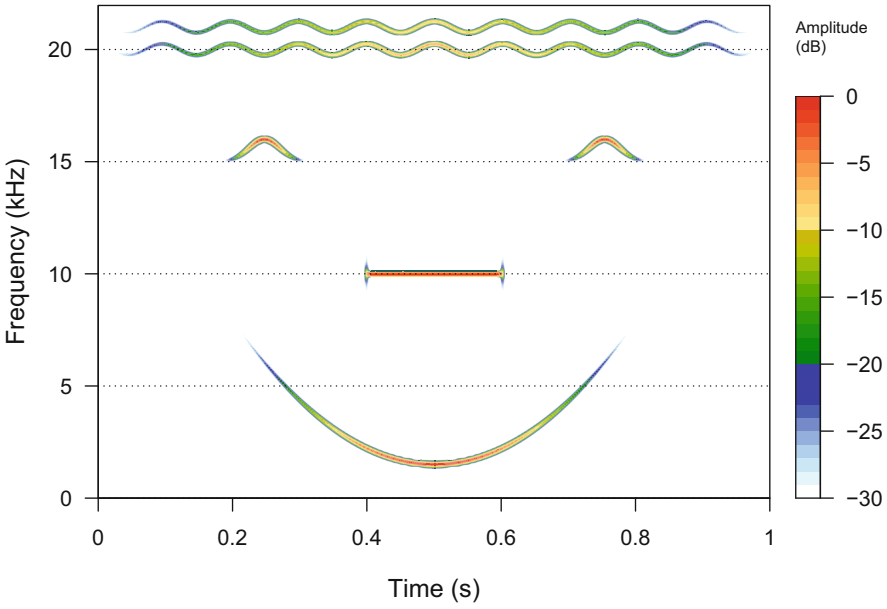


Fig. 11.11 Spectrogram with `spectro() of seewave`. Fourier window size = 512 samples, overlap = 75%, Hanning window

The function spectro() works with an long list of arguments which default values are summarized in Table 11.3 and that can be grouped in seven categories:

1. Input arguments
2. Fourier related arguments
3. Output arguments
4. High-level plot arguments
5. Color arguments
6. Axes arguments
7. Layout arguments

To explore this long list of arguments, we will refer to the strange sound produced by a male hissing cockroach from Madagascar, *Elliptorhina chopardi* (Fig. 11.12). The sound is brief and complex: it is made of two independent voices modulating in frequency. The two voices are due air expelled through a pair of modified abdominal spiracle (Sueur and Aubin 2006). The sound, recorded during a courtship sequence, is stored in the file Elliptorhina_chopardi.wav:

```
cockroach <- readWave("sample/Elliptorhina_chopardi.wav")
cockroach

Wave Object
Number of Samples:      19137
Duration (seconds):     0.43
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

11.7.1.2 Input Arguments

The arguments to control the sound input are:

wave R object, typically a Wave object as tico but could also be a vector, one-column matrix, ts, mts, Sample, audioSample, sound, or WaveMC object.

f sampling frequency of the object wave expressed in Hz. This argument is optional if the sampling frequency is embedded in wave, for instance, if wave is a Wave object. If, for any reason, the sampling frequency should be changed, then the raw data of the sound should be used. For instance, the spectrogram of face with a modified sampling frequency should call face@left as an input.

Table 11.3 Default values of the arguments of the seewave function spectro()

INPUT		
wave		
f		
FOURIER		
wl	=	512
wn	=	"hanning"
ovlp	=	0
zp	=	0
fftw	=	FALSE
OUTPUT		
complex	=	FALSE
norm	=	TRUE
dB	=	"max0"
dBref	=	NULL
listen	=	FALSE
PLOT		
plot	=	TRUE
osc	=	FALSE
scale	=	TRUE
cont	=	FALSE
contlevels	=	NULL
grid	=	TRUE
COLOURS		
collevels	=	NULL
palette	=	spectro.colors
colcont	=	"black"
colbg	=	"white"
colgrid	=	"black"
colaxis	=	"black"
collab	=	"black"
AXES		
cexlab	=	1
cexaxis	=	1
tlab	=	"Time (s)"
flab	=	"Frequency (kHz)"
alab	=	"Amplitude"

(continued)

Table 11.3 (continued)

scalelab	=	"Amplitude\n(dB)"
main	=	NULL
scalefontlab	=	1
scalecexlab	=	0.75
axisX	=	TRUE
axisY	=	TRUE
tlim	=	NULL
trel	=	TRUE
flim	=	NULL
flimd	=	NULL
LAYOUT		
widths	=	c(6, 1)
heights	=	c(3,1)
oma	=	rep(0,4)

Fig. 11.12 Pictures of soniferous animals: the hissing cockroach of Madagascar *Elliptorhina chopardi* (reproduced with the kind permission of Emmanuel Delfosse) and the Kuhl's pipistrelle *Pipistrellus kuhlii*, a bat commonly found in Europe (reproduced with the kind permission of Laurent Arthur)

11.7.1.3 Fourier-Related Arguments

The arguments used to control and compute the short-time Fourier transform are:

`wl` Fourier **w**indow **l**ength corresponding to σ_t of the Heisenberg box width, expressed in number of samples, preferably as a power of 2. The default value is set to $2^9 = 512$ (Fig. 11.13).

`wn` name of the Fourier window or taper specified as a character string. The available functions are `"rectangle"`, `"bartlett"`, `"blackman"`, `"flatto"`, `"hamming"`, and `"hanning"` (default).

`ovlp` percentage of overlap between two successive windows (in %). This value is by default set to 0 and should not be set to 100 (stationary window) (Fig. 11.14).

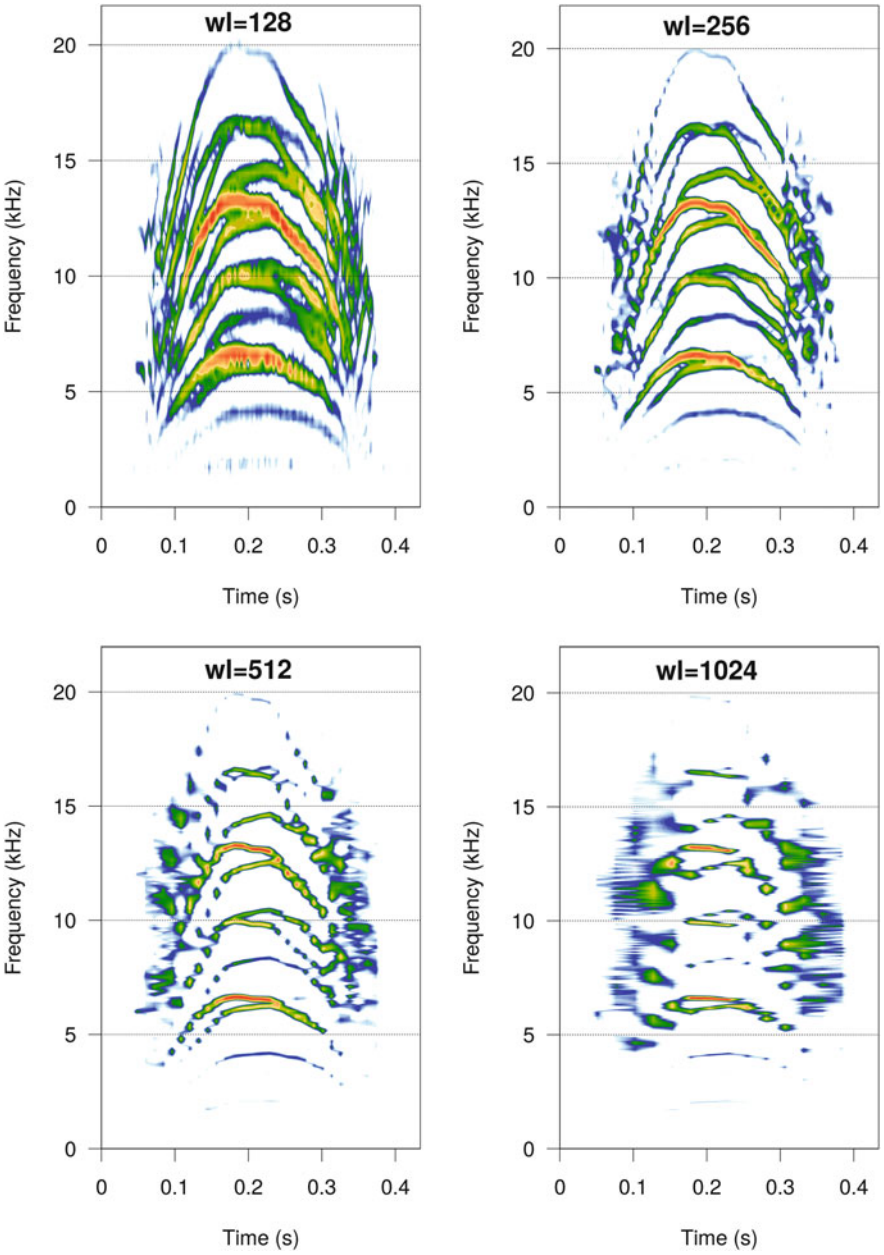


Fig. 11.13 Different Fourier window length with `spectro()`. The spectrogram of `cockroach` was obtained with `wl = {128, 256, 512, 1024}` samples. Other STDFT parameters: Hanning window, 0% of overlap, no zero-padding

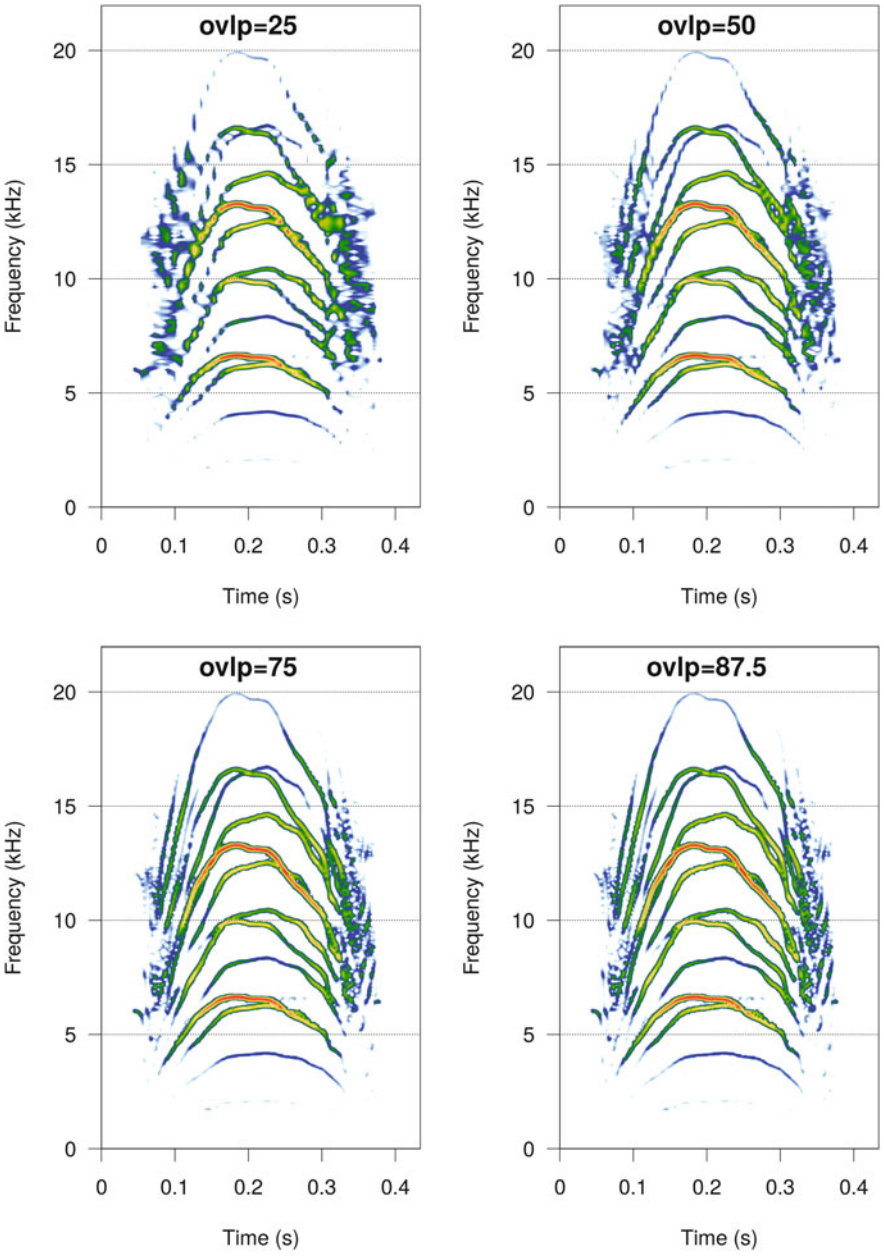


Fig. 11.14 Different Fourier window overlaps with `spectro()`. The spectrogram of `cockroach` was obtained with `ovlp = {25, 50, 75, 87.5}` samples. Other STDFT parameters: Hanning window, 512 samples, no zero-padding

`zp` zero-padding expressed as a numeric vector of length 1. The default value is 0, that is, no zero-padding is applied.

`fftw` this logical argument can be turned to `TRUE` to call the function `FFT` of the library `fftw` instead of the function `fft` of the base to compute the DFTs. This can speed up the time of process. The time saved increases with the duration of the sound to be analyzed and/or with the numbers of sound successively analyzed in a loop.

11.7.1.4 Output Arguments

The arguments that control the type of data returned are:

`complex` a logical which returns the raw data of the STDFT as complex numbers, that is, with real and imaginary parts. In that case, no plot is returned. This argument should be used when applying time-frequency modifications with the help of the inverse short-time Fourier transform as explained in Sect. 15.4.

`norm` if `TRUE` the matrix of the STDFT is normalized by its maximum. If a linear scale is used, then all values of the STDFT matrix ranges in [0, 1],

`correction` this argument can be used to apply an amplitude correction to take into account the amplitude changes induced by the Fourier window function (e.g., Hanning window). This argument is by default set to `"none"`, that is, no correction is applied, but can be changed to `"amplitude"` or `"energy"` to apply an amplitude or energy correction as detailed in Sect. 10.1.2.1. This correction has meaning only when `complex=FALSE`, `norm=FALSE`, and `dB=NULL`,

`dB` this argument is similar to the eponymous argument of the function `spec()` (see Sect. 10.1.2.1). The argument waits therefore a character string, either `"max0"` (default), `"A"`, `"B"`, `"C"`, or `"D"`. If set to `NULL`, then a linear scale is used,

`dBref` a single dB reference value. The default setting is `NULL` so that dB values vary along a relative scale. However, this can be changed to any value.

`listen` a logical. If `TRUE`, then the input sound `wave` is played back using the default player that should be previously appropriately chosen (see Sect. 4.3).

The function `spectro()` returns a list of three items if `plot=FALSE`. The numeric results can also be invisibly returned and saved in an object when `plot=TRUE`. For instance, the following line plots the spectrogram and saves the value in the object `res`:

```
res <- spectro(face)
```

The three items of the list are:

`$time` : a numeric vector corresponding to the time x-axis expressed in s. The length of this vector corresponds to the number of DFTs computed. This length therefore depends on the sampling frequency f_s, the Heisenberg box width σ_t, the overlap, and the zero padding,

`$freq` : a numeric vector corresponding to the frequency y-axis expressed in kHz. The length of this vector equals to $\sigma_t \div 2$, that is, $wl \div 2$,

`$amp` : a complex or a numeric matrix containing the Fourier coefficients of the successive DFTs corresponding to the amplitude z-axis. The number of columns of the matrix is the length of `$time` and the number of lines is the length of `$freq`.

The results can be obtained in different formats combining the arguments detailed just above:

- `spectro(..., norm=FALSE, dB=NULL, complex=TRUE, ...)`: complex values,
- `spectro(..., norm=FALSE, dB=NULL, ...)`: linear real values not normalized, not squared, not corrected,
- `spectro(..., norm=FALSE, dB=NULL, correction="amplitude", ...)`: linear real values not normalized, not squared, with an amplitude correction applied to the Fourier window function (e.g. Hanning window),
- `spectro(..., norm=FALSE, dB=NULL, correction="energy", ...)`: linear real values not normalized, not squared, with an energy correction applied to the Fourier window function (e.g. Hanning window),
- `spectro(..., dB=NULL, ...)`: linear real values normalized to 1, not squared,
- `spectro(...)`: dB real values normalized to 0,
- `spectro(..., norm=FALSE, ...)`: dB real values not normalized,
- `spectro(..., dB="A")`: dB A weighted real values with a prior normalization to 0,
- `spectro(..., norm=FALSE, dB="A")`: dB A weighted real values without a prior normalization to 0,
- `spectro(..., dBref=2*10e-5)`: dB real values according to a reference value.

The application of these argument combinations is illustrated with the following code:

```
# complex values
res <- spectro(cockroach, norm=FALSE, dB=NULL,
               complex=TRUE, plot=FALSE)
res$amp[1:10,1]
 [1]  0.06228362+0.00000000i -0.07776031+0.13572916i
```

(continued)

```
  [3]  0.03860612-0.05658345i  0.00618337-0.01581956i
  [5]  0.00713985+0.01096740i -0.01882588-0.01681960i
  [7]  0.04505415+0.00916466i -0.05292096+0.00477137i
  [9]  0.02918675-0.00081825i -0.01117293-0.00877874i
# real raw values
res <- spectro(cockroach, norm=FALSE, dB=NULL, plot=FALSE)
range(res$amp)
[1] 2.258614e-03 3.299268e+02
# real raw values with amplitude correction
res <- spectro(cockroach, norm=FALSE, dB=NULL,
               correction="amplitude", plot=FALSE)
range(res$amp)
[1] 4.526068e-03 6.611449e+02
# real raw values with energy correction
res <- spectro(cockroach, norm=FALSE, dB=NULL,
               correction="energy", plot=FALSE)
range(res$amp)
[1] 2.887663e-03 4.218151e+02
# real [0,1] values
res <- spectro(cockroach, dB=NULL, plot=FALSE)
range(res$amp)
[1] 6.845803e-06 1.000000e+00
# dB values maximized to 0
res <- spectro(cockroach, plot=FALSE)
range(res$amp)
[1] -103.2915    0.0000
# dB values not maximized
res <- spectro(cockroach, norm=FALSE, plot=FALSE)
range(res$amp)
[1] -52.92316  50.36835
# dB A after maximization to 0
res <- spectro(cockroach, norm=TRUE, dB="A", plot=FALSE)
range(na.omit(res$amp))  # includes NA
[1] -105.729333   -0.303597
# dB A without prior maximization to 0
res <- spectro(cockroach, norm=FALSE, dB="A", plot=FALSE)
range(na.omit(res$amp))  # includes NA
[1] -55.36098  50.06476
# dB ref
res <- spectro(cockroach, dBref=2*10e-5, plot=FALSE)
range(res)
[1] -29.31211  73.97940
```

The value of `spectro()` can be used to plot manually the spectrogram using the function `image()` as previously done with the function `periodogramn()` and `powerspec()` (Fig. 11.15):

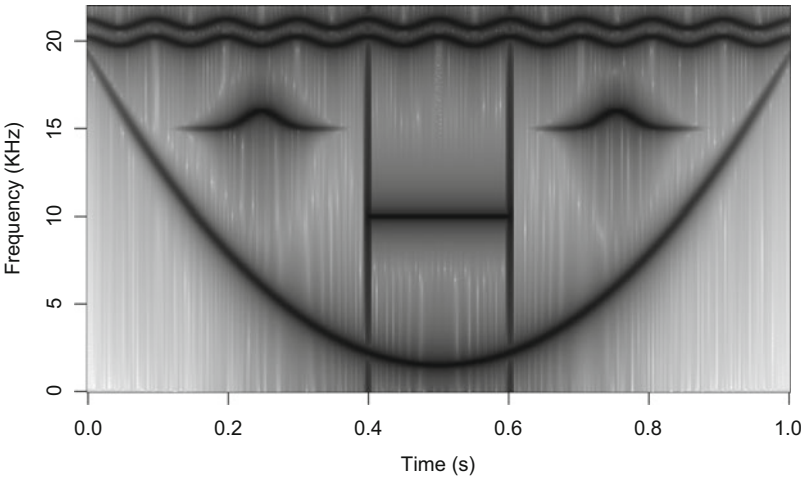


Fig. 11.15 The spectrogram is computed with the function `spectro()` of the package `seewave` and displayed with the function `image()`. Fourier window size = 512 samples, overlap = 75%, Hanning window

```
res <- spectro(face, ovlp=75, plot=FALSE)
str(res)
List of 3
 $ time: num [1:341] 0 0.00294 0.00588 0.00882 0.01176 ...
 $ freq: num [1:256] 0 0.0861 0.1723 0.2584 0.3445 ...
 $ amp : num [1:256, 1:341] -178 -183 -200 -205 -204 ...
image(x=res$time, y=res$freq, z=t(res$amp),
      xlab="Time (s)", ylab="Frequency (KHz)",
      col=gray((512:0)/512))
```

11.7.1.5 High-Level Plot Arguments

The arguments to control the overall organization of the spectrographic display are:

`plot` a logical, to plot the spectrogram.
`osc` a logical, to plot an oscillogram beneath the spectrogram (Fig. 11.16).
`scale` a logical, to plot an amplitude scale on the right side of the spectrogram (Fig. 11.16 and DIY box 11.1).

```
spectro(cockroach, ovlp=87.5, osc=TRUE, scale=TRUE)
```

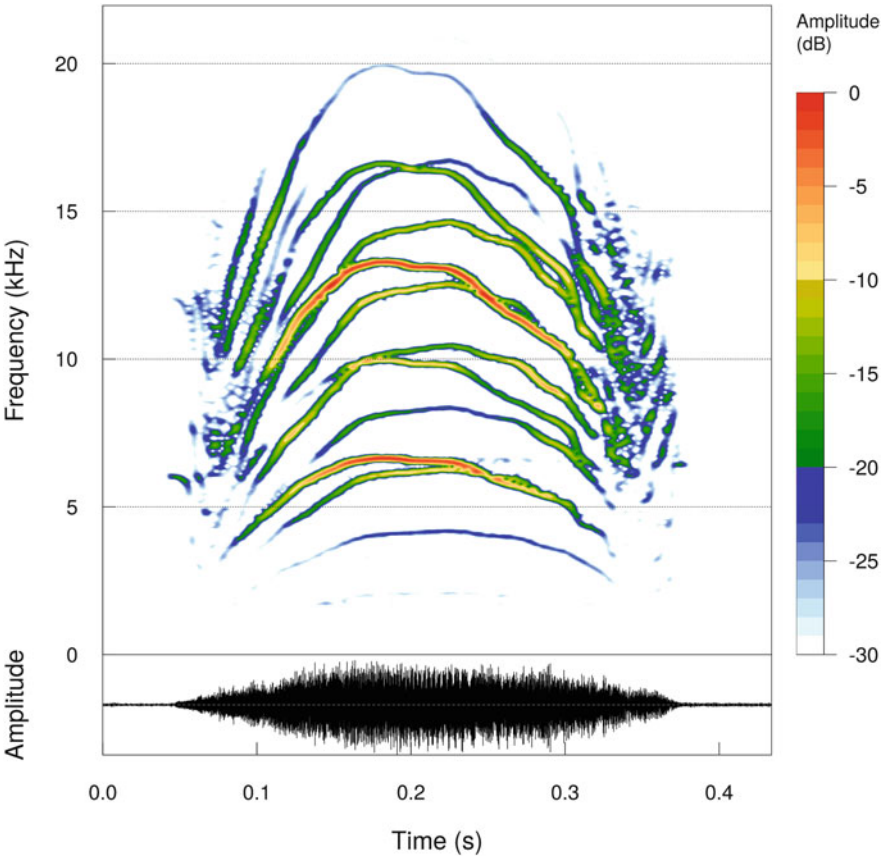


Fig. 11.16 Spectrogram, oscillogram and amplitude scale display with `spectro()`. STDFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

DIY 11.1 — How to change the position of the amplitude scale and plot a spectrum on the side of the spectrogram

The idea is to plot a spectrogram and a frequency spectrum aside. The default display of the spectrogram places the dB amplitude scale on the right. This scale needs them to be moved to another part of the graphic. Here the choice is to flip horizontally and to position the scale above the spectrogram. To do so, we need to manually set a proper layout (see Sect. 3.3.9.4) with one cell empty on the top-right section of the plate:

```
m <- matrix(c(1,0,2,3), nc=2, byrow=TRUE)
layout(m, widths=c(2,1), heights=(c(1,5.5)))
```

(continued)

DIY 11.1 (continued)

Then, we need to set the inner margins and to plot the dB amplitude scale with the help of the seewave function dBscale. This function plots a dB scale to be placed either below (side=1), on the left (side=2), on the top (side=3), or on the right (side=4) of the spectrogram:

```
par(mar=c(0.5,5,4,0))
dBscale(collevels=seq(-30,0,1), side=3,
        textlab="Amplitude (dB)\n")
```

We then fill in the second cell of the layout with new inner margins, and we add a vertical dashed line with abline() to highlight the position of the frequency spectrum that we will plot on the right side:

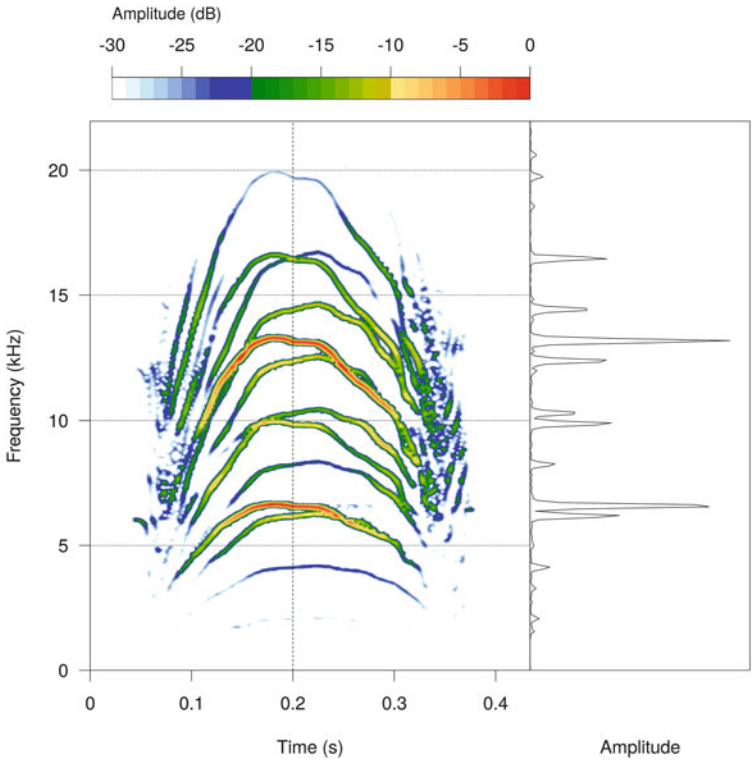
```
par(mar=c(5,4,0.5,0))
spectro(cockroach, ovlp=87.5, scale=FALSE)
abline(v=0.2, lty=2)
```

The last step consists in plotting the frequency spectrum obtained with spec(). We carefully specify that the graphic should be oriented horizontally with plot=2 and that the frequency *x*-axis label and the axes values should not be displayed. The inner margins are chosen to join the frequency spectrum to the right side of the spectrogram:

```
par(mar=c(5,0,0.5,2))
spec(cockroach, at=0.2, plot=2,
     flab="", xaxt="n", yaxt="n")
```

(continued)

DIY 11.1 (continued)



`cont` a logical, to overplot contour lines (i.e., iso-dB lines) on the spectrogram.

`contlevels` a numeric series specifying the set of levels which are used to partition the dB amplitude range for contour overplot. This argument works only if `cont=TRUE`. Usually the series is defined using the function `seq()` following the form `seq(from, to, by)`, with `from` the lower amplitude limit, `to` the upper amplitude unit and `by` the distance between two successive iso-dB lines. For instance, setting `contlevels = seq(-30, 0, 5)` draws iso-dB lines from -30 to 0 by step of 5, that is, iso-dB lines in $\{-30, -25, -20, -15, -10, -5, 0\}$. The following code shows how to plot exclusively the contours, the arguments `palette` and `colcont` being detailed below (Fig. 11.17):

```
blanc <- colorRampPalette("white")
spectro(cockroach, contlevels=seq(-30, 0, 4),
        cont=TRUE, colcont=temp.colors(8),
        palette=blanc, scale=FALSE)
```

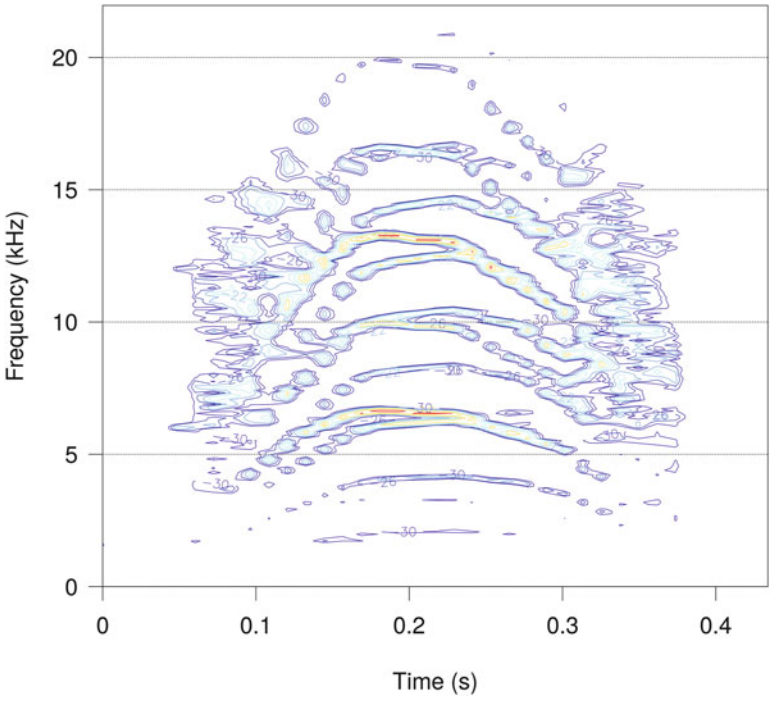


Fig. 11.17 Contour plot with `spectro()`. The contours shows iso-dB lines from −30 to 0 dB regularly spaced by 4 dB. STDFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

`flog` a logical, to plot the frequency *y*-axis on a logarithmic scale (Fig. 11.18):

```
spectro(cockroach, ovlp=87.5, flog=TRUE)
```

`grid` a logical, to plot a frequency *y*-axis grid made with dashed black lines.

`...` these three dots indicate that the arguments of the functions `contour()` and `oscillo()` can be parsed from `spectro()`. The most interesting arguments of `contour()` are probably `drawlabels` a logical to control whether the labels of the contour lines should be drawn or not, `method` to choose where the labels should be positioned (default `"flattest"`), and `labcex` to control their size. The numerous arguments of `oscillo()` are detailed in Sect. 5.1.

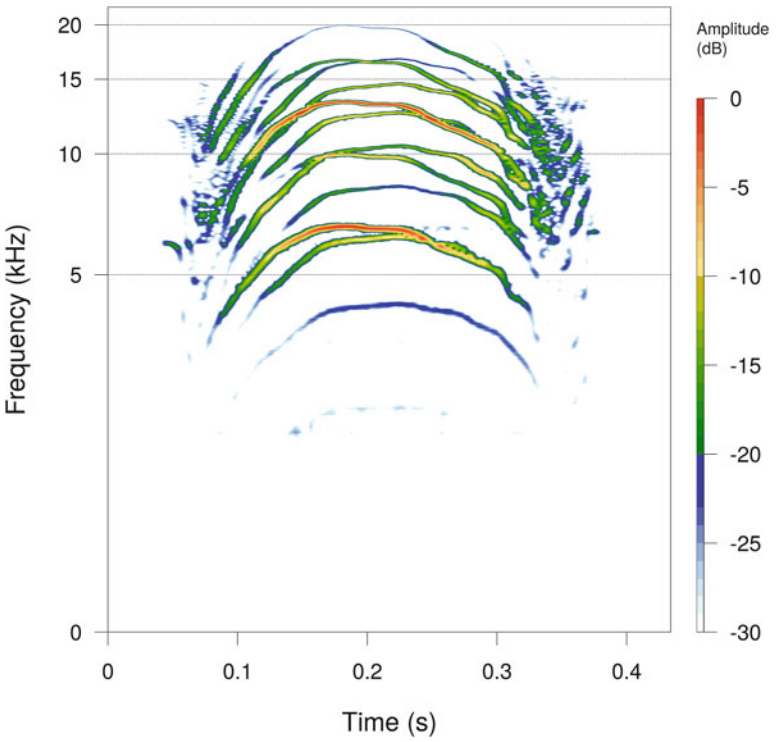


Fig. 11.18 Spectrogram with a logarithmic frequency scale. The logarithmic scale obtained the argument `flog=TRUE`

11.7.1.6 Colour Arguments

The arguments to control the color of the lines, surfaces, and annotations of `spectro()` are:

`collevels` a numeric series specifying the set of levels to partition the dB amplitude range for the spectrogram. Usually the series is defined using the function `seq()` following the form `seq(from, to, by)`, `from` the lower amplitude limit `to` the upper amplitude unit and `by` the number of dB per color. The default value `contlevels=seq(-30, 0, 1)` builds a spectrogram with 36 colors along a dynamic ranging from −30 to 0 dB. Setting a value of `contlevels=seq(-60, 0, 2)` produces a spectrogram with $60 \div 2 = 30$ colors over a range of 60 dB (Fig. 11.19).

`palette` a color palette function to be used to assign colors to the spectrogram (Figs. 11.20, 11.21, 11.22, and 11.23). The number of colors is set by `collevels`. The default palette is the `seewave` palette named `spectro.colors`. Any other palette can be used including `reverse.heat.colors`,

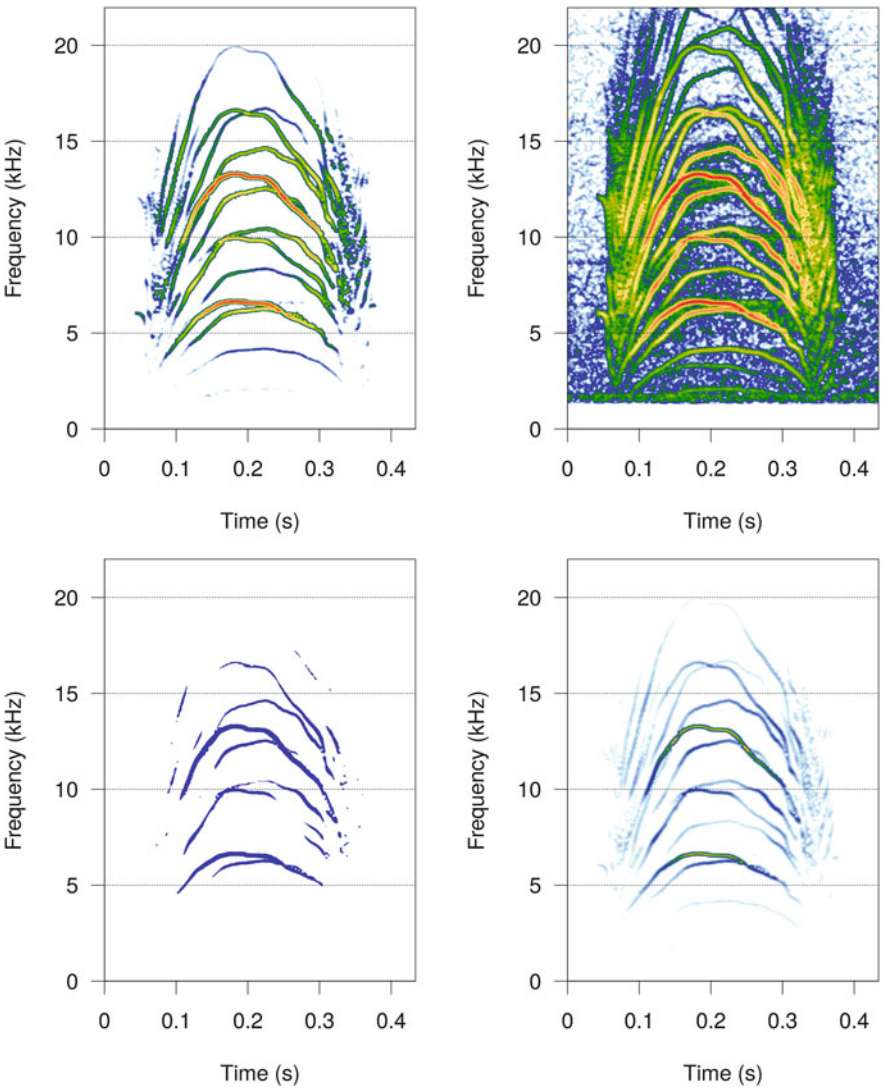


Fig. 11.19 Different color levels with `spectro()`. The spectrogram of `cockroach` was obtained with four different series of color levels: a linear series going from −30 to 0 by step of 1 (`collevels=seq(-30,0,1)`), a linear series going from −60 to 0 by step of 4 `collevels=seq(-60,0,4)`, a linear series going from −30 to 0 by step of 15 creating a two-color scale (`collevels=seq(-30,0,15)`), and a logarithmic series from −30 to 0 (`collevels=c(-exp(seq(log(30), 0, length=30)))`). Other STDFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

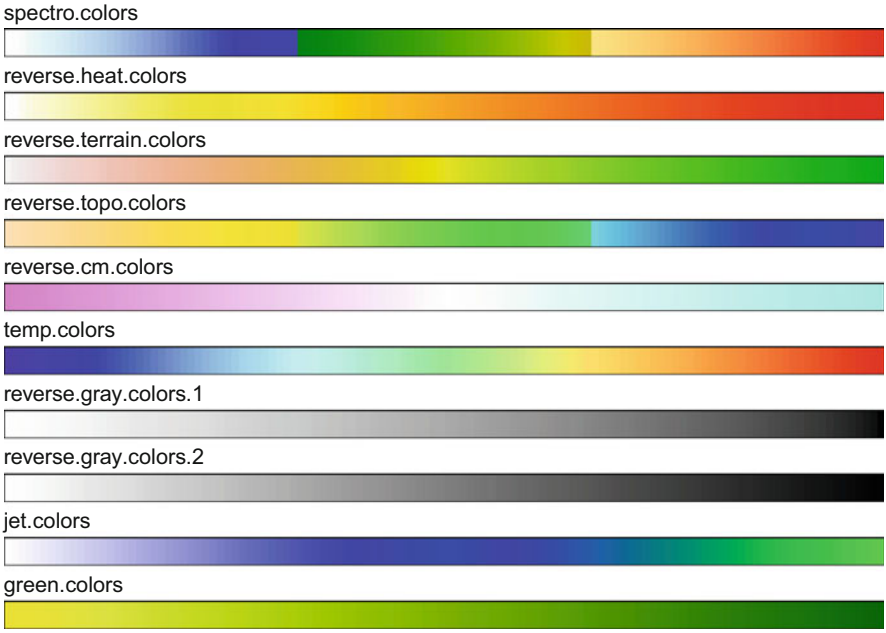


Fig. 11.20 Color palettes to be used with `spectro()`. Examples of different colour palettes for the amplitude scale of a spectrogram. The `jet.colors` and `green.colors` palettes were obtained with `colorRampPalette()`. See text for details

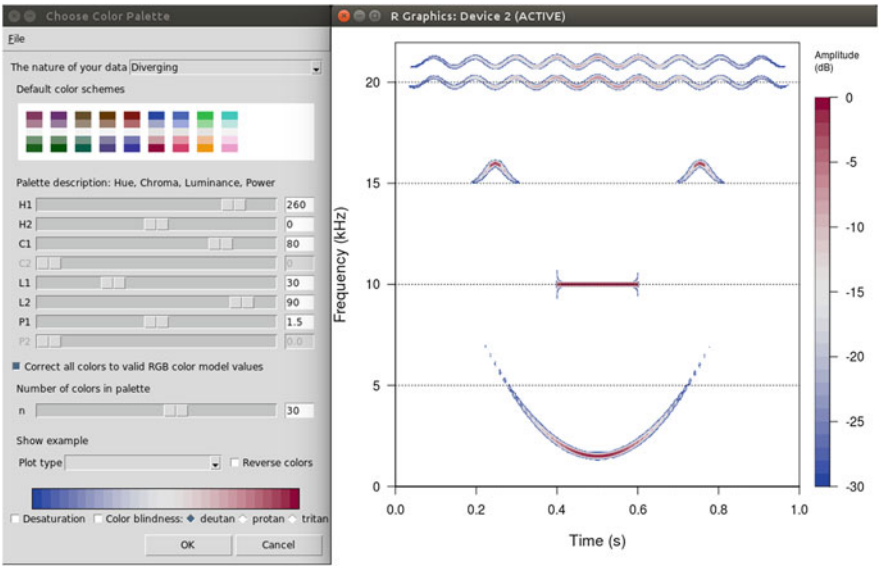


Fig. 11.21 Change of colour palette with the function `choose_palette()` of the package `colorspace`. This screenshot shows the interactive tool to select a colour palette according to several parameters and the result on the `face` spectrogram. Operating system: Ubuntu

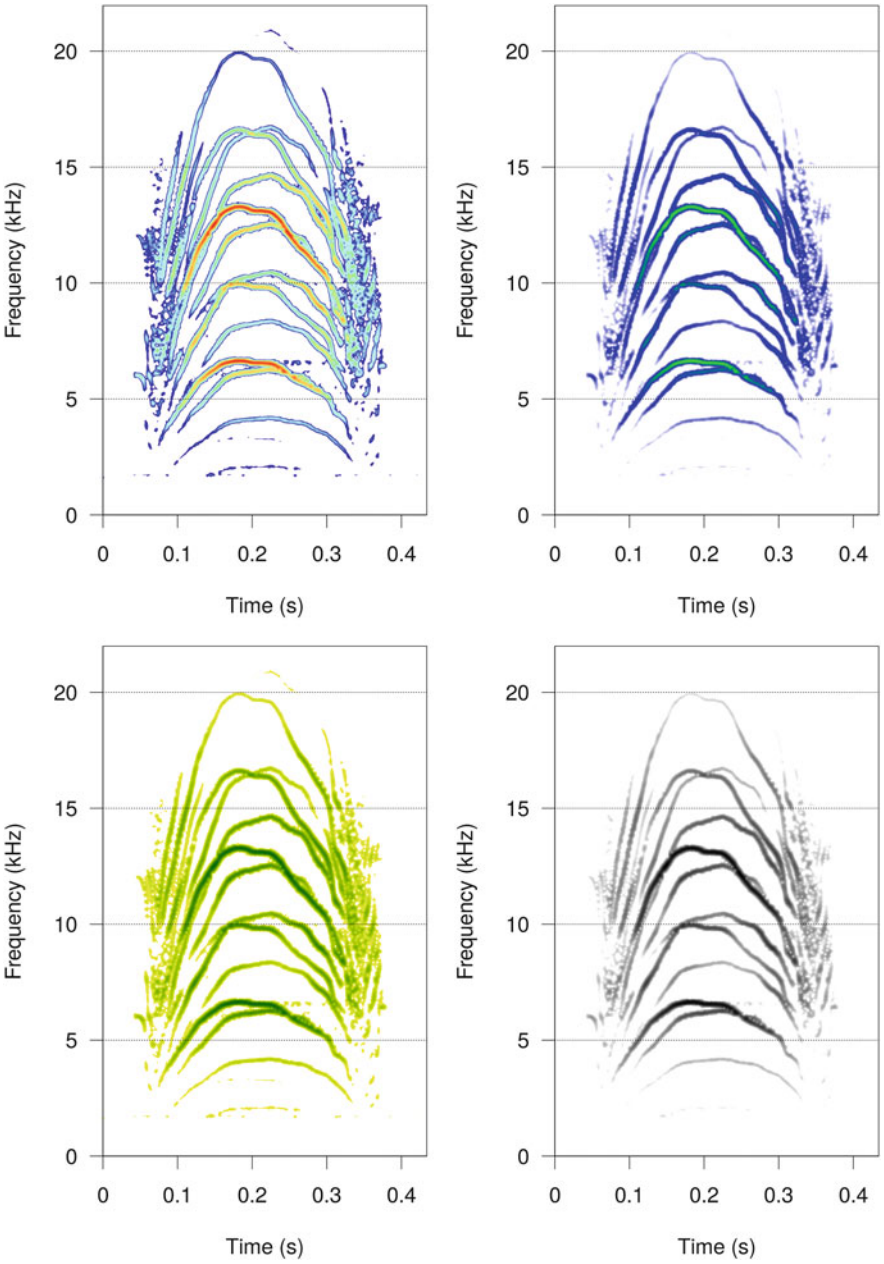


Fig. 11.22 Different colur palettes with `spectro()`. The spectrogram of `cockroach` was obtained with the palettes `temp.colors`, `jet.colors`, `green.colors`, and `reverse.gray.colors`. STDFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

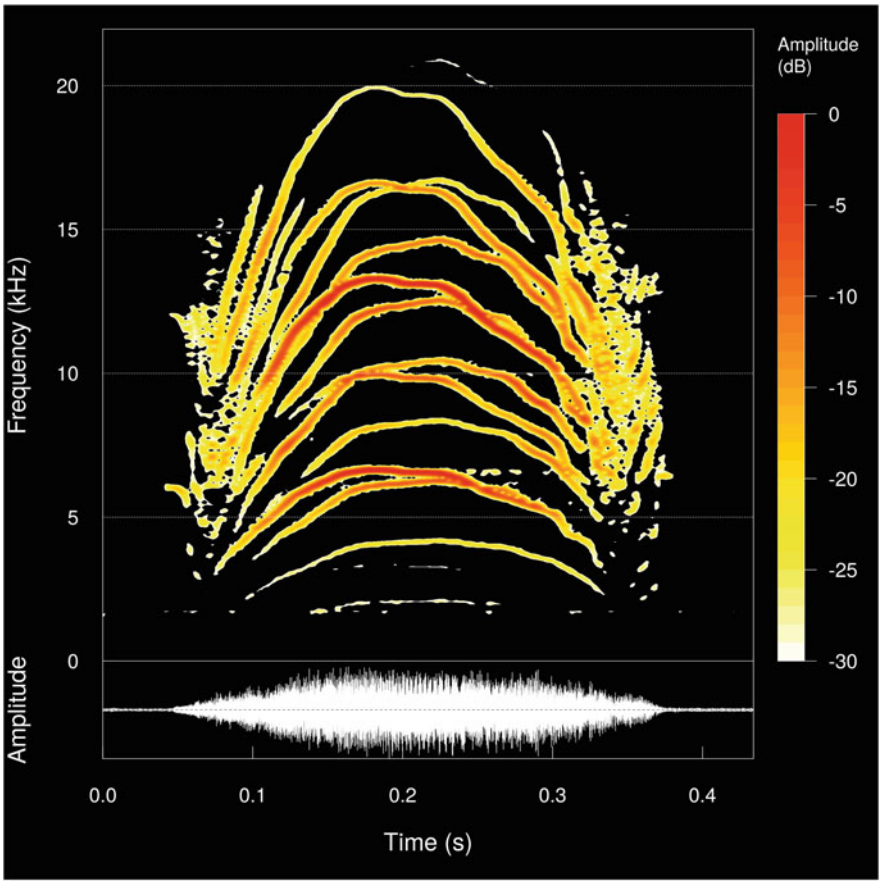


Fig. 11.23 Color changes with `spectro()`. The colors of the grid, the axes, the labels, and oscillogram are set to white when the background is turned to black. The palette was also changed for a better contrast with the background

`reverse.terrain.colors`, `reverse.topo.colors`, `reverse.cm.colors` corresponding to the reverse of `heat.colors`, `terrain.colors`, `topo.colors`, `cm.colors` (Fig. 11.20). The palette `temp.colors` is a palette joining "cold" and "hot" colors. The palettes `reverse.gray.colors.1` and `reverse.gray.colors.2` can be used for a gray level plot. The base function `colorRampPalette()` greatly facilitates the creation of new palettes. The following examples show how to generate two new palettes, one varying from white to green through blue and another one going from yellow to dark green:

```
jet.colors <- colorRampPalette(c("white", "blue", "green"))
green.colors <- colorRampPalette(c("yellow", "darkgreen"))
```

Another possibility is to take advantage of the package colorspace that proposes a wonderful interactive tool to choose and test a color palette (Fig. 11.21) with, for instance,:

```
library(colorspace)
pal <- choose_palette()
spectro(face, palette=pal)
```

The Fig. 11.22, which was obtained with the following code, shows four examples of palettes on cockroach:

colcont a vector of length 1 to define the color for the contour lines. This argument works only if cont=TRUE (Fig. 11.17).

colbg a vector of length 1 to define the background color of the complete plot, including the margins (Fig. 11.23).

colgrid a vector of length 1 to define the color of the grid. This argument works only if grid=TRUE (Fig. 11.23).

colaxis a vector of length 1 to define the color of time and frequency axes and amplitude scale (lines and labels) (See Fig. 11.23).

collab a vector of length 1 to define the color of time, frequency, and amplitude labels, that is, the colour of tlab, flab and alab (Fig. 11.23).

Figure 11.23 was obtained with:

```
col <- "white"
spectro(cockroach, ovlp=87.5, osc=TRUE,
        palette=reverse.heat.colors,
        colgrid=col, colaxis=col, collab=col,
        colwave=col,
        colbg="black")
```

11.7.1.7 Axes Arguments

The arguments to control the axes labels and values are:

cexlab the size of both time and frequency axis labels.

cexaxis the size of both time and frequency axis values.

tlab the label of the time axis.

flab the label of the frequency axis.

`alab` the label of the amplitude axis of the oscillogram. This argument works only if `osc=TRUE`.
`scalelab` the label of the amplitude scale.
`main` the label of the main title.
`scalefontlab` the font of the amplitude scale label.
`scalecexlab` the size of the amplitude scale label.
`axisX` a logical, to plot the time *x*-axis.
`axisY` a logical, to plot frequency *y*-axis.
`tlim` a numeric vector of length 2 to specify the lower and upper limits of the time *x*-axis. This argument works as the argument `xlim` of `plot()`. The two values should be expressed in s, such that `tlim=c(1,2)` displays the spectrogram between the positions 1 and 2 s.
`trel` a logical argument to control whether the values of the time *x*-axis vary along a relative scale (`TRUE`) or not (`FALSE`). This argument works only when `tlim` is not null. For instance, if `tlim=c(1,2)` the time *x*-axis shows values between 0 and 1 if `trel=FALSE` and values between 1 and 2 if `trel=TRUE`.
`flim` a numeric vector of length 2 to specify the lower and upper limits of the frequency *y*-axis limits. This argument works as the argument `ylim` of `plot()`. The two values should be expressed in kHz, such that `flim=c(2,6)` displays the spectrogram between 2 and 6 kHz.
`flimd` a numeric vector of length 2, this argument works in all points as `flim` but also tries to adapt automatically the values of `wl` (σ_t) and `ovlp` for an optimal time-frequency resolution. The new values are corrected by first computing the following factor:

$$\beta = \frac{f_s}{2000 \times (f_{upper} - f_{lower})}$$

with f_s, f_{upper}, and f_{lower} the sampling frequency frequency, the upper limit and the lower limit of the frequency limits expressed in Hz. This factor is then used to obtain the new `wl'` ($\sigma_t\prime$) and `ovlp'`:

$$\sigma_t\prime = \beta \times \sigma_t$$

$$ovlp\prime = 100 - \frac{ovlp}{\beta}$$

Figure 11.24, which shows axes changes, was obtained with:

```
spectro(cockroach, ovlp = 87.5,
        osc=TRUE,              # layout including the
        oscillogram            # a
        cexlab=1.3,            # size of axes labels and
```

(continued)

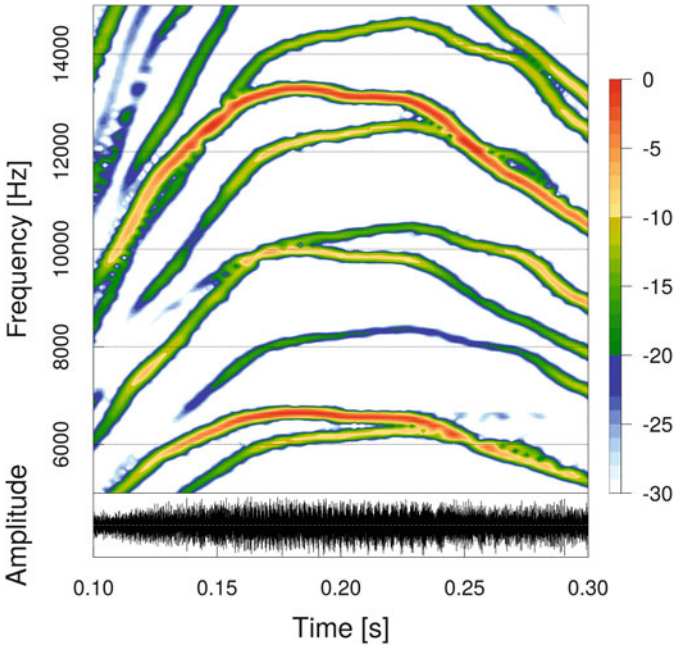


Fig. 11.24 Zoom-in and axes changes with `spectro()`. The spectrogram of `cockroach` is zoomed in in time and frequency, and changes are applied to the axes: the size of the labels and values are changed, and the unit of the frequency axis is changed to Hz

```
        values                  # a
        cexaxis=1.2,            # size of axes values
        tlab="Time [s]",        # new time label
        flab="Frequency [Hz]", # new frequency label
        scalelab="",            # no label for dB scale
        axisY=FALSE,            # no frequency y-axis
        tlim=c(0.1, 0.3),       # time zoom in
        trel=TRUE,              # relative time
        flim=c(5,15),           # frequency zoom in
        oma=rep(2,4))           # add outer margins
lab <- seq(6,14,length=5)       # new Hz frequency axis
values                          # a
axis(side=2, at=lab, labels=lab*1000, cex.axis=1.2, las=0)
```

11.7.1.8 Layout Arguments

The spectrogram display calls the function `layout()` when `osc=TRUE` and/or `scale=TRUE`. It can be then necessary to control some parameters setting the organization of the three plots (spectrogram, oscillogram, amplitude scale).

`widths` a numeric vector of length 2 to control the relative width of the layout columns. When `scale=TRUE`, the spectrogram figure is structured by two columns: one with the spectrogram and optionally the oscillogram beneath if `osc=TRUE` and a second column with the scale. By default the first column is six times larger than the second column such that `widths=c(6,1)`. However, these relative dimensions can be changed to give more or less space to the spectrogram. For instance, setting `widths=c(8,1)` widens the spectrogram and compresses the scale. Playing with this argument can be useful when printing long spectrogram into an image file (see Sects. 11.9.2 and 11.9.3).

`heights` a numeric vector of length 2 to control the relative heights of the layout rows. When `osc=TRUE` the spectrogram figure is structured by two lines: one with the spectrogram and optionally the scale on the side if `scale=TRUE` and a second row including the oscillogram. By default the first row is three times higher than the second row such that `heights=c(3,1)`. However, these relative dimensions can be changed to give more or less space to the spectrogram. For instance, setting `heights=c(5,1)` increases the space allocated to the spectrogram and reduces the space for the oscillogram.

`oma` a numeric vector of length 4 to control the size of outer margins when either `scale=TRUE` or `osc=TRUE`. This argument corresponds to the argument `oma` of the base function `par()`. By default, there are no outer margins (`oma=rep(0,4)`), but it can be useful to add margins around the figure to add annotations with `mtext()`.

11.7.1.9 Graphical Adding

We have just explored how to change several graphical parameters, including how to modify the default axes. In the following example, we illustrate with one example the use of low-level plot functions to decorate a spectrogram by adding some graphical elements (see Sect. 3.3.9.3). The spectrogram is first plotted, and then additional points, arrows, and rectangles are used to highlight specific features of the sound. The positions of the different elements were defined using the function `locator()` (see Sect. 11.8.1) (Fig. 11.25):

```
spectro(cockroach, ovlp=87.5)
arrows(x0=0.18, y0=21, x1=0.18, y1=20, len=0.1, lwd=2)
text(0.2, 21.25, "FM inflection point")
```

(continued)

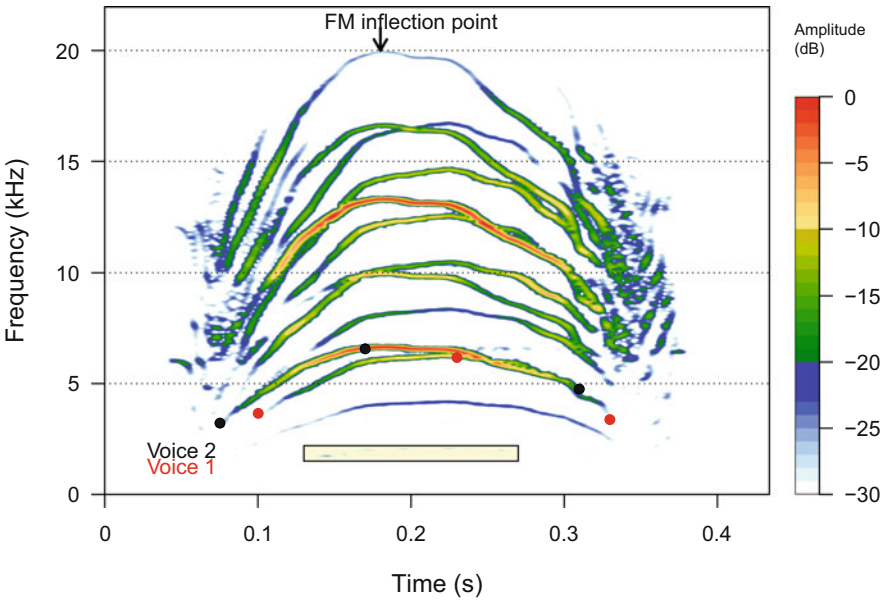


Fig. 11.25 Spectrogram decoration. The spectrogram of `cockroach` obtained with `spectro()` is decorated with the low level plot functions `arrows`, `text`, `points`, and `rect`

```
text(0.05, 1.25, "Voice 1", col=2)
text(0.05, 2, "Voice 2")
points(x=c(0.075, 0.17, 0.31), y=c(3.22, 6.57, 4.76), pch=19)
points(x=c(0.1, 0.23, 0.33), y=c(3.66, 6.18, 3.38), pch=19, col=2)
rect(xleft=0.13, ybottom=1.5,
     xright=0.27, ytop=2.2, col=rgb(1,1,0,0.25))
```

11.7.2 External Computing of the Short-Time Fourier Transform

The computation of the spectrogram, which is of the short-time discrete Fourier transform, can be quite time-consuming, in particular for a long sound.

The computation process can be speed up by avoiding the import of the `.wav` file(s) and by computing externally the short-time discrete Fourier transform using external tools that run faster than `R`. The function `stft.ext()` does this task by

reading externally the sound file with the C library libsndfile[1] and applying the C library fftw3[2] to compute the successive DFTs. All the process is therefore achieved externally, but the process and the results are controlled from R. Gain in process time is significant when analyzing a file longer than 10 min and when handling a group of more than 1000 files.

The function stft.ext() has less arguments than her sister function spectro(). The input and analysis arguments are file to specify the .wav file to treat, wl and ovlp for the Fourier parameters, norm to normalize the STDFT matrix to a maximal value of 1, dB to obtain values dB with a maximum value set to 0. There is no argument for zero-padding and no argument to specify the Fourier window shape, a Hanning window being used by default. The function includes an additional argument, named mean, to compute the average spectrum, that this the mean of the STDFT matrix according to the columns (see Sect. 11.14). The logical argument verbose can be set to TRUE to print some information on the file processed and on the computation. The value returns either a matrix corresponding to the STDFT matrix or a list containing the matrix ($amp) and the mean spectrum ($mean). The following examples show the use of stft.ext() with different argument combinations on the file containing a sound that was acquired in the tropical forest of the Kaw mountain in French Guiana. The sound, which was sampled at 44,100 Hz during 60 s, is stored in the file forest.wav:

```
forest <- readWave("sample/forest.wav")
forest

Wave Object
Number of Samples:      2646000
Duration (seconds):     60
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

The results of stft.ext() on "forest.wav" are quickly explored with str() and range(). Note that the function does not apply any correction on the data, that is, the raw values used are the values directly obtained by the DFT with neither multiplying the values by 2 to take into account the mirror property of the DFT nor by applying an amplitude or energy correction to the Fourier window function (e.g., Hanning window):

[1] http://www.mega-nerd.com/libsndfile/

[2] http://www.fftw.org/

```
file <- "sample/forest.wav"
## no normalization, linear scale, meta-information file printed
res <- stft.ext(file=file, verbose=TRUE)
File name = 'sample/forest.wav'
Number of samples = 2646000
Sampling rate (Hertz) = 44100
Duration (seconds) = 60.000000
Window length (number of samples)= 512
Window overlap (percentage) = 0.000000
str(res)
 num [1:256, 1:5167] 0.057 0.0606 0.0522 0.0262 0.0131 ...
range(res)
[1] 8.900688e-06 7.153301e+00
## normalization, linear scale
res <- stft.ext(file=file, norm=TRUE)
str(res)
 num [1:256, 1:5167] 0.00796 0.00848 0.0073 0.00366 0.00183 ...
## normalization, dB scale
res <- stft.ext(file=file, dB=TRUE)
range(res)
[1] -118.1017    0.0000
## STDFT and mean spectrum, normalization, linear scale
res <- stft.ext(file=file, norm = TRUE, mean = TRUE)
str(res)
List of 2
 $ mean: num [1:256] 0.1043 0.0986 0.0595 0.0378 0.0292 ...
 $ amp : num [1:256, 1:5167] 0.00796 0.00848 0.0073 0.00366 0.00183 ...
range(res$mean)
[1] 0.003821462 1.000000000
range(res$amp)
[1] 1.244277e-06 1.000000e+00
```

11.7.3 Inverse Short-Time Fourier Transform

The basic process of the inverse short-time Fourier transform (ISTFT, ISTDFT) is to compute the inverse Fourier transform (IFT, IDFT) of the frequency function obtained for each window of the STDFT. The IFT is divided by the window function (e.g., Hanning window). However, such raw operation often leads to distorted signals in the time domain.

The overlap-add (OLA) method consists in adding overlapping windows such that the window function has no effect on the process (Quatieri 2002). The OLA method can be expressed as:

$$x(t) = \frac{1}{2\pi w(0)} \int_{-\infty}^{\infty} F(\tau, \omega) e^{i\omega m} \mathrm{d}\omega$$

where $w(0) = \sum_{n=-\infty}^{\infty} w(n)$ and where $x(t)$ is the recovered signal that can be different or not from the original signal $s(t)$ if modifications were or were not applied on the STDFT matrix.

The discrete form of the OLA equation is:

$$x[n] = \frac{1}{w(0)} \sum_{-\infty}^{\infty} \left[\frac{1}{N} \sum_{k=0}^{N-1} F[m,k] e^{i \frac{2pi}{N} kn} \right]$$

The OLA operation is implemented in the function `istft()` of `seewave`. The function waits the complex form of the STDFT, that is, the complex values of the Fourier transform computed for each window. In addition, to avoid any distortion in the reconstructed signal, the STDFT and ISTDFT should be computed such that the window overlap parameters should be:

$$ovlp = 100 \times \left(1 - \frac{1}{4n}\right)$$

with n being a positive integer. The argument `ovlp` of `istft()` has a default value of 75% that works in most cases. To avoid any issue, the arguments `wl`, `ovlp`, and `wn` should be similar when calling `spectro()` and `istft()`. The following is a reconstruction of `tico`:

```
# parameters
wl <- 1024
ovlp <- 75
wn <- "hanning"
# STDFT = getting the amplitude complex values
data <- spectro(tico, wl=wl, ovlp=ovlp, wn=wn,
                plot=FALSE,
                norm=FALSE, dB=NULL, complex=TRUE)$amp
# ISTFT = reconstruction of the original signal
recons <- istft(data, f=22050, ovlp=ovlp, wl=wl, wn=wn,
             out="Wave")
```

Note that the function `istft()` does not preserve the energy of the signal so that the RMS of the original and reconstructed signals differ:

```
rms(tico@left)
[1] 3927.516
rms(recons@left)
[1] 3.842945
```

11.8 Measurements and Annotations on the Spectrogram

To take measurements on a spectrogram is not a good idea as the time and frequency precision of the measurements are severely constrained by the STDFT parameters and the Heisenberg uncertainty principle. However, if necessary, some options, from very rough to more fancy, are available to take time and frequency features and annotate spectrograms.

11.8.1 Simple Measure

The function `locator()` can be used to fetch the coordinates of points of interest:

```
spectro(cockroach, ovlp=87.5)
res <- locator()
```

Here three points where localized. The function `locator()` returns the coordinates in a two-item list, `$x` being the time coordinates in s and `$y` the frequency coordinates in kHz:

```
res
$x
[1] 0.07542123 0.19255667 0.32542672

$y
[1] 3.460243 6.690117 3.972296
```

11.8.2 Fancy Measure and Annotation

11.8.2.1 Function of the Package `warbleR`

The package `warbleR` offers an option for time measurements and annotations on the spectrogram. The dedicated function, named `manualoc()`, generates an interactive spectrogram based on `spectro()`.

There are two main actions. The user can first zoom in time by clicking on two time positions t_1 and t_2 with $t_1 < t_2$. The view is then refreshed and the new spectrogram is displayed. The user can also add selections on the spectrogram by clicking on two time positions t_1 and t_2 with $t_1 > t_2$. This action adds vertical dotted

red lines at t_1 and t_2 and a series number in between. This action can be repeated *ad libitum* to select as many as desired selections.

The function `manualoc()` has been clearly developed for the batch processing of several sound files. This explains why the function does not wait for a single `R` object as an input but directly works on all the `.wav` (not `.mp3`) files stored in the working directory. This implies to use `setwd()` to select the appropriate directory where the files of interest are grouped. The treatment of several files in series motivated as well the creation of fancy-colored buttons overlaid on the top-right area of the spectrogram. These buttons facilitate the navigation within and between files. The buttons allow the following actions:

- Full view: to operate a complete zoom out so that the spectrogram of the complete sound is displayed.
- Previous view: to display the previous view.
- Stop: to get out of the interactive session and go back to the console.
- Next rec: to jump to the next recording stored in the working directory.
- Play: to listen to the current sound displayed.
- Del-sel: to delete the selections.

After, having selected the appropriate directory with `setwd()`, `manualoc()` can be run directly with no arguments:

```
manualoc()
```

The function displays then the first sound file, and zoom and time segments selections can be operated. The function has several arguments to tune the display: the Fourier window length (`wl`), the Fourier window function (`wn`), the frequency limits that are set by default for bird vocalizations between 0 and 12 kHz (`flim`), and the duration of the section displayed (`tdisp`). We can then tune the display with the following code:

```
manualoc(wl=256, wn="hamming", flim=c(0,22.05), tdisp=20)
```

The Fig. 11.26 shows the display obtained for the sound sample `forest` and eight selections around sound of interests.

These eight selections were saved automatically in a `.csv` file named `manualoc_output.csv` stored in the working directory. The file contains a line for each time selection and seven columns. The definition of the columns is `sound.files` for the names of the files analyzed, `selec` for the series number of selections, `start` for the start time in second of the selection, `end` for the end time in s of the selection , `sel.comment` for the optional comment attributed

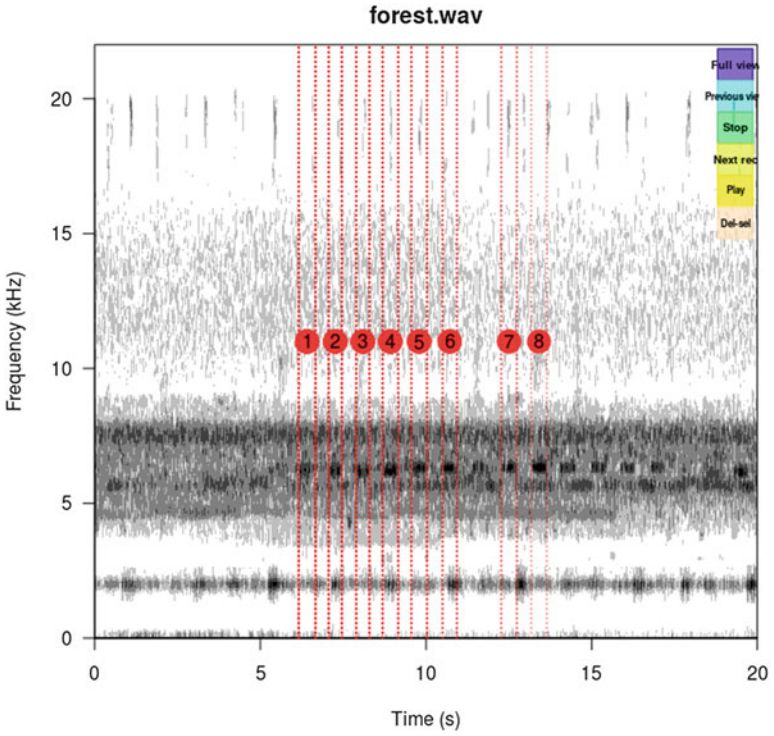


Fig. 11.26 Spectrogram selections with `manualoc()`. Manual annotations were added by clicking on the spectrographic display. Here eight regions of interest were delimited

to the selection, and `rec.comment` for the optional comment attributed to the recording (argument `reccomm=TRUE`).

The data can be imported into `R` with `read.csv()`, here saved in a directory `data`:

```
meas <- read.csv("data/manualoc_output.csv")
head(meas)
  sound.files selec     start       end sel.comment
1  forest.wav     1  6.121480  6.736457          NA
2  forest.wav     2  7.098208  7.532310          NA
3  forest.wav     3  7.966411  8.364337          NA
4  forest.wav     4  8.798439  9.196365          NA
5  forest.wav     5  9.594291 10.100743          NA
6  forest.wav     6 10.571020 11.005121          NA
  rec.comment
1          NA
2          NA
```

(continued)

```
3           NA
4           NA
5           NA
6           NA
```

11.8.2.2 Function of the Package monitoR

monitoR is a package primary dedicated to the automatic detection and identification of template sounds in recordings (Katz et al. 2016b). The main utilities of monitoR are considered in Sect. 17.4, but the package embeds a nice spectrogram function to explore long sound. viewSpec() uses spectro() but plots the results using image() with the argument useRaster to TRUE when spectro() calls filled.contour() that smooths the image. This has the inconvenient to produce quite rough images but the great advantage to be very fast and so to opens the opportunity to explore quickly a long sound with time navigation tools. We first load the package:

```
library(monitoR)
```

and then do an elementary test with forest:

```
viewSpec(forest)
```

viewSpec() plots the first 30 s of the sound from 0 to 12 kHz, but this can changed with the arguments start.time, page.length, and freq.lim. Here we display a 20 s spectrogram starting at 10 s over a [0, 22.05] kHz frequency range:

```
viewSpec(forest, start.time=10, page.length=20,
         frq.lim=c(0,22.05))
```

The beauty of viewSpec() is the navigation tool offered by the argument interactive. Turning this argument to TRUE enables the options to page through spectrograms, play, zoom in and out in time, extract segment of interests and save them as .wav files, and change the STDFT parameters. If we call:

```
viewSpec(forest, start.time=10, page.length=20,
         frq.lim=c(0,22.05), interactive=TRUE)
```

then the console prints the following intuitive commands to be run with the keyboard:

```
Reading file...
Enter:
  n(m) for next page,
  b(v) for previous page,
  p to play,
  z to zoom in,
  x to zoom out,
  s to save page as wave file,
  c to change spectrogram parameters,
  q to exit
```

The function is even more fancy with the use of the `annotate` argument. This option allows manual annotations of specific sound events by drawing and labeling a rectangle around a region of interest.

```
viewSpec(forest, start.time=10, page.length=20,
         frq.lim=c(0,22.05), interactive=TRUE, annotate=TRUE)
```

The console prints the same message as with `interactive=TRUE` but this time with two additional options: `a` to add annotations, and `d` to delete annotations.

The process is again very intuitive, the user just needs to follow the instructions provided in the console. Figure 11.27 was obtained by selecting and labeling three regions of interest.

The results can be saved in a `.csv` file that can be then read back by giving the file path in the argument `anno`.

The annotations can be read in `R` with the function `read.csv()` for further statistical analysis. The annotations were here saved in a directory `data`:

```
annotations <- read.csv("data/forest_annotations.csv")
annotations
  start.time end.time min.frq max.frq               name
1     16.157   20.369  1.4862  2.4967        unidentified
2     20.408   22.378 16.9071 20.9930               bats
3     21.258   26.745  3.4193  3.9905 Lerneca_fuscipennis
```

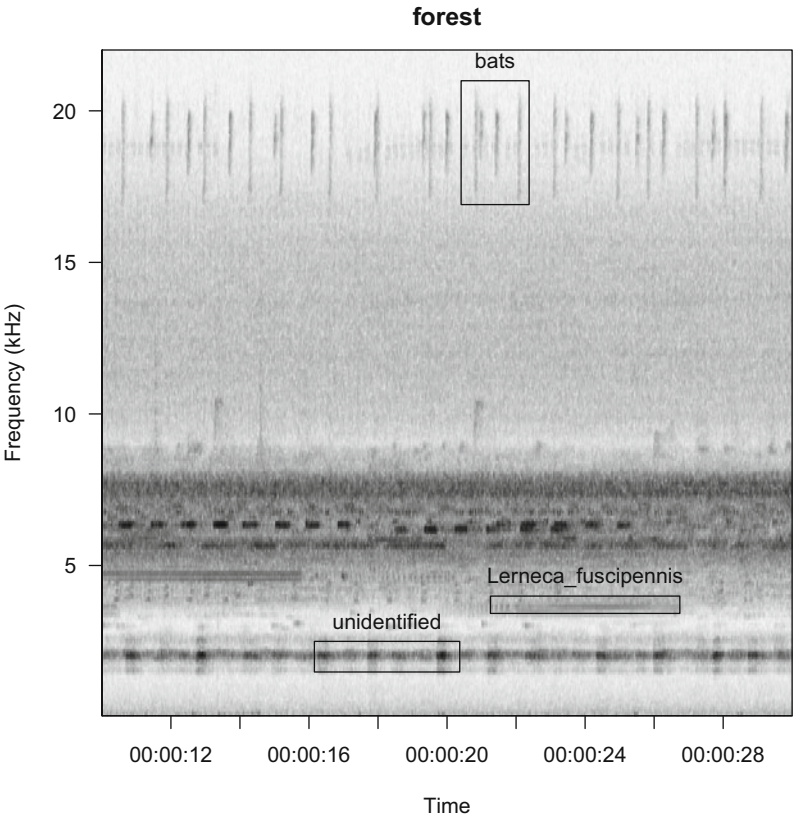


Fig. 11.27 Spectrogram annotations with `viewSpec()`. Three regions of interest were delimited, saved, and read back with `viewSpec()`

or can be used directly with `viewSpec()` to read them back and display them on the spectrogram and then to carry on the process of annotation:

```
viewSpec(forest, start.time=10, page.length=20,
         frq.lim=c(0,22.05), annotate=TRUE,
         anno='data/forest_annotations.csv')
```

11.8.3 Automatic Parametrization

The functions `locator()`, `manualoc()`, and `viewSpec()` all rely on manual measurements, a process that is not always possible when handling numerous files. Another option is to try to parametrize automatically the spectrogram along both

time and frequency dimensions. The principle of "acoustat", which was developed by Fristrup and Watkins (1992), consists in describing the time and frequency profiles of the STDFT matrix with summary statistics of central tendency (median, percentile) and dispersion (interpercentile range).

The following workflow of "acoustat" is:

1. computes the STDFT matrix;
2. computes an aggregation function such as the sum, the variance, or the standard-deviation across rows and columns of STDFT matrix. This aggregation process results in two components: (1) the time contour which is a kind of time envelope and (2) the frequency contour (Fig. 11.28);
3. transforms each contour into a probability mass function (PMF) and then into a cumulative distribution function (CDF);
4. computes the following features for each CDF: the median (M), the initial percentile (P1), the terminal percentile (P2), and the interpercentile range (IPR). P1, P2, and IPR are defined using a fraction parameter (fraction) that sets the percent of the contour amplitude to be captured by the initial and terminal percentile values. A fraction of 50% would result in the familiar quartiles and interquartile range. An energy fraction of 80% would return the 10th and 90th percentile values, and the width of the range in between.

The eponymous function `acoustat()` computes these parameters and plots optionally the contours and the statistics in a two-frame graphic. The following example uses most of default parameters except the `ovlp` argument that set, as usually, the percentage of overlap between successive Fourier windows (Fig. 11.29):

```
res <- acoustat(cockroach, ovlp=87.5)
```

Both contours and all statistics are returned in a list made of 10 items

```
str(res)
List of 10
 $ time.contour: num [1:292, 1:2] 0 0.00149 0.00298 0.00447 0.00596 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:2] "time" "contour"
 $ freq.contour: num [1:256, 1:2] 0 0.0861 0.1723 0.2584 0.3445 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:2] "frequency" "contour"
 $ time.P1     : num 0.0775
 $ time.M      : num 0.215
 $ time.P2     : num 0.353
```

(continued)

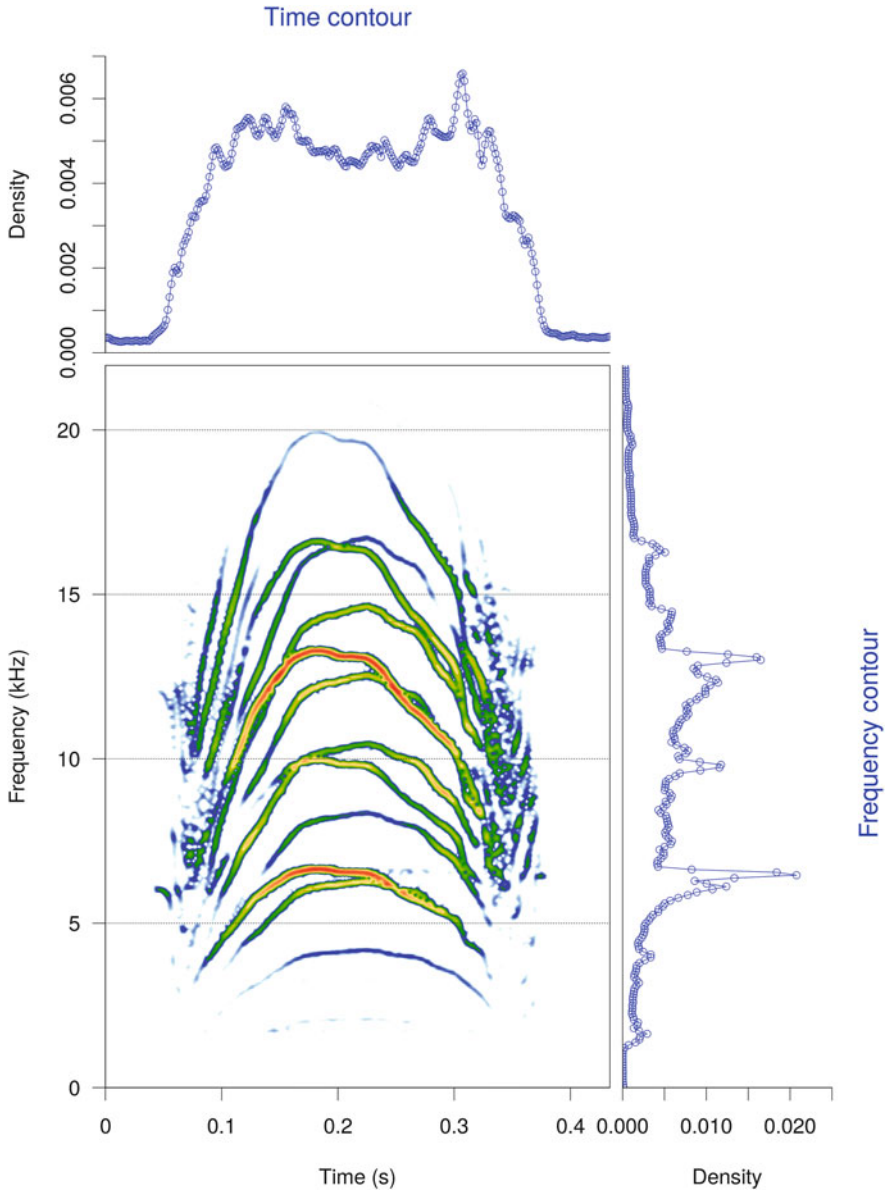


Fig. 11.28 The main principle of acoustat. One of the most important stages in the process is to estimate a time and a frequency contour through an aggregation of the columns and rows of the STDFT matrix. The example, here based on `cockroach`, shows the spectrogram and the contours. The contours are drawn with a line and points to show the discretization due to the STDFT. STDFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

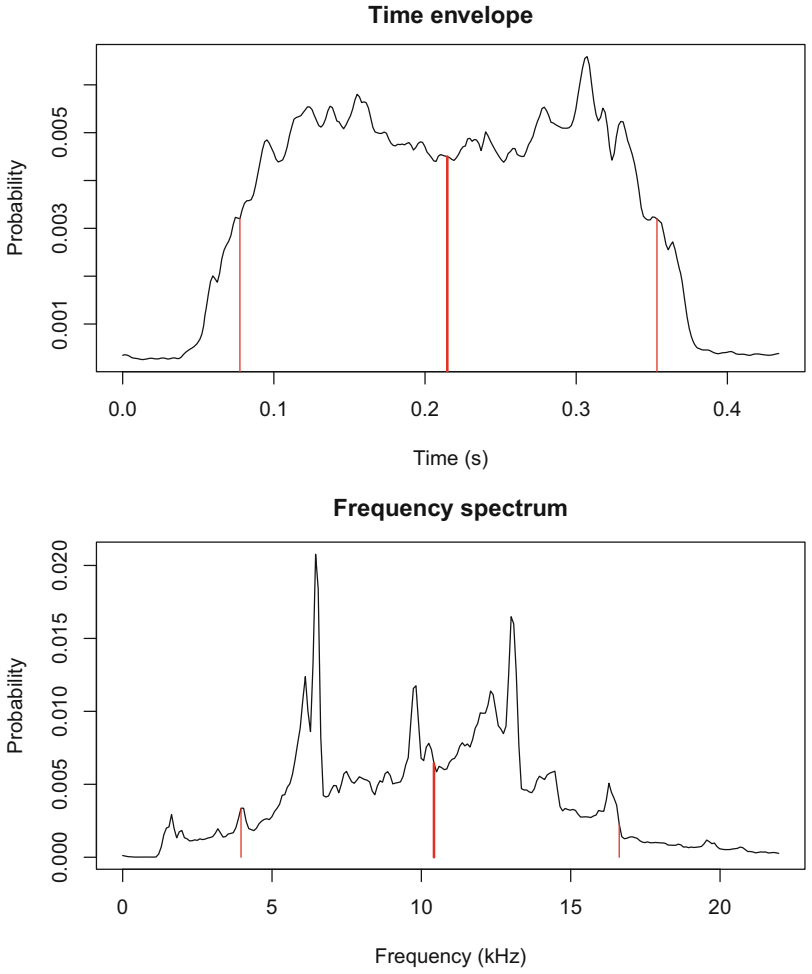


Fig. 11.29 Parametrization of the spectrogram with `acoustat()`. Visual display of the function `acoustat()` with the time envelope (top) and the frequency contour (bottom). The median and quartiles are indicated with vertical red segments

```
$ time.IPR    : num 0.276
$ freq.P1     : num 3.96
$ freq.M      : num 10.4
$ freq.P2     : num 16.6
$ freq.IPR    : num 12.7
```

The function `acoustat()` gives access to two other STDFT parameters (`wl` and `wn`) and also opens the possibility to apply a time and/or a frequency selection

with the arguments `tlim` and `flim`.The aggregation function can be tuned with the argument `aggregate` which is set to the sum by default and with the argument `fraction` to set the percentiles. Here is a tuned example with the standard deviation instead of the sum, and quartiles instead of 10th and 90th percentiles:

```
res <- acoustat(cockroach,
                wl=1024, ovlp=87.5,             # STDFT parameters
                tlim=c(0.1,0.3), flim=c(5,15), # zoom
                aggregate=sd, fraction=50,      # acoustat parameters
                plot=FALSE)                     # no plot
```

11.9 Complex Display and Printing

11.9.1 Multi-Spectrogram Graphic

For comparison purposes, it might be useful to plot several spectrograms combined in a single graphic display. This can be achieved using `layout()`. However, the function `spectro()` uses by default `layout()` to display the amplitude scale color and/or the oscillogram. Because, it is not possible to embed two `layout()` calls, it is not possible to build a plate of spectrograms with the amplitude scales and/or the oscillogram. Nonetheless, it is still possible to combine different "naked" spectrograms, as illustrated in Figs. 11.13, 11.14, 11.19, and 11.22, and then to add manually a shared amplitude scale as exemplified here with the spectrograms of four sounds (Fig. 11.30):

```
# overvall parameters
cex <- 1.75
m <- matrix(c(1,0,2:5), byrow=TRUE, ncol=2)
layout(m, heights=c(0.3,1,1))
par(oma=c(2,2,1,0))
# plot 1: shared amplitude colour scale
par(mar=c(1,2,4,1))
dBscale(side=3, collevels=seq(-30,0,1), textlab="")
# plot 2: tico
par(mar=c(3,2,0,1))
spectro(tico, scale=FALSE, tlab="", flab="")
text(x=0.15,y=10.4, label="(A)", cex=cex)
# plot 3: orni
spectro(orni, scale=FALSE, tlab="", flab="")
text(x=0.06,y=10.4, label="(B)", cex=cex)
```

(continued)

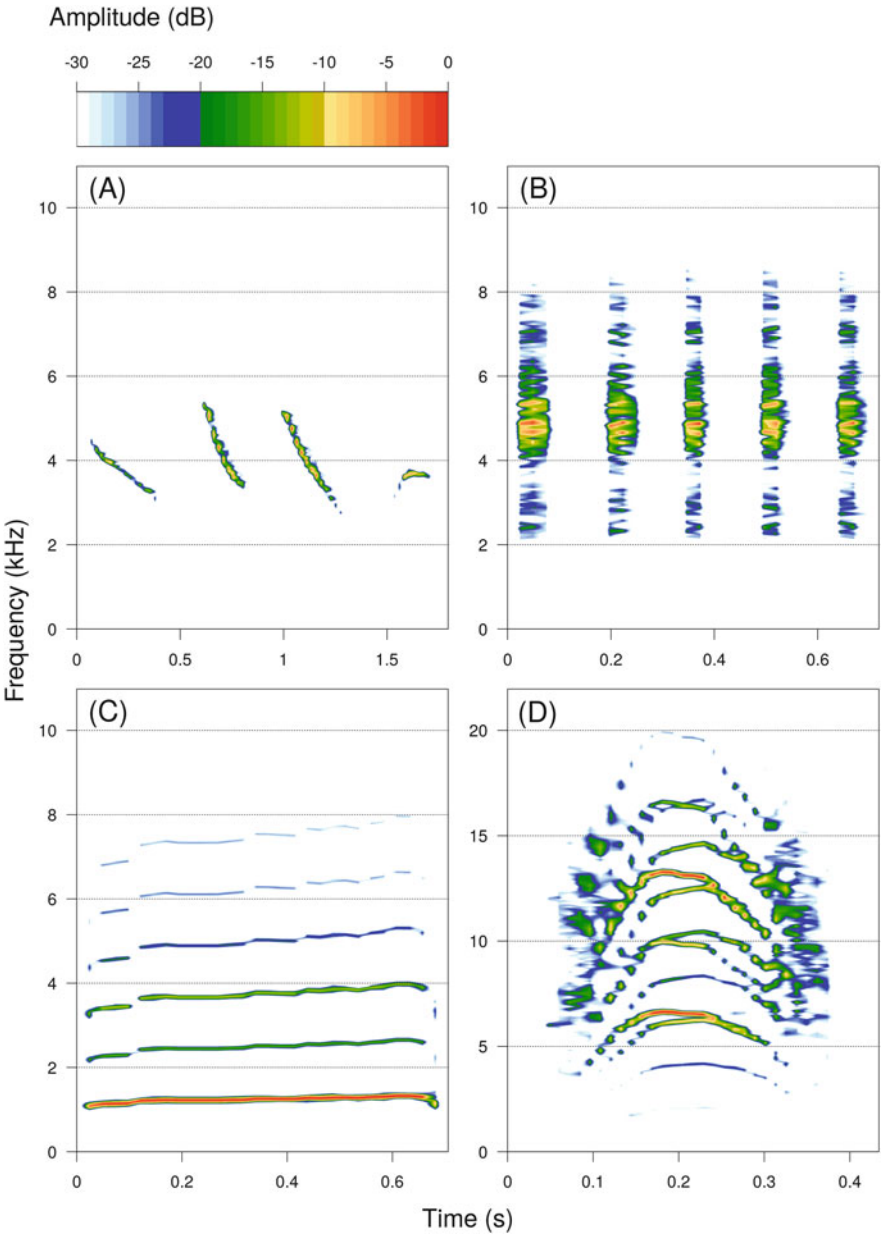


Fig. 11.30 Several spectrograms in a single graphic display. The spectrogram of `tico`, `orni`, `peewit`, and `cockroach` are arranged to be all plotted in a single graphic display. The amplitude color scale is added with the function `dBscale()`

```
# plot 4: peewit
spectro(peewit, scale=FALSE, tlab="", flab="")
text(x=0.06,y=10.4, label="(C)", cex=cex)
# plot 5: cockroach
spectro(cockroach, scale=FALSE, tlab="", flab="")
text(x=0.035,y=20.75, label="(D)", cex=cex)
# axes labels in the outer margins
mtext(side=1, text="Time (s)", out=TRUE)
mtext(side=2, text="Frequency (kHz)", out=TRUE, line=0.5, las=0)
mtext(side=3, text="Amplitude (dB)", out=TRUE, at=0.11, line=-1)
```

11.9.2 Printing in a File

As seen in Sect. 3.3.9.6, `R` offers several options to print a graphic in a file, either a vectorial or a raster format. The vectorial format, such as the `.pdf` format, is often preferred for its high quality. However, this format might not be adapted to spectrograms which may contain numerous points and therefore may generate very heavy files. The most convenient solution seems to use the loesless raster format `.png` with a high resolution. Usually setting the function `png()` as in the following example returns an appropriate graphic with a square shape:

```
png("cockroach.png", width=1200, height=1200, point=24)
spectro(cockroach, ovlp=87.5)
dev.off()
```

If the sound is particularly long, it might be necessary to change the dimensions of the spectrogram to make it wider as in the following test with the sound `forest`. Here the relative width of the spectrogram and of the amplitude color scale needs to be adjusted with the argument `widths` of `spectro()`, and the levels of the colors are also changed to make apparent distant (Fig. 11.31):

```
png("forest.png", width=2000, height=1200, point=24)
spectro(forest, collevels=seq(-40,0,1), widths=c(8,1))
dev.off()
```

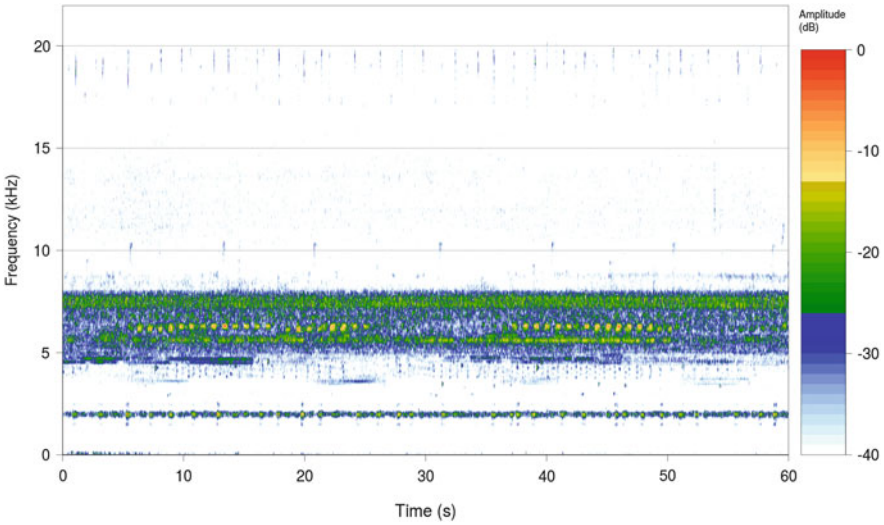


Fig. 11.31 Saving a spectrogram in a raster file. This image was produced using the function `png()` to print the spectrogram of `forest` into a `.png` file. The settings of `png()` and `spectro()` were adjusted to widen the spectrogram

11.9.3 Long Spectrogram Graphic

When a sound sample is particularly long, lasting more than 60 s, it might be more efficient to print and save the result in a file rather than to display it on the graphic device. The process is indeed faster. This idea is behind the principle of the function `lspec()` of the package `warbleR`. This fancy and very useful function generates long spectrograms, produced by `spectro()`, expanded over several lines and potentially over several raster `.jpeg` files, named pages. The function can handle a single or a series of files all stored in default working directory. The file names, potential selections determined with `manualoc` (see Sect. 11.8.1), the start and end times of the display, and the optional selection comments and recording comments can be specified in a data frame with the respective following column names: `sound.files`, `selec`, `start`, `end`, `sel.comment`, and `rec.comment`.

For a basic use with a single `.wav` file, here with `forest.wav` file, the data frame is simple with only four columns:

```
X <- data.frame(sound.files="forest.wav", selec=NA,
                start=0, end=duration(forest))
X
  sound.files selec start end
1  forest.wav    NA     0  60
```

This information is then used to generate a long spectrogram divided in 6 rows (argument `rows`) with each row lasting 5 s (argument `sxrow`):

```
setwd("sample")
lspec(X=X, rows=6, sxrow=5)
```

It was first necessary to change the working directory with `setwd()` so that `lspec()` can find the `.wav` file. The function has generated two files, named `forest-p1.jpeg` and `forest-p2.jpeg` containing each 6 rows of spectrograms (Fig. 11.32).

Additional information provided in the data frame can be used to overlay on the spectrogram the selection limits, selection comments, and the recording comments. This process can be applied to a batch of files such that several long-annotated spectrograms can be generated with two lines of code only.

A more direct way, but without control on annotations, to produce the same images is to use the argument `flist` where the file(s) name(s) can be provided:

```
lspec(flist="forest.wav", rows=6, sxrow=5)
```

The main other arguments of `lspec()` are given in Table 11.2.

11.10 Dynamic Spectrogram

The function `dynspectro()` of `seewave` opens the possiblity to navigate along a sound in a similar way as the dynamic spectrum does (Sect. 11.1.1). Here successive STDFTs, or spectrograms, are computed and are plotted successively, thanks to a position control button created with the library `rpanel`. In the following example, we apply the function `spectro()` on `forest`. The size of the spectrograms is controlled in percentage of the total duration with the argument `slidframe` (Fig. 11.33):

```
library(rpanel)
dynspectro(forest, slidframe=15, osc=TRUE)
```

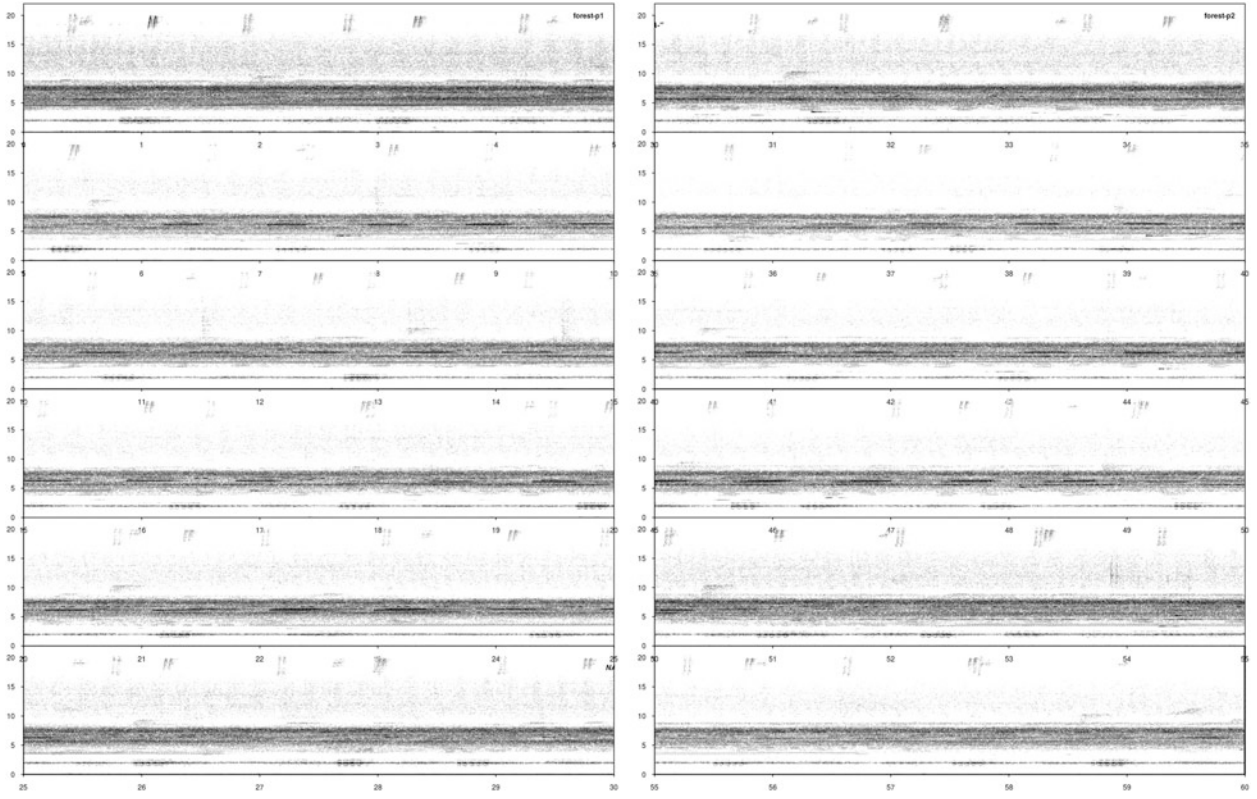


Fig. 11.32 Saving a long spectrogram in a series of raster files. These two images were produced using the function `lspec()` of `warbleR` to split and print the 60 s spectrogram of `forest`

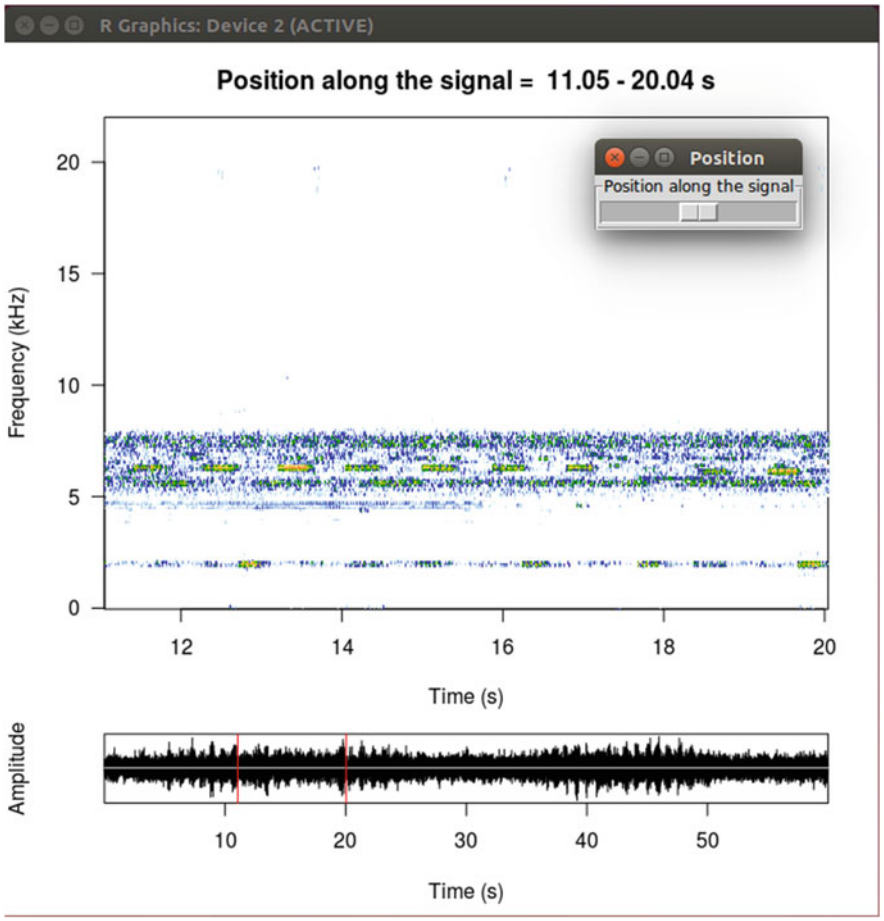


Fig. 11.33 Dynamic spectrogram. The function `dynspectro()` can be used to navigate along a long sound. A series of STDFT are computed along the signal, here the sound `forest`, for a given number of frames set with the argument `slidframe`. The screenshot here shows the STDFT computed for the frame between 11.05 and 20.04 s. Moving along the signal is made possible, thanks to the small control pop-up window entitled "Position." Operating system: Ubuntu

11.11 Movie

There are no way to create a movie directly within `R`. However, Marcelo Araya-Salas, the maintainer of `warbleR`, shared in his blog a very nice solution to produce an `.mp4` file of a spectrogram moving in phase in sound.[3]

[3] https://marce10.github.io/2016/12/12/Create_dynamic_spectro_in_R.html

The solution consists in saving a series of spectrograms as image files and associating afterward these files as a movie with the UNIX utility `ffmpeg`.[4] Here we adapt this brilliant idea to create an image with a static spectrogram but with a vertical cursor moving with sound.

The first step consists in creating a series of images that build the movie. The images contain the same "background," which is the spectrogram, and the cursor is added at successive positions related to the frame rate or frame per second fps parameter. To generate this series of images, we can use a `repeat` loop taking advantage that the function `jpeg()` can increment automatically the name of the output file:

```
# set the working directory
# where images and movie will be stored
setwd("image/movie")
# frame per second rate
fps <- 75
# repeat loop
jpeg("cockroach-movie-%02d.jpg",
     width=1100, height=1100, pointsize=20)
x <- 0
repeat{
  # spectrogram
  spectro(cockroach, scale=FALSE, osc=TRUE)
  # cursor
  abline(v=x, lty=2, col=2, lwd=2)
  x <- x + 1/fps
  # exit loop when exceeding the duration of the sound
  if(x > duration(cockroach)) break
  }
dev.off()
```

Once the images are generated, we call `ffmpeg` through the `system()` function (see Sect. 3.3.11) to first create the video file `cockroach_movie.mp4`:

```
system("ffmpeg -loglevel quiet -framerate 75
-i cockroach-movie-%02d.jpg -c:v libx264
-profile:v high -crf 2 -pix_fmt yuv420p
cockroach_movie.mp4")
```

We then associate the audio file `Elliptorhina_chopardi.wav` and generate a new and ultimate `.mp4` file named `cockroach_movie_with_sound.mp4`:

[4] https://ffmpeg.org/

```
system("ffmpeg -loglevel quiet -i cockroach_movie.mp4
-i ../../sample/Elliptorhina_chopardi.wav
-vcodec libx264 -acodec libmp3lame
-shortest cockroach_movie_with_sound.mp4")
```

As we are tidy people, we clean up the directory by removing unnecessary files, and we reset the working directory:

```
file.remove(c("cockroach_movie.mp4", dir(pattern="*.jpg")))
[1] FALSE
setwd("../../")
```

11.12 Waterfall Display

The waterfall display is another solution to plot in two dimensions a three-dimensional object. The waterfall is a perspective plot that shows each frequency spectrum of the STDFT slightly offset from its neighbor on a diagonally oriented time axis. Frequency is represented along the *x*-axis, amplitude along the *y*-axis, and time along a diagonal. The function `wf()` of `seewave` can produce such a graphic with different options to control STDFT parameters (arguments `wl`, `ovlp`, `zp`, `wn`, `dB`, `dBref`, and `fftw` similar to those of `spectro()`), the orientation or perspective (arguments `hoff` and `voff`), and the global appearance (arguments `col`, `density`, `border`, `lines`, and `lwd`). The input is, as usual, an object describing a sound (argument `wave`), but it can also be any matrix (argument `x`). Figure 11.34 groups four examples of waterfalls produced with the following code:

```
# layout
par(mfrow=c(2,2), mar=c(4.5,4.5,1,1))
wl <- 512 ; ovlp <- 50  # STDFT parameters
hoff <- 1; voff <- 5    # perspective parameters
# plot 1: default display
wf(cockroach, wl=wl, ovlp=ovlp)
# plot 2: changes in the orientation of the waterfall
wf(cockroach, wl=wl, ovlp=ovlp,
    hoff=hoff, voff=voff)
# plot 3: lines plot
wf(cockroach, wl=wl, ovlp=ovlp,
    hoff=hoff, voff=voff,
    lines=TRUE, col=1)
# plot 4: surface plot with home-made palette
```

(continued)

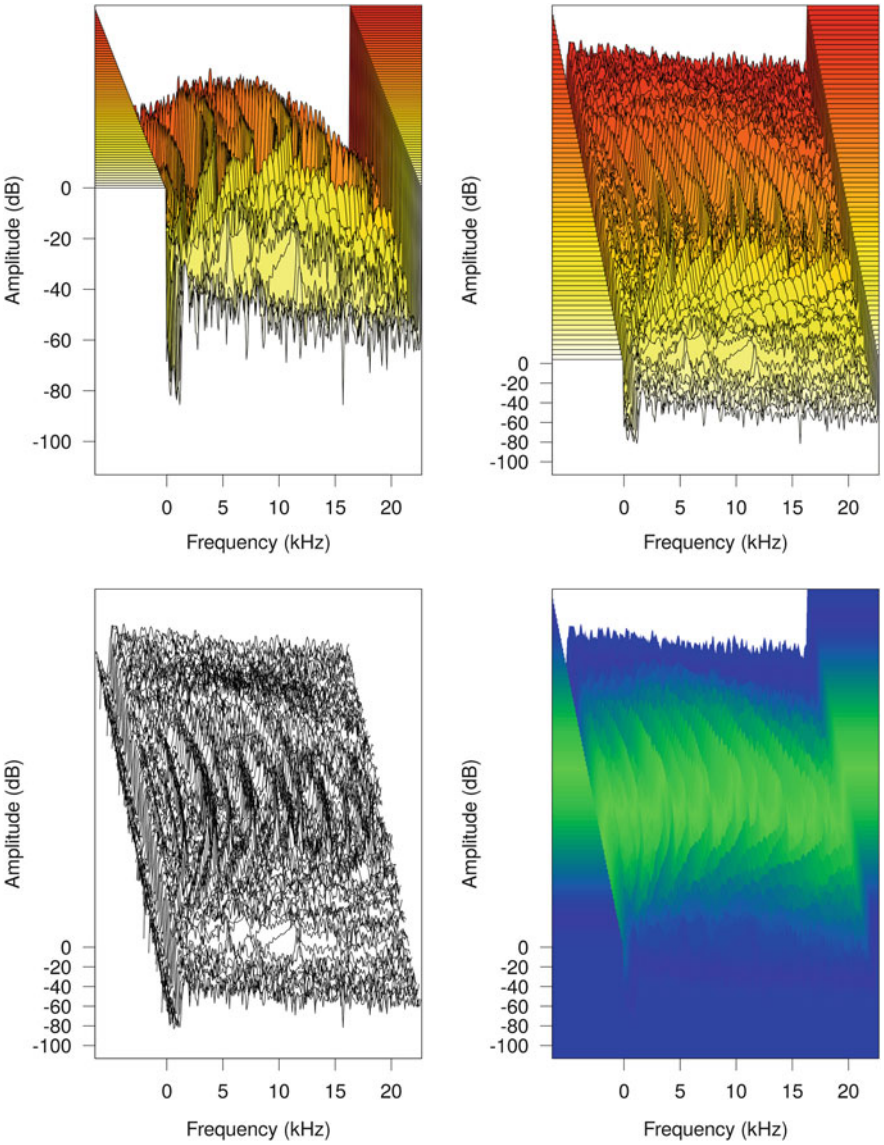


Fig. 11.34 Waterfall display. The figure shows four examples of waterfall display obtained by applying the function `wf()` on `cockroach`. STDFT parameters: Hanning window, 512 samples, 50% of overlap, no zero-padding

```
jet.colors.2 <- colorRampPalette(c("blue", "green", "blue"))
wf(cockroach, wl=wl, ovlp=ovlp,
    hoff=hoff, voff=voff,
    col=jet.colors.2, border = NA)
```

The waterfall display is not commonly used because as in any perspective plot, the information placed in the rear of the plot, here the first frequency spectra, can be hidden by the information in the front of the plot, here the last frequency spectra. However, this plot can still be used as a nice illustration of frequency modulations.

11.13 3D Spectrogram

The last solution to plot a spectrogram is to make use of a true 3D tool that allows the user to turn around the object in all directions. The wonderful package `rgl` opens the possibility to manipulate a 3D visualization device system (Adler and Murdoch 2016). The package is an interface with the external library Open Graphics Library (OpenGL), an application for 2D and 3D graphics rendering that can be found on any operating system. The `seewave` function `spectro3D()` computes the STDFT as `spectro()` and displays it in 3D using the main functionalities of `rgl`. A simple example can be run with:

```
spectro3D(cockroach, ovlp=87.5)
```

`R` opens a 3D device. The user can navigate around, in, and out the spectrogram. The length of the time and frequency axes is under the control of the STDFT parameters. Increasing the length and the overlap of the successive DFTs increases the size of the spectrogram in the 3D space. The axes can be compressed or stretched out using the magnification arguments named `magt`, `magf`, and `maga` for the time, frequency, and amplitude axes, respectively. Colors can also be changed using the argument `col`. The following example increases the amplitude axis by a magnification factor of 4 and chooses the `spectro.colors` palette as a reference color system:

```
spectro3D(cockroach, ovlp=87.5, maga=4, palette=spectro.colors)
```

The interaction provided by the `rgl` 3D device is quite fancy but is not that useful when writing a report or a paper. By chance, the package `rgl` has a function, `rgl.snapshot()`, that can save the current view into a `.png` file. By changing regularly the perspective, or point of view, we can imagine quickly a way to produce

a kind of movie by taking a snapshot at each change of point of view. Setting the perspective is achieved thanks to the function `rgl.viewpoint()`. This function waits the coordinates of the object in reference to a spherical coordinate system. The coordinates are the radial distance ρ from the origin (argument `zoom`), the polar angle θ (argument `theta`), and azimuthal angle ϕ (argument `phi`). The following script is an example of animation around the spectrogram of `cockroach`. The spectrogram is first displayed with `spectro3D()`. The successive coordinates (ρ, θ, ϕ) are generated in three vectors that are then used in a `for` loop. This loop calls the function `rgl.viewpoint()`, generates an appropriate file name in the format `'cockroach-3D-XX'` where `XX` is the number of the perspective, and saves the image with `rgl.snapshot()`:

```
# angle parameters
n <- 100
theta <- seq(-90, 270, length.out=n)
phi <- theta/4.5 + 30
zoom <- seq(0.2, 1.5, length.out=n)
# open rgl device
spectro3D(cockroach, ovlp=87.5, maga=4, palette=spectro.colors)
# animation
for (i in 1:n){
rgl.viewpoint(theta[i], phi[i], zoom=zoom[i])
filename <- paste("image/animation/cockroach-3D-",
                  formatC(i, digits=2, flag="0"),".png",sep="")
rgl.snapshot(filename)
}
```

The script generates a total of 100 images numbered from 001 to 100 that can be grouped in a numbered series to produce an animated image as shown in Fig. 11.35.

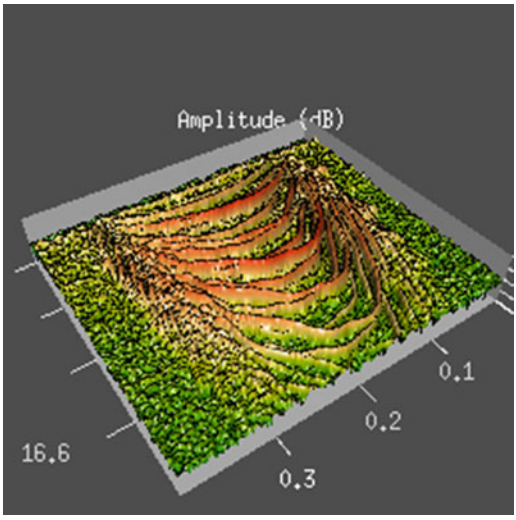


Fig. 11.35 3D animation of the `cockroach` spectrogram. Animation around the 3D spectrogram of `cockroach` based on a series of 100 `.png` images. Animated on electronic version only

Eventually, because a spectrogram is fundamentally a 3D object, it can be transferred from the digital to the real world through a 3D printer. The recipe for real 3D printing is given in the DIY box 11.2.

DIY 11.2 — How to print in 3D a spectrogram

The spectrogram is a three-dimensional object usually projected on a two-dimensional plan. However, 3D printers open the possibility to use the output of `spectro()` as a 3D model. A 3D printer requires the model to be of the format `.stl` [stereolithography]. The trick to print a spectrogram as a three-dimensional object is to convert the amplitude data saved as a numeric matrix into a `.stl` files. To do so, we need first to use the package `r2stl`:

```
library(r2stl)
```

Then it is necessary to get the data of a spectrogram and to convert the negative dB values into positive values:

```
z <- spectro(tico, plot=FALSE)$amp
z <- z + abs(min(z))
```

Eventually, the function `r2stl()` ensures the export to a `.stl` file:

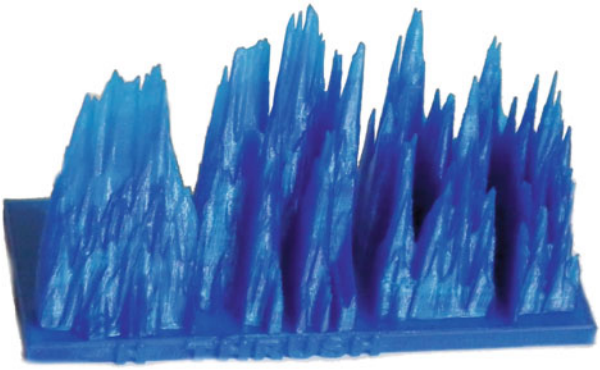
```
r2stl(x=1:nrow(z), y=1:ncol(z), z=z,
      filename="tico.stl")
```

The image below shows an example of a 3D printing produced by David Cartmell. The object is based on a spectrogram of a phrase of the hermit thrush (*Catharus guttatus*) from North America. It represents a segment of sound frozen in plastic allowing one to feel, scrutinize, rotate, and handle in what ever manner a piece of sound. The dimensions of

(continued)

DIY 11.2 (continued)

the model are: X length = 12.7 cm (1.46 s), Y width = 5.1 cm (2.70–6.00 kHz), Z height = 7.6 cm (61–96 dB amplitude):



Object design and picture reproduced with the kind permission of David Cartmell.

11.14 Mean Spectrum

The mean frequency spectrum, or average frequency spectrum, is obtained by computing the STDFT matrix and then the mean of the matrix rows. If we refer to the mathematical writing of the STDFT (see Sect. 11.1.1), then the mean frequency spectrum can be written as:

$$mean.spectrum = \frac{1}{J}\sum_{j=1}^{J} a_{kj}$$

where J is the number of Fourier windows computed along the signal, that is, the number of columns of STDFT matrix.

In terms of `R` code, the function `meanspec()`, which computes the mean frequency spectrum, can be seen as a combination of both `spectro()` and `spec()`. As such, the arguments used in `meanspec()` can be found in the two other functions. The arguments related to `spectro()` are `wl` for the window length, `ovlp` for the window overlap, and `wn` for the window function (see Sect. 11.7 for details regarding these arguments). The arguments in link with `spec()` are `norm`, `PSD`, `PMF`, `dB`, `dBref`, `from`, `to`, `identify`, and several additional graphical parameters (see Sect. 10.1.2 for details regarding these arguments). The average

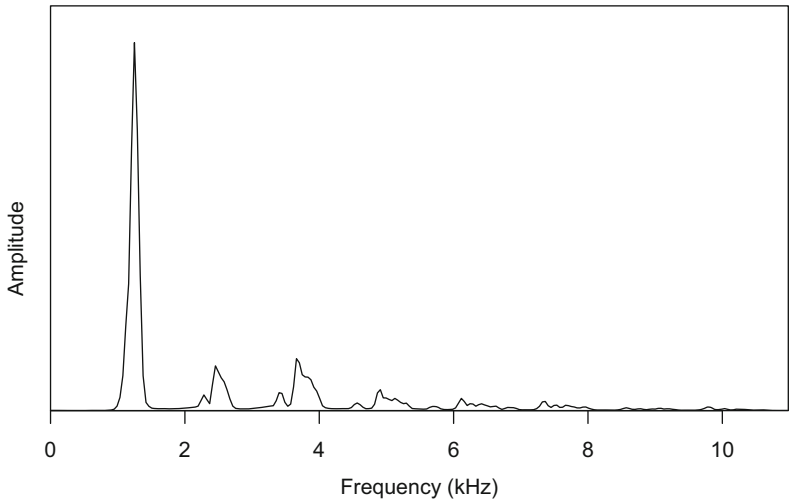


Fig. 11.36 Mean frequency spectrum with `meanspec()`.The plot shows the mean frequency spectrum of `peewit`, a sound with few frequency modulations. STFT parameters: Hanning window, 512 samples, 87.5% of overlap, no zero-padding

spectrum of `peewit` with a 512 sample window (default) and 87.5% overlap can be obtained with (Fig. 11.36):

```
res <- meanspec(peewit, ovlp=87.5)
```

The values can be explored in the object `res`:

```
dim(res)
[1] 256   2
head(res)
              x            y
[1,] 0.00000000 0.0004504876
[2,] 0.04306641 0.0004752211
[3,] 0.08613281 0.0004834852
[4,] 0.12919922 0.0003740508
[5,] 0.17226563 0.0004068626
[6,] 0.21533203 0.0003694860
```

`res` is a two-column matrix, the first column `x` corresponds to the frequency values and the second `y` refers to the amplitude, such that the frequency spectrum

could be visually reconstructed with:

```
plot(res, type="l", xlab="Frequency (kHz)", ylab="Amplitude")
```

The mean frequency spectrum is supposed to summarize the frequency content of a sound over a certain duration. However, the results should be inspected and treated carefully because frequency modulations and irregular temporal patterns may return unexpected profiles.

Audio samples including sounds with frequency modulations can produce meaningless flat frequency spectra (Fig. 11.37, top-left and top-right).

When an audio sample is composed of several sounds that markedly differ in their frequency and duration features, the mean spectrum can give more weight to long sounds than to brief ones. If we have, for instance, a continuous sound produced at a specific frequency and that at the same time we have a brief sound produced only once at another frequency, then the mean frequency spectrum will show a significantly higher frequency peak corresponding to the long pure tone. This discrepancy between the two tones can stand even if the short tone is produced with a higher energy than the long one (Fig. 11.37, bottom-left). This phenomenon can be easily explained by the averaging process: each row of the STDFT matrix is summed and divided by the same number, that is, by the number of the STDFT columns. If a lot of energy is found at a particular frequency, then the sum of the corresponding STDFT matrix row, and so its mean, will be high. The mean spectrum should be then carefully interpreted. For instance, the mean spectrum of `cockroach` does not provide relevant information (Fig. 11.37, bottom-right).

11.15 Soundscape Spectrum

A particular case of a frequency spectrum is the spectrum computed for soundscape ecology studies (see Chap. 16). Following Kasten et al. (2012), the `seewave` function `soundscapespec()` computes the soundscape frequency spectrum applying Welch method (Welch 1967): (1) computation of the STDFT matrix; (2) calculation of the square of the STDFT matrix; (3) computation of the sum of the matrix rows, that is, $\sum_{j=1}^{J} a_{kj}$ where J is the number of DFTs; (4) normalization of the resulting vector with a division by $J \times f_s$; and (v) multiplying the vector by 2. This Welch's frequency spectrum is then binned (discretized) into 1 kHz frequency bands.

Parameters used in Kasten et al. (2012) were a Hamming window of 1024 samples with 50% of overlap and are used as default values (`wl=512`, `ovlp=50`, `wn="hamming"`). A test on the data `forest()` is run hereafter (Fig. 11.38). The value of the function is a two-dimensional matrix with kHz frequency given in the

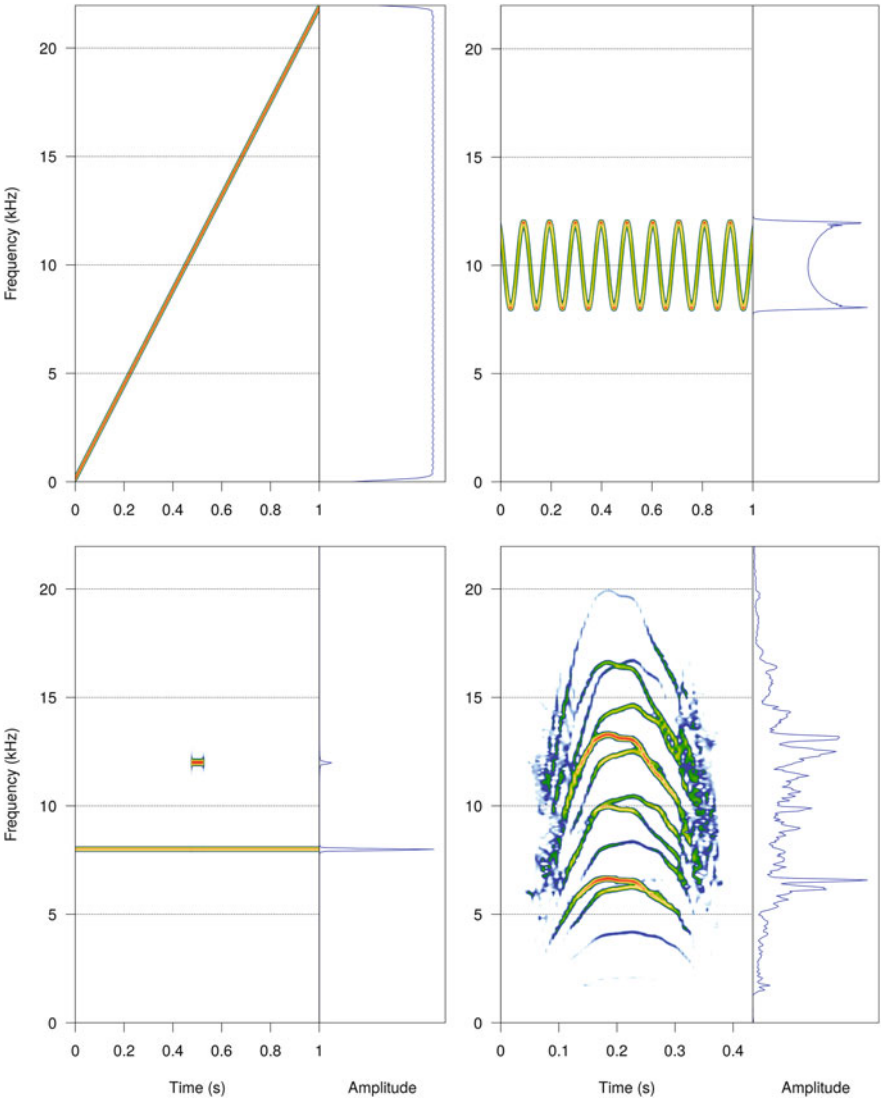


Fig. 11.37 Issues with the mean spectrum. The mean spectrum can returned counterintuitive results as illustrated with three synthetic samples (top-left, top-right, bottom-left) and the natural `cockroach` whistle (bottom-right). For each case the spectrogram is shown on the left and the mean spectrum on the right

first column and the power according to Welch's definition in the second column (Fig. 11.38).

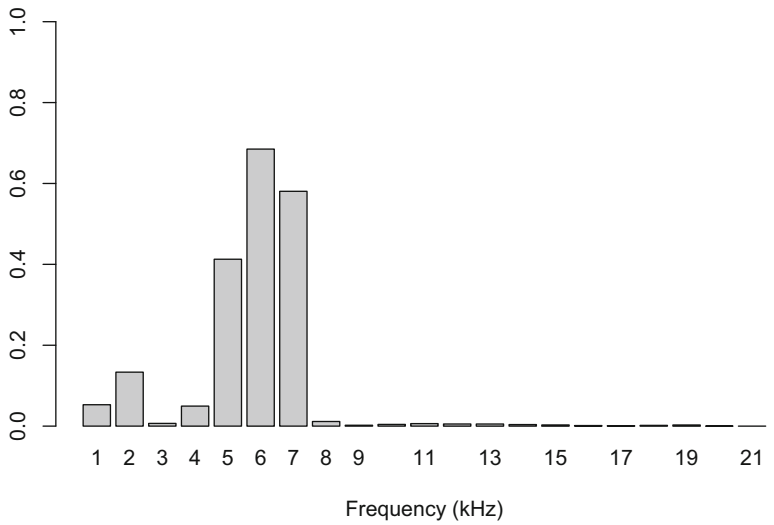


Fig. 11.38 Soundscape frequency spectrum. The soundscape frequency spectrum, here computed and displayed for the recording `forest` consists in a Welch frequency spectrum binned into 1 kHz frequency bands. The graphic is based on the high-level plot graphic function `barplot()`

```
res <- soundscapespec(forest)
```

Here is the content and structure of the object returned by `soundspec()`:

```
dim(res)
[1] 21  2
head(res)
     frequency  amplitude
[1,]         1 0.05286976
[2,]         2 0.13343057
[3,]         3 0.00672790
[4,]         4 0.04949914
[5,]         5 0.41266958
[6,]         6 0.68500654
```

Chapter 12
Mel-Frequency Cepstral and Linear Predictive Coefficients

Chapter 11 was an exploration of time-frequency analysis based on the frequency spectrum obtained with the Fourier transform. This chapter introduces two other features of time-frequency variations: the Mel-frequency cepstral coefficients (MFCCs) and the linear predictive coefficients (LPC).

12.1 Mel-Frequency Cepstral Coefficients (MFCCs)

Sections 9.8 and 10.2 introduced the theory and the practice of the quefrency cepstrum, a curious derivation of the frequency spectrum that can provide useful information about echoes or fundamental frequency. Here we will discover the mel-frequency cepstral coefficients, abbreviated MFCCs, which are used in automatic recognition processes, mostly in speech analysis but also occasionally in music, bioacoustics, and ecoacoustics.

12.1.1 Theory

Mel-frequency cepstral coefficients were developed in the context of word recognition in spoken language (Davis and Mermelstein 1980). The central idea is to compress speech data by keeping only relevant information for the detection of phonetic differences. The principle refers to human audition by using the logarithmic mel(ody) scale which definition is based on how the human ear perceives frequency and loudness (see Sect. 9.4.1).

MFCCs are literally defined as "the result of a cosine transform of the real logarithm of the short-term energy spectrum expressed on a mel-frequency scale"

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_12

(Davis and Mermelstein 1980). This definition hides several computing steps that are enumerated hereafter:

1. Input: MFFCs can be computed for any sound. Nonetheless, these coefficients have been developed and tuned to process an automatic classification of speech signals.
2. Preemphasis filter: a high-pass frequency filter is applied to the signal in order to reduce the importance of low-frequency components and increase the importance of high-frequency components. This frequency filter, which is adapted to speech, is parametrized by a time constant α as detailed in Sect. 14.1.
3. Short-time Fourier discrete transform (STDFT) computation: the signal is divided into successive windows of duration σ_t, and the FFT is computed for each window (see Sect. 11.1). For each frame, a tapering window is previously applied, usually a Hamming window (see Sect. 9.6).
4. Mel conversion of the STDFT matrix: the frequency scale of the STDFT matrix is converted from Hz to mel. The mel scale is a subjective and logarithmic frequency scale related to human audition (see Sect. 9.4.1).
5. Generation of a bank of mel-frequency filters: this step does not consist in any filtering process but in preparing the mel-frequency response of a series, or bank, of filters. The filters can cover the full bandwidth of the spectrum, that is, from 0 to $f_s/2$, or they can expand over a selected bandwidth delimited by a pair of lower and upper cutoff frequencies, denoted f_l and f_u, respectively. Each filter has a typical triangular shape. The central frequency, that is, the frequency at the top of the triangle, is obtained with (Sharan and Moir 2016):

$$f_{cm} = f_l + \frac{k(f_u - f_l)}{m+1}$$

In the following, the central frequencies of a bank of 40 mel-frequency filters expanding from $f_l = 300\,\text{Hz}$ to $f_u = 8000\,\text{Hz}$ are manually computed:

```
fl <- mel(300)                    # lower frequency in mel
fu <- mel(8000)                   # upper frequency in mel
m <- 40                           # number of filters
k <- 1:m                          # i^th filter
fcm <- fl + k*(fu-fl)/(m+1)       # central frequency in mel
fcm
 [1]  461.4420  520.9075  580.3731  639.8386  699.3042
 [6]  758.7697  818.2353  877.7008  937.1664  996.6319
[11] 1056.0975 1115.5630 1175.0286 1234.4942 1293.9597
[16] 1353.4253 1412.8908 1472.3564 1531.8219 1591.2875
[21] 1650.7530 1710.2186 1769.6841 1829.1497 1888.6153
[26] 1948.0808 2007.5464 2067.0119 2126.4775 2185.9430
[31] 2245.4086 2304.8741 2364.3397 2423.8052 2483.2708
[36] 2542.7364 2602.2019 2661.6675 2721.1330 2780.5986
fcm.hz <- mel(fcm, inverse=TRUE) # central frequency in Hz
```

(continued)

```
fcm.hz
 [1]  354.1808  411.2972  471.5081  534.9814  601.8937
 [6]  672.4313  746.7908  825.1790  907.8145  994.9272
[11] 1086.7597 1183.5677 1285.6210 1393.2035 1506.6149
[16] 1626.1711 1752.2049 1885.0674 2025.1284 2172.7781
[21] 2328.4275 2492.5101 2665.4829 2847.8274 3040.0516
[26] 3242.6906 3456.3087 3681.5009 3918.8941 4169.1495
[31] 4432.9640 4711.0721 5004.2483 5313.3091 5639.1150
[36] 5982.5733 6344.6405 6726.3248 7128.6891 7552.8537
```

The successive filters are arranged such that they overlap by a factor of 50%. The function `melfilterbank()` of `seewave` can be used to generate and visualize these filters according to five main arguments: the sampling frequency f_s (`f`), the Fourier window length σ_t (`wl`), the lower frequency of the filter bank f_l (`minfreq`), the upper frequency of the filter bank f_u (`maxfreq`), and the total number of filters m (`m`). The following use of `melfilterbank()` generates a bank of 26 filters starting at 0 Hz and ending at half the sampling frequency $f_s = 44{,}100 \div 2 = 22{,}050$ Hz. The filters, actually their frequency response, can be displayed on a single graph with a palette of colors determined with the graphical argument `palette` (Fig. 12.1):

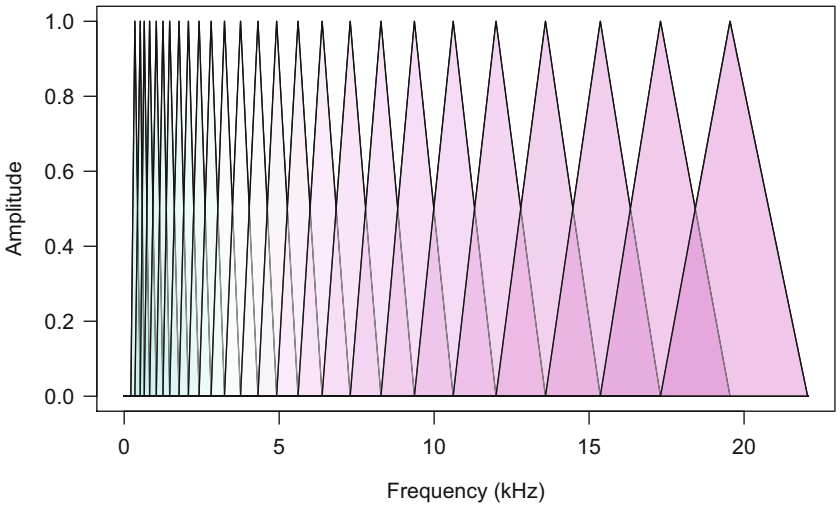


Fig. 12.1 Mel-frequency filter bank. A bank of mel-frequency triangular filters is generated and displayed with the `seewave` function `melfilterbank()`. The bank includes 26 filters starting from 0.3 to 22.05 kHz

```
res <- melfilterbank(f=44100, wl=1024, minfreq=300, m=26,
                     palette=cm.colors, plot=TRUE)
```

The object returned is a list containing a matrix with all filters (`$amp`) and a numeric vector with the central frequencies of the filters expressed in kHz (`$central.freq`):

```
head(res$central.freq)
[1] 0.4226867 0.5604254 0.7150628 0.8886721 1.0835811
[6] 1.3024027
tail(res$central.freq)
[1] 10.66135 12.05524 13.62014 15.37703 17.34946 19.56389
```

6. Application of the mel-frequency filters to each STDFT window: each mel-frequency spectrum obtained for each time window of the STDFT is squared and multiplied by each mel-frequency filter. If 26 filters were used, then 26 multiplications are processed for each window, leading to a series of 26 vectors for each window.
7. Estimation of the energy for each mel-frequency filtered spectrum: the values of each vector obtained after the filtering process are summed up. This reduces considerably the dimension of the data. If the STDFT is computed with a 512 sample Fourier window, then each FFT has a length of 256 samples. The filtering increases the dimension by returning 26 vectors of length 256, but the sum summarizes the 256 values of each vector into a single value and hence reduces the dimension to 26 filter bank energies only.
8. Logarithmic transformation: the filter bank energy values are log transformed.
9. Discrete cosine transform: a type III discrete cosine transform (DCT) is applied on the log filter energies. The DCT is a kind of Fourier transform but using only real coefficients when the discrete Fourier transform (DFT) returns complex coefficients. There are different types of DCTs, the type II is the most common type, and the type III corresponds to the inverse of the type II, also named the inverse discrete cosine transform (IDCT). Applying the type III DCT at this stage of the MFCC process is therefore similar to use of the inverse Fourier transform on the logarithm of the Fourier transform when computing the cepstrum (see Sect. 9.8). Having transformed and filtered the data in reference to the mel scale, the DCT operates a travel from the mel-frequency domain to the mel-quefrency domain.
10. Selection of relevant MFCCs: a second reduction of data dimension is operated by discarding the first coefficient and keeping the next n coefficients, with usually $n = 13$. This selection can be done because the DCT compressed the relevant information on the speech properties in the first coefficients.

11. Application of a lifter on MFCCs: a lifter is a kind of filter applied to cepstral coefficients (see Table 9.2). A lifter can be seen as a weighting function that gives more importance to midrange coefficients, here to midrange MFCCs.

To summarize, the MFCCs are obtained at the end of the following transform chain:

$$\begin{aligned}&\text{input signal} \rightarrow \text{preemphasis} \rightarrow \text{STDFT} \rightarrow \text{mel scale} \rightarrow \text{mel filtering}\\ &\quad\rightarrow \text{compression by sum} \rightarrow \log \rightarrow \text{DCT} \rightarrow \text{selection} \rightarrow \text{liftering}\end{aligned}$$

The resulting data, or features, are the mel-frequency cepstral coefficients, or MFCCs. This abbreviation summarizes the different operations described above (frequency spectrum, mel conversion and filtering, cepstral transform through the DCT). The operation consists, among others, in an important data compression: the process starts with a time signal lasting d s sampled at f_s Hz, travels to a time $\times$ frequency matrix of dimension $(\sigma_t \div 2, d \times f_s \div \sigma_t)$ due to a STDFT computed with a non-overlapping window lasting σ_t, goes through a time $\times$ quefrency filter bank energy matrix of dimension $(26, d \times f_s \div \sigma_t)$, and arrives at a MFCC matrix of dimension $(13, d \times f_s \div \sigma_t)$.

In some cases, the first and second time derivatives of the MFCCs, that is, the speed and acceleration of MFCC changes over time, are used as additional sound features in particular for automatic speech classification. The speed of MFCCs is known as the delta coefficients and the acceleration as delta-delta coefficients. If the number of MFCCs was 13, then the number of features increases to 26 if delta coefficients are included (13 MFCCs + 13 delta coefficients) and 39 if delta and delta-delta coefficients are included (13 MFCCs + 13 delta coefficients + 13 delta-delta coefficients).

12.1.2 Practice

The function `melfcc()` from the package `tuneR` is the only `R` solution to compute MFCCs. `melfcc()` can be seen as master function that calls several other functions to proceed the different steps of the MFCC computation. The preemphasis filter is achieved with the function `filter()` of the package `signal` (see Sect. 14.1); the STDFT is computed with `powspec()` (see Sect. 11.4); the mel conversion and mel-frequency filtering are applied with `audspec()`; the logarithm conversion, the DCT, and the selection of the coefficients refer to `spec2cep()`; and the liftering calls `lifter()`. The functions `audspec()`, `spec2cep()`, and `lifter()` are introduced in the next two sections before to get into the details of `melfcc()`.

The file `hello.wav`, including a 48,000 Hz recording of the English word "hello" pronounced by a 7-year-old French native girl, will be used to test these functions:

```
hello <- readWave("sample/hello.wav")
hello

Wave Object
Number of Samples:      38400
Duration (seconds):     0.8
Samplingrate (Hertz):   48000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

12.1.2.1 Mel-Frequency Conversion and Filtering

The `tuneR` function `audspec()` does the conversion from Hertz to mel and proceeds the mel-frequency filtering in reference to a mel-frequency filter bank leading to a reduction of dimensionality. The result can be named an auditory spectrogram. The function waits a STDFT (spectrogram) matrix as input, obtained for instance with `powspec()`, and the sampling frequency f_s of the original signal. It is necessary to specify the properties of the mel-frequency filter bank by informing its lower and upper frequency limits (arguments `minfreq` usually set to 0, `maxfreq` usually set to $f_s \div 2$), the total number of mel-frequency filters (argument `nfilts` usually set to 26), and the scale used (argument `fbtype` commonly set to `"htkmel"` for standard mel scale):

```
## spectrogram matrix
f <- hello@samp.rate
wl <- 512
p.hello <- powspec(hello@left, sr=f,
                   wintime=wl/f, steptime=wl/f)
## mel conversion and filtering
a.hello <- audspec(p.hello, sr=f,
                   minfreq=0, maxfreq=f/2,
                   nfilts=26, fbtype="htkmel")
```

The result is stored in the list item `$aspectrum` as a (frequency, time) matrix. There are here 26 rows corresponding to the number of filters and 74 columns corresponding to a STDFT with a 512 sample Fourier window:

```
str(a.hello$aspectrum)
 num [1:26, 1:74] 2351014 12989 153933 213479 38594 ...
```

`audspec()` is not a graphical function and has no plot method associated to. The following code offers a solution to display the auditory spectrogram saved in the matrix `a.hello` by calling the function `image()` (Fig. 12.2):

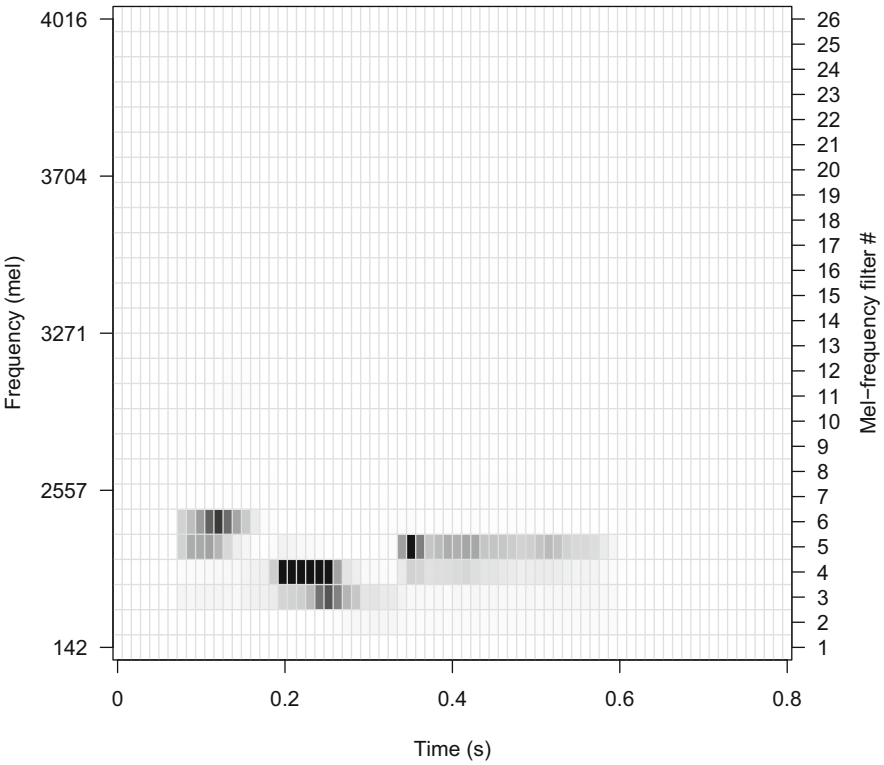


Fig. 12.2 Auditory spectrum. The result of the function `audspec()` is displayed with the function `image()`. The left *y*-axis refers to frequency expressed in mel and the right *y*-axis indicates the index of the 26 mel-frequency filters used. Time was divided into 74 windows by the STDFT

```
# time scale
at <- seq(0, 1, length=5)
time <- round(seq(0, duration(hello), length=5), 1)
# Hz frequency scale
hz <- round(seq(f/512, f/2, length=5))
# mel frequency scale
mel <- round(hz2mel(hz, htk=TRUE))
# plot
par(mar=c(5.1, 4.1, 4.1, 4.1), las=1)
col <- gray((512:0)/512)
image(t(a.hello$aspectrum), col=col,
      axes=FALSE, xlab="Time (s)", ylab="Frequency (mel)")
axis(side=1, at=at, labels=time)
axis(side=2, at=at, labels=mel)
axis(side=4, at=0:25/25, labels=1:26,)
mtext("Mel-frequency filter #", side=4, las=0, line=2.5)
abline(h=(0:25/25)+1/(25*2), col="lightgray")
abline(v=(0:73/73)+1/(73*2), col="lightgray")
box()
```

12.1.2.2 Cepstral Coefficients

The `tuneR` function `spec2cep()` does the logarithm transform, the DCT computation, and the selection of n preferred coefficients. `spec2cep()` needs to be fed with a (mel-frequency, time) matrix returned by `audspec()`. The number of coefficients kept is given by the argument `ncep` and the DCT type by the argument `type`. To proceed a classical MFCC computation, we set up the function as:

```
cep.hello <- spec2cep(a.hello$aspectrum, ncep=13, type="t3")
```

The object returned by `spec2cep()` is a two-item list. The first item `$cep` includes the cepstral coefficients, and the second item `$dctm` gives the DCT matrix used to obtain `$cep`. The most important item, which will be used in the next steps, is therefore `$cep`:

```
str(cep.hello)
List of 2
 $ cep : num [1:13, 1:74] 76.94 0.233 3.885 0.457 1.934 ...
 $ dctm: num [1:13, 1:26] 0.277 0.277 0.275 0.273 0.269 ...
```

12.1.2.3 Lifter

A lifter, that is, a filter in the cepstral language, is optionally applied to the MFCCs as an ultimate step to emphasize peculiar coefficients. The `tuneR` function `lifter()` does such changes by applying a $[0, \pi]$ sine function on htk-mel coefficients. The length of the lifter, and therefore its shape, can be controlled with the argument `lift`. Setting `lift=1` does not induce any change when setting `lift=ncep-1`, here `lift=12`, applies a sine $[0, \pi]$ weight function as shown in:

```
lifter.hello <- lifter(cep.hello$cep, lift=13-1, htk=TRUE)
str(lifter.hello)
 num [1:13, 1:74] 76.94 0.596 15.541 2.397 11.98 ...
```

Figure 12.3 illustrates the shape of seven lifters differing in their length.

12.1.2.4 Complete MFCC Computation

Considering the previous sections, the MFCCs can be obtained by using a series of `tuneR` functions (see DIY box 12.1) or more simply with the function `melfcc()`.

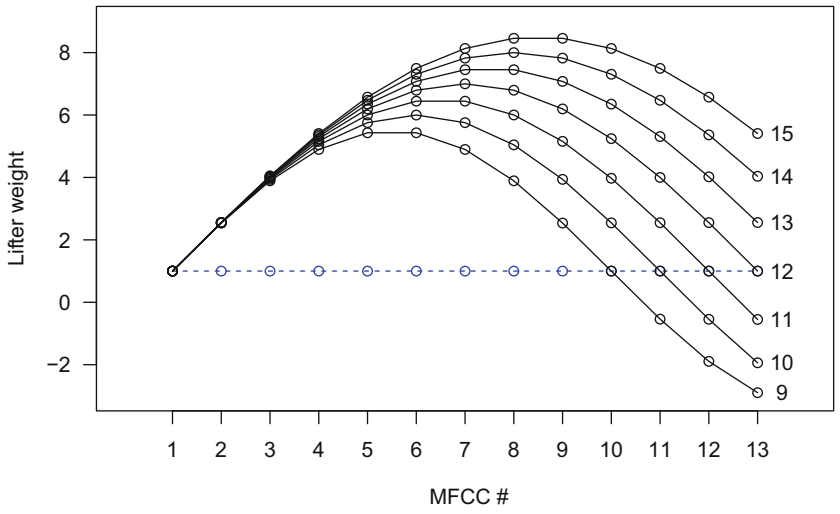


Fig. 12.3 Lifters on 13 MFCCs that are all equal to 1. The blue and dashed line displays the 13 MFCCs. The plain black lines show the weighting function of seven lifters differing in their length, from 9 to 15. The lifter of length 12, that is, the number of MFCCs-1, applies a perfect sine function between 0 and π

DIY 12.1 — How to obtain MFCCs step by step

The MFCCs can be obtained by hand following a five-step process involving `preemphasis()` (replacing here the function `filter()` of the package `signal` for a sake of simplification), `powspec()`, `audspec()`, `spec2cep()`, and `lifter()`:

```
f <- hello@samp.rate  # sampling frequency
wl <- 512             # STDFT window size
ncep <- 13            # final number of MFCCs
f.hello <- preemphasis(hello, alpha=0.97, output="Wave")
p.hello <- powspec(f.hello@left, sr=f,
                   wintime=wl/f, steptime=wl/f)
a.hello <- audspec(p.hello, sr=f,
                   nfilts=ncep*2, fbtype="htkmel")
cep.hello <- spec2cep(a.hello$aspectrum,
                      ncep=ncep, type="t3")$cep
mfccs <- lifter(cep.hello, lift=ncep-1, htk=TRUE)
str(mfccs)
 num [1:13, 1:74] 62.51 -30.92 8.57 -5.56 8.7 ...
```

The function `melfcc()` contains a long list of arguments that can be related to the successive steps of the MFCC computation described in Sect. 12.1.1:

1. Input: the argument `samples` waits a `Wave` object, and the argument `sr` can be used to specify the sampling rate or sampling frequency f_s. This latter argument is fed by default by the sampling frequency of the `Wave` object.
2. Preemphasis filter: the time constant α is set with the argument `preemph`. The default value is the standard value 0.97, and choosing a value of 0 removes the filter.
3. Short-time Fourier transform (STDFT) computation: the duration in s of the FFT window σ_t has to be provided with the argument `wintime`, and the hop between adjacent windows in s should be specified with the argument `hoptime`. These two arguments correspond to the arguments `wl` and `ovlp` of `spectro()` but differ from them in the time unit used (second *vs* samples).
4. Mel conversion of the STDFT matrix: different logarithmic scales can be selected with the argument `fbtype`, in particular the mel and the HTK-mel scales. A Bark scale can also be used (see Sect. 9.4.1 for details on these scales).
5. Generation of a bank of mel-frequency filters: we have seen that the definition of the limits of the mel-frequency depends on a lower frequency, an upper frequency, and the total number of filters. These parameters can be set up with the arguments `minfreq`, `maxfreq`, and `nbands`, respectively.
6. Application of the mel-frequency filters to each STDFT window: this process is based on squaring the STDFT results. This square operation is actually applied by default by `melfcc()` which works directly on the spectrogram, the spectrogram being the square of the STDFT as defined in Sect. 11.2. This can be canceled by turning `sumpower` to `FALSE`.

7. Logarithmic transformation: there is no specific argument for this step.
8. Discrete cosine transform: the type of the DCT, I, II, III, or IV, can be chosen here. For a usual MFCC computation, the DCT should be set to type III with `dcttype="t3."`
9. Selection of relevant MFCCs: the number of cepstral coefficients to keep can be fixed with the argument `numcep`.
10. Application of a lifter on MFCCs: the lifter is controlled with the argument `htklifter` that should be set to `TRUE` for HTK-mel and with the argument `lifterexp` that matches with the argument `lift` of `lifter()` and that should be then fed with an integer number, usually the number of MFCCs-1.

By default, the value of `melfcc()` is the matrix of the MFCCs with each column corresponding to a time window of the original signal. The number of rows equals to the number of MFFCs kept, that is, the value given to the argument `numcep`. If the argument `frames_in_rows` is `TRUE`, then the matrix is transposed with time windows in rows.

If the argument `spec_out` is `TRUE`, `melfcc()` returns a list of four items:

1. `$cepstra` containing the matrix of the MFCCs as just described above,
2. `$aspectrum` the auditory spectrogram of the signal as obtained with `audspec()`,
3. `$pspectrum` the spectrogram of the signal, as obtained with `powspec()`,
4. `$lpcas` the linear predictive coefficients if the argument `modelorder > 0` (see Sect. 12.2 for details regarding the LPCs).

We here compute the MFCCs based on a STDFT with a sliding window made of 512 samples expressed here in s (argument `wintime`) and no overlap between adjacent windows meaning a hop parameter similar to the window length of `spectro()` (argument `hoptime`). The number of mel-frequency filters is set to 26 (argument `nbands`), and the number of MFCCs kept at the end of the process is 13 (argument `numcep`). The most common mel scale is used with `fbtype="htkmel"`, and the type III DCT transform is selected with `dcttype="t3."` Complete results are saved in an object `res` by the call of `spec_out=TRUE`, and the matrices are transposed with `frames_in_rows=FALSE` to have a classical organization of time windows in columns. The default value `sumpower=TRUE`, `preemph=0.97`, `minfreq=0`, and `maxfreq=sr/2` are not modified:

```
wl <- 512
ncep <- 13
mfcc.hello <- melfcc(hello, sr=f,
                wintime=wl/f, hoptime=wl/f,
                numcep=ncep, nbands=ncep*2,
                fbtype="htkmel", dcttype="t3",
                htklifter=TRUE, lifterexp=ncep-1,
```

(continued)

```
                frames_in_rows=FALSE,
                spec_out=TRUE)
```

We can now explore the data, in particular their dimensions:

```
str(mfcc.hello)
List of 4
 $ cepstra  : num [1:13, 1:74] 62.5 -30.94 8.53 -5.61 8.65 ...
 $ aspectrum: num [1:26, 1:74] 2197.4 26.1 500 856.7 334.8 ...
 $ pspectrum: num [1:256, 1:74] 5658.2 2303.3 11.4 38 519.7 ...
 $ lpcas    : NULL
```

The STDFT (spectrogram) is a matrix with $512 \div 2 = 256$ lines and 74 columns corresponding to 74 time windows (item `res$psectrum`). The auditory spectrum has the same number of columns as the time resolution is the same but has a reduced number of rows corresponding to the 26 mel-frequency filters (item `res$aspectrum`). The MFCCs are stored in a matrix with a similar number of columns as the time resolution is still unchanged but with a reduced number of 13 rows corresponding to the number of coefficients kept (item `res$cepstra`).

The MFCCs can be visualized using `image()` as previously tested with the output of `audspec()` (Fig. 12.4):

```
## time scale
at <- seq(0, 1, length=5)
time <- round(seq(0, duration(hello), length=5), 1)
## plot
col <- gray((512:0)/512)
par(las=1)
image(t(mfcc.hello$cepstra), col=col,
      axes=FALSE, xlab="Time (s)", ylab="MFCC #")
axis(side=1, at=at, labels=time)
axis(side=2, at=0:12/12, labels=1:13,)
abline(h=(0:12/12)+1/(12*2), col="lightgray")
abline(v=(0:73/73)+1/(73*2), col="lightgray")
box()
```

The delta coefficients can be obtained with the `tuneR` function `deltas()`; the use is rather simple:

```
d <- deltas(mfcc.hello$cepstra)
str(d)
 num [1:13, 1:74] -1.77 -2.66 32.28 36.17 47.36 ...
```

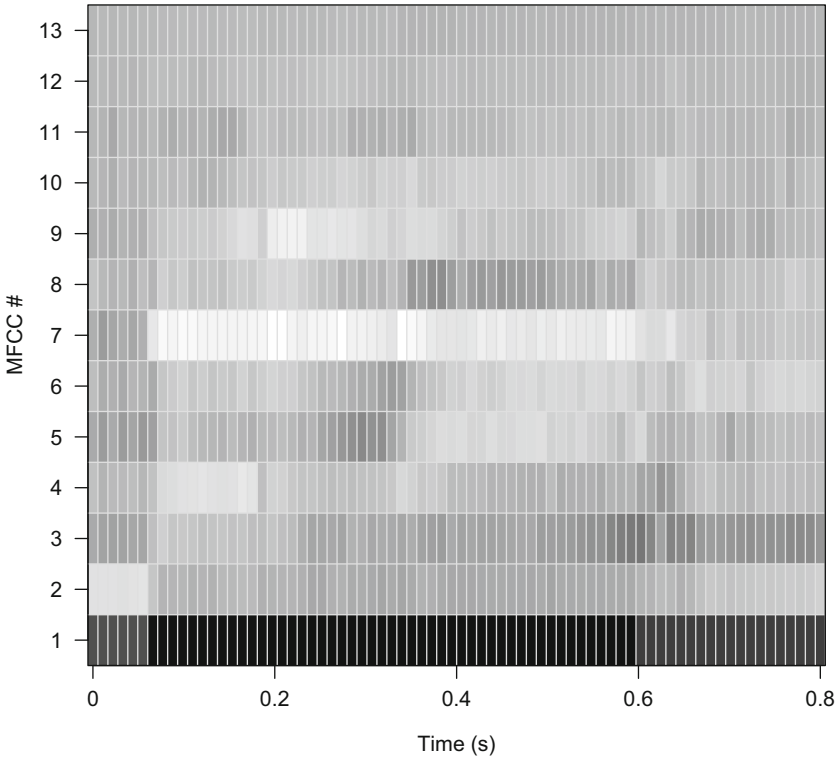


Fig. 12.4 Display of the MFCCs. The 13 MFCCs selected are displayed according to time that was divided into 74 windows by the STDFT

An so the delta-delta coefficients are obtained with:

```
dd <- deltas(d)
str(dd)
 num [1:13, 1:74] 3648 1857 -1617 -1255 -943 ...
```

which is of course equivalent to:

```
dd <- deltas(deltas(mfcc.hello$cepstra))
```

12.2 Linear Predictive Coefficients (LPCs)

12.2.1 Theory

Linear predictive coding (LPC), leading to linear predictive coefficients (LPCs), is another technique similarly developed for speech analysis. The stem idea is to modelize speech production as an additive model composed of a source and a filter with one or more resonant frequencies f_r. The source corresponds to the primary vibrations of the vocal folds, and the filter is due to the shape and movement of the vocal tract, that is, of the throat, the tongue, and the lips. LPC technique aims at separating the source from the filter by estimating the transfer function, and therefore the frequency response, of the vocal tract filter (see Chap. 14).

In terms of signal analysis, the LPC is an autoregressive (AR) model (Cryer and Chan 2008). A pth-order AR consists in predicting the current sample $s[n]$ through a linear polynomial expression that includes the p previous samples. If the prediction is noted $\hat{s}[n]$, the AR model equation is written:

$$\begin{aligned}\hat{s}[n] &= a_1 s[n-1] + a_2 a[n-2] + \cdots + a_p s[n-p] \\ &= \sum_{k=1}^{p} a_k s[n-k]\end{aligned}$$

The a_k coefficients are the linear predictive coefficients. These coefficients can be estimated minimizing the difference between the true value $s[n]$ and the predicted value $\hat{s}[n]$, so that we have an error or residual term, $e[n]$, which satisfies:

$$e[n] = s[n] - \hat{s}[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k]$$

The total prediction error E for the complete signal is then the sum of the squared errors for each sample:

$$E = \sum_n e^2[n] = \sum_n \left(s[n] - \sum_{k=1}^{p} a_k s[n-k] \right)^2$$

The a_k coefficients, which can be estimated through an autocorrelation process, are the coefficients of the vocal tract filter transfer function which is defined as:

$$\begin{aligned}H(z) &= \frac{1}{A(z)} \\ &= \frac{1}{1 - \sum_{k=1}^{p} a_k z^{-k}}\end{aligned}$$

with z a complex variable $z = x + iy = r(\cos\theta + i\sin\theta) = re^{i\theta}$, and $i^2 = -1$.

This transfer function is used to get the frequency response of the vocal tract. This can be viewed as a spectral envelope of the frequency spectrum. The frequency response can be used to localize and describe speech formants.

12.2.2 Practice

The LPC method is encoded in the function `lpc()` of the package `phonTools`. The function returns the a_k coefficients linear predictive coefficients of an input sound by (1) applying a preemphasis filter (argument `preemph`) (see Sect. 14.1), (2) removing any potential DC offset, (3) multiplying the sound by a Hanning window, and (4) estimating the coefficients a_k with an autocorrelation method. The number of coefficients, k, can be set with the argument `order`. The function can also go a step further by using the coefficients a_k to obtain the frequency response of the associated filter.

We can test the function on `hello` speech dataset. The LPC is usually applied on a 10–20 ms section of sound where the signal is supposed to be stationary. We then first select 10 ms of `hello`, and we downsample it to 12,000 Hz to keep relevant frequencies for speech:

```
sel.hello <- cutw(hello, from=0.07, to=0.17, output="Wave")
sel.hello <- resamp(sel.hello, g=12000, output="Wave")
```

We then apply the function `lpc()` by seeting the number of LPC coefficients with the argument `order`. The function plots at the same time the result of the transfer function with `show=TRUE` (Fig. 12.5):

```
coeffs <- lpc(sel.hello@left, fs=sel.hello@samp.rate, order=16, show=TRUE)
```

The coefficients are:

```
coeffs
 [1]  1.00000000 -0.85082213  0.98910359 -0.84965549
 [5]  0.60744100 -0.56381546  0.56646631  0.11406063
 [9] -0.07487837  0.44027795 -0.30491669  0.15437669
[13] -0.13023889  0.21344999 -0.13889342 -0.01019266
[17] -0.06004648
```

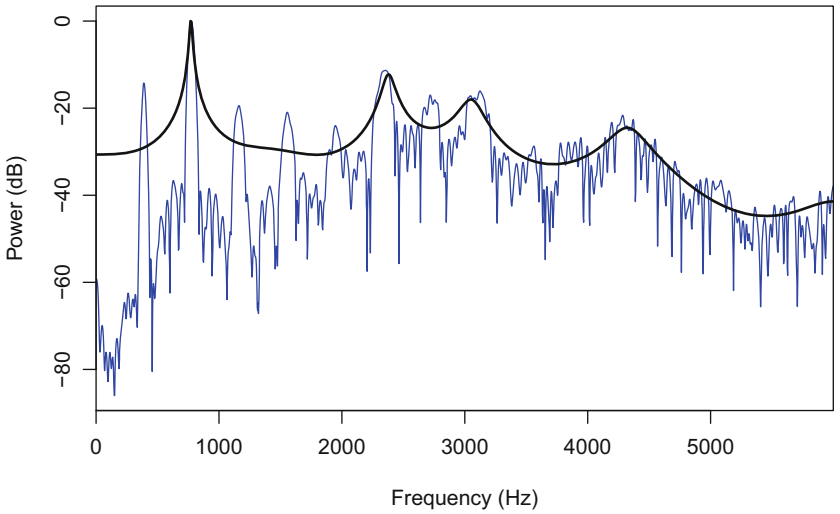


Fig. 12.5 Filter frequency response deriving from LPC. The function `lpc()` returns the LPC coefficients of a sound, here `hello`, and plots the resulting filter frequency response (black line). The original frequency spectrum obtained after a pre-emphasis filter is also shown (blue line)

This code could also be written in two steps by using the function `freqresponse()` on `phonTools` that finds the frequency response of any filter defined by a transfer function $H(z) = b \div a$. Here with $b = 1$ with $a = a_k$, we have:

```
coeffs <- lpc(sel.hello@left, fs=sel.hello@samp.rate, order=16, show=FALSE)
freqresponse(b=1, a=coeffs, fs=sel.hello@samp.rate)
```

The next step is to identify the resonant frequency f_r and $-3\,\text{dB}$ bandwidth $\Delta_{-3\,\text{dB}} f$ of each formant. This estimation is achieved with the function `findformants()` that uses the following expression (Snell and Milinazzo 1993):

$$f_r = \frac{f_s}{2\pi}\theta_0$$

and

$$\Delta_{-3\,\text{dB}} f = -\frac{f_s}{\pi}\log(r_0)$$

with f_s the sampling frequency and $z_0 = r_0 e^{i\theta_0}$ the complex root pairs of $A(z)$, that is, the complex solutions of $A(z) = 0$.

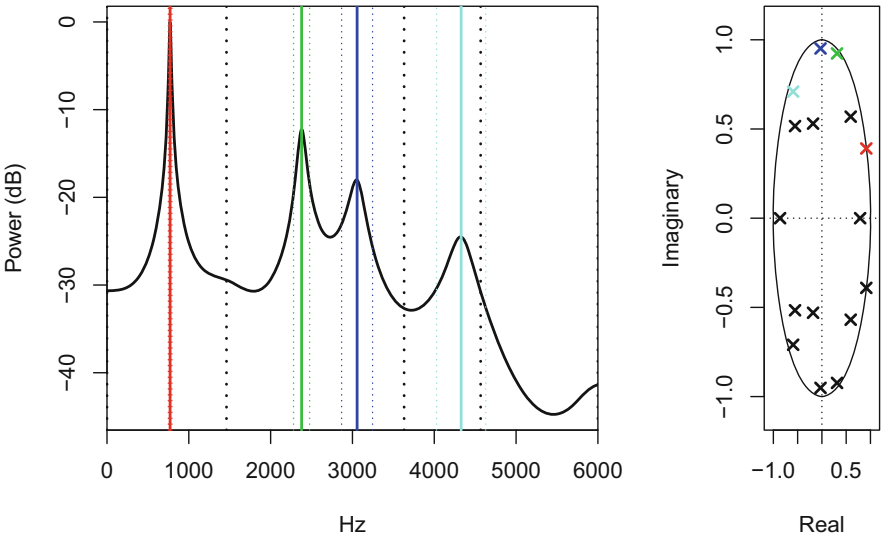


Fig. 12.6 Formant analysis based on LPC. The function `findformants()` can estimate the resonant frequency f_r and -3 dB bandwidth $\Delta_{-3\,\mathrm{dB}} f$ of each formant. A pole-zero diagram (right) completes the spectral display (left) to show the position of the formants in the complex unit circle

The function uses the coefficients a_k as data (Fig. 12.6):

```
formants <- findformants(coeffs=coeffs, fs=sel.hello@samp.rate,
                                    showbws=TRUE)
```

```
formants
  formant bandwidth
1  771.25  21.61550
2 2378.54  98.45518
3 3056.91 189.28495
4 4328.59 300.44634
```

or can accept directly the sound as input. In that case the argument `coeffs` waits the number of LPC coefficients to be used, here 16:

```
formants <- findformants(sound=sel.hello@left, fs=sel.hello@samp.rate,
                                 coeffs=16)
formants
  formant bandwidth
1  771.25  21.61550
2 2378.54  98.45518
3 3056.91 189.28495
4 4328.59 300.44634
```

The results are plotted with the formants highlighted on the filter frequency response and with the positions of the complex roots pairs inside the complex unit circle (Fig. 12.6), a plot known as the pole-zero diagram.

Chapter 13
Frequency and Energy Tracking

With similar efforts developed in Chaps. 11 and 12 to characterize the time-frequency properties of sound, we will here detail ways to estimate frequency changes along time by tracking (1) the dominant frequency, (2) the fundamental frequency, (3) the formants, and (4) the instantaneous frequency. We will also introduce a technique, known as Teager-Kaiser energy operator (TKEO), that can evaluate both instantaneous frequency and instantaneous amplitude.

To illustrate frequency and energy tracking, we will use a new sound that consists into a brief and high-frequency call emitted by a common European bat, *Pipistrellus kuhlii* (Fig. 11.12). The sound, included in the file `Pipistrellus_kuhlii.wav`, was sampled at 192,000 Hz:

```
bat <- readWave("sample/Pipistrellus_kuhlii.wav")
bat

Wave Object
Number of Samples:      3841
Duration (seconds):     0.02
Samplingrate (Hertz):   192000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

We will also refer to a sound produced by a theremin, a wonderful electronic instrument which is played without physical contact by acting on two "antennas"

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_13

which control the amplitude and the frequency of the output sound. The sound here used is saved in the file `theremin.wav`:

```
theremin <- readWave("sample/theremin.wav")
theremin

Wave Object
Number of Samples:      626176
Duration (seconds):     14.2
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

13.1 Frequency Tracking

13.1.1 Dominant Frequency

The dominant frequency, that is the frequency of highest energy, can be tracked using the `seewave` function `dfreq()`. This function operates a short-time Fourier transform in the same way as `spectro()` does and then looks for the maximum of the frequency spectrum obtained for each Fourier window. The principle is then rather simple: (1) obtain the STDFT matrix and (2) identify the maximum of each STDFT matrix column. The parameters of the STDFT can be accessed with the arguments `wl` for the Fourier window length, `ovlp` for the overlap between adjacent Fourier windows, and `wn` for the Fourier tapering window as detailed in Sect. 11.7.1.3. By default, the function returns a plot displaying the estimation of the dominant frequency against time. We can run a first direct test on `sheep` taking the precaution to increase the time and frequency resolutions with appropriate `wl` and `ovlp` value (Fig. 13.1):

```
data(sheep)
wl <- 1024 ; ovlp <- 87.5
df <- dfreq(sheep, wl=wl, ovlp=ovlp)
```

The result of the analysis is saved in a two-column matrix which first column corresponds to time in s and the second column to frequency in kHz. The number of lines is the number of Fourier windows used:

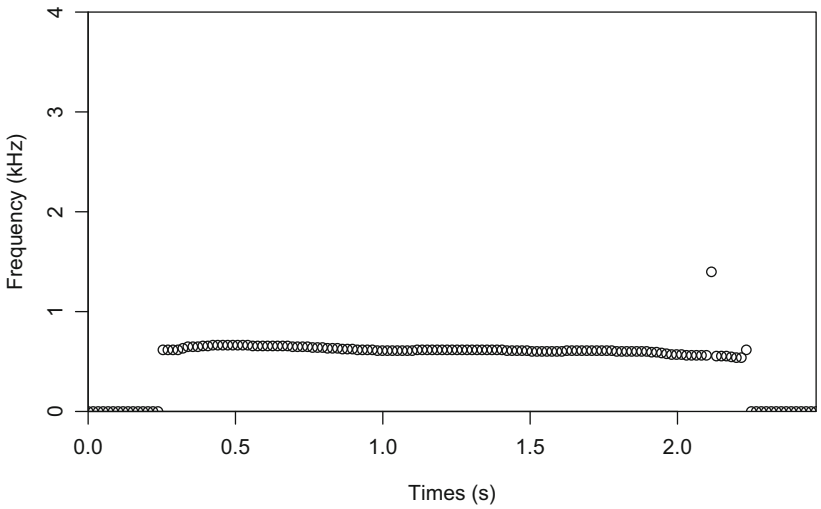


Fig. 13.1 Dominant frequency tracking with `dfreq()`. The dominant frequency of `sheep` is tracked along time calling the function `dfreq()` which computes in background a STDFT, here with a Fourier window length of 512 samples (`wl=512`) and an overlap between successive Fourier windows of 87.5% (`wl=87.5`)

```
class(df)
[1] "matrix"
dim(df)
[1] 147   2
head(df)
             x y
[1,] 0.00000000 0
[2,] 0.01692123 0
[3,] 0.03384247 0
[4,] 0.05076370 0
[5,] 0.06768493 0
[6,] 0.08460616 0
```

The dominant frequency at the start and at end of the sound was estimated to 0 corresponding to a period of silence. These values are of no meaning and should be not considered for a proper description. There are four ways to keep only relevant results calling the following arguments:

`tlim` selection of a time section of interest. Using this argument is similar to cut the original wave with either `cutw()` of `seewave` or `extractWave()` of `tuneR`.

`threshold` amplitude threshold on the signal. This argument is a shortcut to the function `afilter()`. This function, which is described in Sect. 15.3, simply replaces every amplitude value below a specific threshold by a 0 value. This amplitude threshold is expressed as a percentage in relation with the maximum of the absolute amplitude envelope. A threshold of 5% means that every value falling below 5% of the maximum of the absolute amplitude envelope is replaced by a 0. This operation may induce different degrees of artifacts as shown in Figs. 13.2 and 15.4.

`clip` amplitude threshold on the STDFT matrix with the argument `clip`. This second threshold is applied on the STDFT matrix scaled between 0 and 1. Setting `clip=0.1` discards all value of the STDFT below 0.1.

`bandpass` band-pass frequency filter with the argument `bandpass` to keep only relevant frequencies. This argument can be thought as a frequency window where to look for the dominant frequency. As an example, specifying `bandpass=c(2000,4000)` constraints the search of the dominant frequency within the [2000, 4000] Hz frequency band.

Figure 13.2, which is based on the code below, illustrates the effects of these arguments, applied independently or jointly:

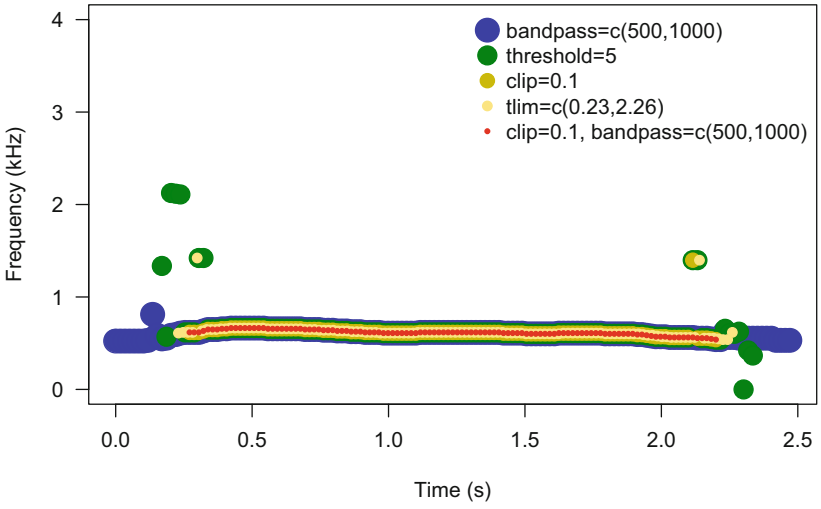


Fig. 13.2 Dominant frequency tracking with different settings of `dfreq()`. The graphic displays the results obtained with the function `dfreq()` using five different settings

```
wl <- 1024 ; ovlp <- 87.5
df1 <- dfreq(sheep, wl=wl, ovlp=ovlp,
             bandpass=c(500,1000), plot=FALSE)
df2 <- dfreq(sheep, wl=wl, ovlp=ovlp,
             threshold=5, plot=FALSE)
df3 <- dfreq(sheep, wl=wl, ovlp=ovlp,
             clip=0.1, plot=FALSE)
df4 <- dfreq(sheep, wl=wl, ovlp=ovlp,
             tlim=c(0.23,2.26), plot=FALSE)
df5 <- dfreq(sheep, wl=wl, ovlp=ovlp, clip=0.1,
             bandpass=c(500,1000), plot=FALSE)
```

We have seen in Sect. 10 a simple use of `dfreq()` with the arguments `wl` and `at`. The argument `at` can accept any numeric vector such that we can use it in a rather fancy way to estimate dominant frequency at several successive time positions. For instance, one could wish to get the dominant frequency every 0.1 s all along the recording:

```
df <- dfreq(sheep, at=seq(0, duration(sheep), by=0.1), plot=FALSE)
```

In that case the object returned includes `NA` values at the beginning and end of the recording where no measurements were taken:

```
head(df)
       x       y
[1,] 0.0      NA
[2,] 0.0 0.00000
[3,] 0.1 0.00000
[4,] 0.2 0.00000
[5,] 0.3 0.62500
[6,] 0.4 0.65625
tail(df)
           x        y
[22,] 2.0000 0.562500
[23,] 2.1000 0.546875
[24,] 2.2000 0.000000
[25,] 2.3000 0.000000
[26,] 2.4000 0.000000
[27,] 2.4705       NA
```

These NA values can be removed by using the base function na.omit():

```
na.omit(df)
```

We can do the same for a section of the recording beginning at 0.5 s and ending at 2 s:

```
df <- dfreq(sheep, at=seq(0.5, 2, by=0.1), plot=FALSE)
head(df)
       x        y
[1,] 0.0       NA
[2,] 0.5 0.656250
[3,] 0.6 0.656250
[4,] 0.7 0.656250
[5,] 0.8 0.640625
[6,] 0.9 0.625000
```

Another option could be to return a fixed number of measurements, here 25 measurements regularly spaced between the beginning and end of the recording. This is obtained by using the argument length.out of seq():

```
df <- dfreq(sheep,
            at=seq(0, duration(sheep), length.out=25), plot=FALSE)
head(df)
             x       y
[1,] 0.0000000      NA
[2,] 0.0000000 0.00000
[3,] 0.1029375 0.00000
[4,] 0.2058750 0.00000
[5,] 0.3088125 0.62500
[6,] 0.4117500 0.65625
```

A last idea could be to take a measurement at the start, in the middle, and at the end of the sound of interest. The trick is first to use the function timer() as detailed in Sect. 8.3 to obtain the start and end time positions:

```
pos <- timer(sheep, threshold=5, msmooth=c(75,0), plot=FALSE)
```

and afterward to use these time coordinates to specify where the dominant frequency has to be estimated:

```
start <- pos$s.start
end <- pos$s.end
middle <- start+(end-start)/2
df <- dfreq(sheep, at=c(start, middle, end), plot=FALSE)
head(df)
             x        y
[1,] 0.0000000       NA
[2,] 0.2734523 0.609375
[3,] 1.2541088 0.625000
[4,] 2.2347653 0.000000
[5,] 2.4705000       NA
```

13.1.2 Fundamental Frequency

13.1.2.1 `seewave` Solutions

The fundamental frequency can be assessed through (1) an autocorrelation process with the function `autoc()` or (2) a cepstral transform with the function `fund()`.

The autocorrelation consists in a cross-correlation of the signal against itself after a certain time lag (see Sect. 17.1). In other words the successive correlations are computed between $s[n]$ and $s[n+m]$, where m is the time lag, usually set to one audio sample. It is expected that correlation returns a maximal value when $m = T_{fund}$ where T_{fund} is the period of the fundamental frequency.

The `autoc()` function applies this principle on successive windows operating a kind of short-time autocorrelation: the autocorrelation is processed for each window. There are four important arguments that can help `autoc()` in finding the fundamental frequency:

`wl` the window length expressed in number of samples, by default `wl=512`,

`fmin` the minimum expected value of the fundamental frequency expressed in Hz, this means that `autoc()` will not look for a fundamental frequency below this limit,

`fmax` similarly, the maximum fundamental frequency expected expressed in Hz, this means that `autoc()` will not look for a fundamental frequency above this limit; the main motivation of this argument is to accelerate the function. However, setting a low value of `fmax`, that is a value close to the expected fundamental frequency, drastically reduces the resolution of the autocorrelation and then might return inaccurate results and even `NA` values,

`threshold` an amplitude threshold expressed in % as in `dfreq()`.

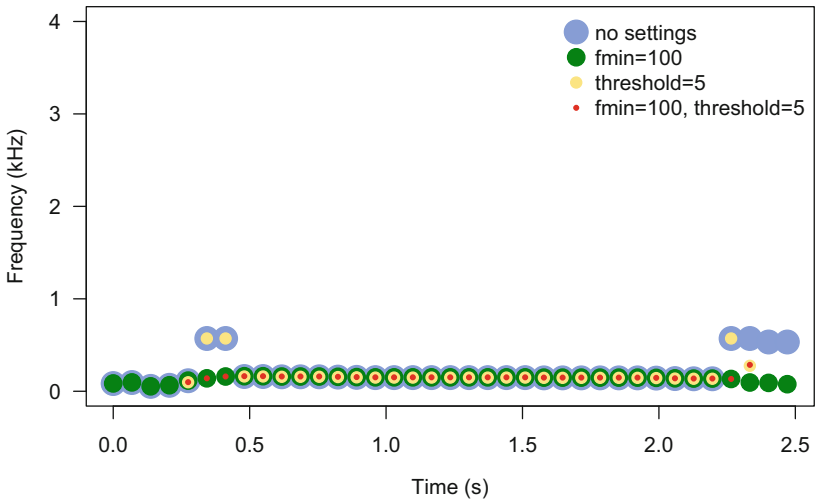


Fig. 13.3 Fundamental frequency tracking with `autoc()`. The graphic displays the results obtained with the function `autoc()` on `sheep` using four different settings. The figure was manually obtained with `plot()`, `points()`, and `legend()`

Here are successive uses of `autoc()` on `sheep` (Fig. 13.3):

```
ff1 <- autoc(sheep, plot=FALSE)
ff2 <- autoc(sheep, fmin=100, plot=FALSE)
ff3 <- autoc(sheep, threshold=5, plot=FALSE)
ff4 <- autoc(sheep, fmin=100, threshold=5, plot=FALSE)
```

As mentioned above, the argument `fmax` can be used to reduce the time of process. However, it should be used with caution as it reduces the resolution of the analysis. For instance, applying the following code on `hello` speech dataset returns `NA` only:

```
hello <- readWave("sample/hello.wav")
ff <- autoc(hello, fmax=500, threshold=5, plot=FALSE)
head(ff)
              x  y
[1,] 0.00000000 NA
[2,] 0.01111111 NA
[3,] 0.02222222 NA
[4,] 0.03333333 NA
[5,] 0.04444444 NA
[6,] 0.05555556 NA
```

A solution is to downsample the wave with the function `resamp()` (see Sect. 6.1) so that the frequency resolution is increased:

```
hello.r <- resamp(hello, g=hello@samp.rate/4, output="Wave")
ff1 <- autoc(hello.r, fmax=500, threshold=5, plot=FALSE)
head(ff1)
        x         y
[1,] 0.00        NA
[2,] 0.05 0.3529412
[3,] 0.10 0.3636364
[4,] 0.15 0.3750000
[5,] 0.20 0.4285714
[6,] 0.25 0.4137931
```

DIY 13.1 — How to plot the dominant frequency and fundamental frequency tracks on a single spectrogram

Overlaying the dominant and fundamental frequency tracks on a spectrogram can help in understanding the frequency dynamics. The use of `spectro()` as a high-level plot function and `points()` as a low-level plot function make the overplot of the frequency tracks rather easy. Here we first store the results of `dfreq()` and `dfund()` on `sheep` in dedicated objects:

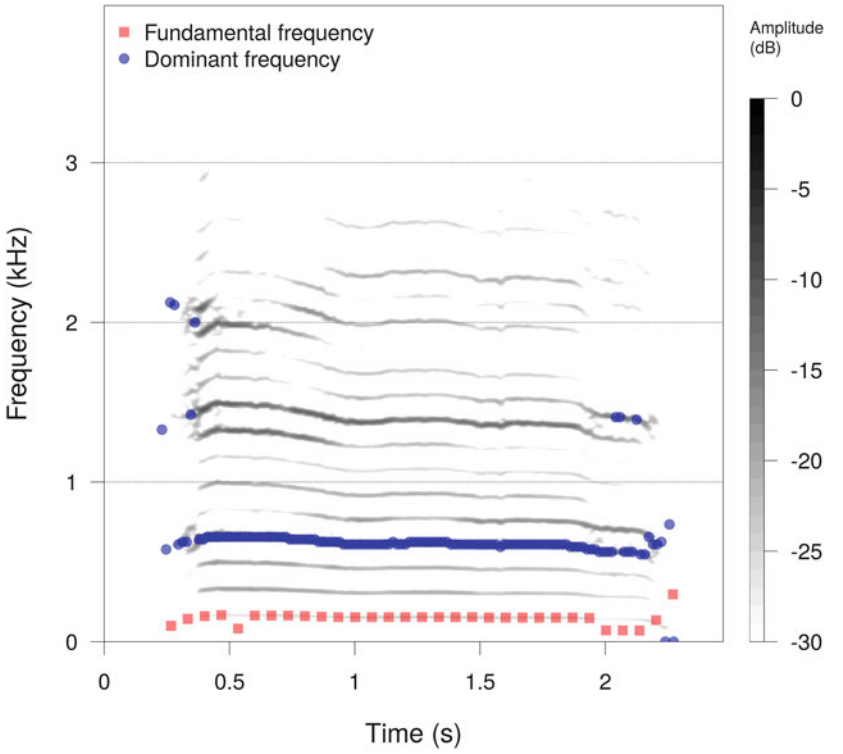
```
df <- dfreq(sheep, ovlp=75, threshold=5, plot=FALSE)
ff <- fund(sheep, fmax=300, threshold=5, plot=FALSE)
```

We then use successively `spectro()`, `points()`, and `legend()` to produce a complete graphic:

```
col <- c(rgb(1,0,0,0.5), rgb(0,0,1,0.5))
spectro(sheep, ovlp=87.5,
        palette=reverse.gray.colors.2)
points(ff, pch=15, col=col[1])
points(df, pch=19, col=col[2])
legend("topleft",
       legend=c("Fundamental frequency",
                "Dominant frequency"),
       pch=c(15,19), col=col, bty="n")
```

(continued)

DIY 13.1 (continued)



The other option is to run a cepstral analysis with the function `fund()` already introduced in Sect. 10.1.3.4. The function estimates the first rhamonic of the cepstrum that corresponds to the fundamental frequency. As `dfreq()`, `fund()` can be used for a single time position or for a series of time positions such that variations of the fundamental frequency can be estimated along time. The main arguments of the function `fund()` are:

`wl` the window length expressed in number of samples, by default `wl=512`,
`ovlp` the overlap between successive windows, as explained in Sect. 11.1.2.2, by default `ovlp=0`,
`fmax` the maximum expected value of the fundamental frequency expressed in Hz, this means that `fund()` will not look for a fundamental frequency above this limit,
`threshold` an amplitude threshold expressed in % as in `dfreq()`.

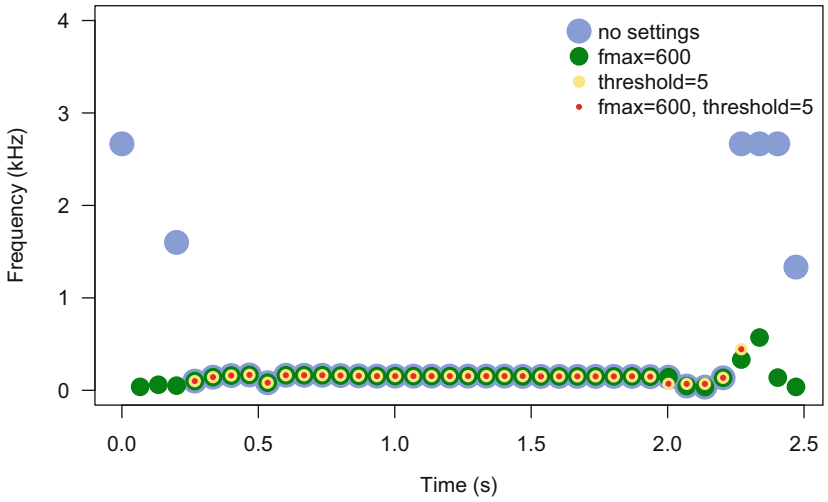


Fig. 13.4 Fundamental frequency tracking with `fund()`. The graphic displays the results obtained with the function `fund()` on `sheep` using four different settings. The figure was manually obtained with `plot()`, `points()`, and `legend()`

Here are successive uses of `fund()` on `sheep` (Fig. 13.4 and DIY box 13.1):

```
ff1 <- fund(sheep, plot=FALSE)
ff2 <- fund(sheep, fmax=600, plot=FALSE)
ff3 <- fund(sheep, threshold=5, plot=FALSE)
ff4 <- fund(sheep, fmax=600, threshold=5, plot=FALSE)
```

13.1.2.2 `tuneR` Solutions

The package `tuneR` includes the function `FF()`, which is a wrapper of another function `FFpure()`, to identify the fundamental frequency of a sound based on analysis of the STDFT matrix.

We can first try the function `FF()` on `sheep` as we did with the `seewave` functions in Sect. 13.1.2.1. The function `FF()` requires a periodogram as an input so that we obtain the results in a two-step process: (1) computation of the periodogram, that is the STDFT, with a Fourier window of 512 samples, and (2) extraction of the fundamental frequency `FF()` from the periodogram (Fig. 13.5):

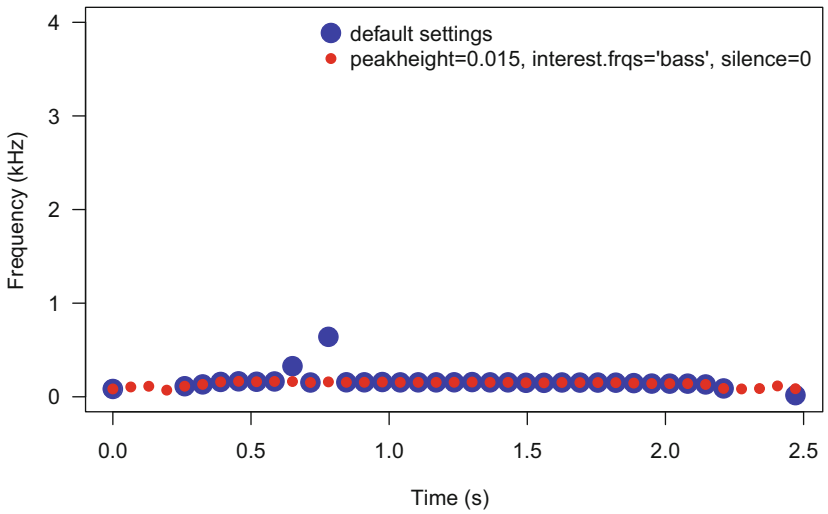


Fig. 13.5 Fundamental frequency tracking with `FF()`. The graphic displays the results obtained with the function `FF()` on `sheep` using default and tuned settings. The figure was manually obtained with `plot()`, `points()`, and `legend()`

```
p <- periodogram(sheep, width=512)
ff1 <- FF(p)
ff1
  [1]  84.01657        NA        NA        NA 113.26127
  [6] 132.88944 159.16253 165.45161 161.00843 164.11472
 [11] 327.36702 153.00641 640.99107 155.16750 155.01134
 [16] 158.00507 154.30934 154.71226 154.48895 155.20370
 [21] 157.48849 154.35723 154.37106 149.96773 149.43067
 [26] 153.39314 152.11066 152.36927 151.24859 146.49484
 [31] 141.59863 140.87896 140.97265 132.18679  89.64569
 [36]        NA        NA        NA  17.82383
```

The function `FF()` has arguments to improve the fundamental frequency detection. Most of these arguments are related to Western music and even more particularly to singing:

`peakheight` an amplitude threshold for the frequency peak height of the fundamental frequency,

`silence` a time threshold expressed in proportion specifying the proportion of silence or noise,

`diapason` the diapason frequency (440 Hz),

`notes` a vector of integers indicating the notes (in halftones from diapason A, that is from 440 Hz) that are expected (see Sect. 9.4.2),

`interest.frqs` a frequency filter to indicate where the fundamental frequency is expected, specifying `"bass"` will for instance force the function to look into low frequency.

Here is an example of the use of these arguments applied on `sheep` (Fig. 13.5):

```
ff2 <- FF(p, peakheight=0.015, interest.frqs="bass", silence=0)
```

Now, we can play with the music of the `theremin` and try to go from fundamental frequency estimation toward musical notation:

```
p <- periodogram(theremin, width=512)
ff <- FF(p)
```

The fundamental frequency stored in `ff` can be changed in note numbers using the function `noteFromFF()`, as introduced in Sect. 9.4.2:

```
notes <- noteFromFF(ff, 440)
head(notes)
[1]  NA  NA -24 -24  NA -22
```

If we wish to convert these quite mysterious numbers into English musical notation, we could use the function `notenames()` taking care of removing `NA` values:

```
head(notenames(na.omit(notes)))
[1] "A"  "A"  "B"  "A#" "a"  "a"
```

The function `melodyplot()` can display these notes in a nice time $\times$ note plot. The function needs the periodogram and the note numbers as inputs (Fig. 13.6):

```
melodyplot(p, notes)
```

At this step, we follow the changes of notes at a time resolution corresponding to the parameters of the STDFT computed with `periodogram()`. This resolution is too high if one wish to translate the sound into a musical notation. It is necessary to

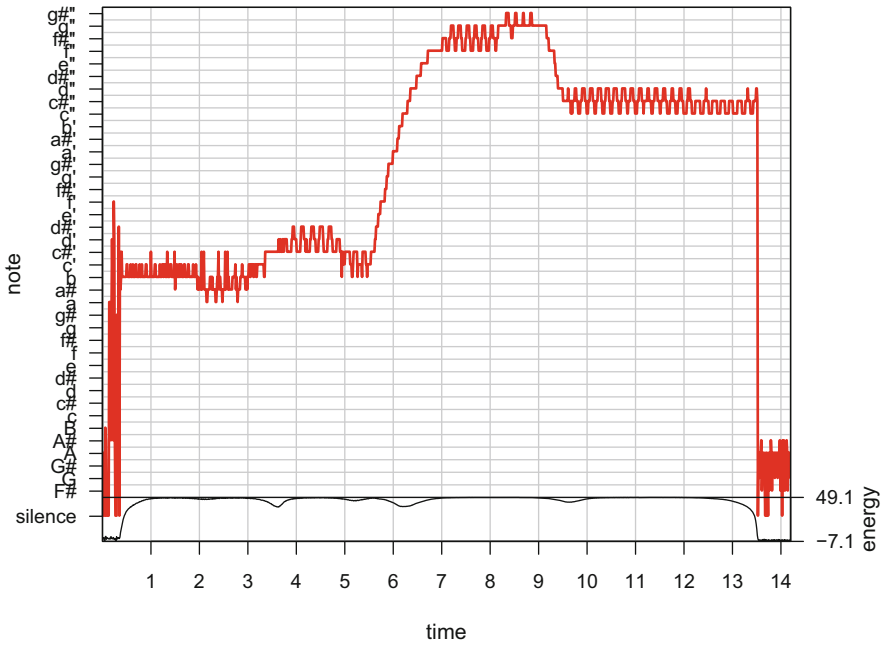


Fig. 13.6 Melody plot. The `tuneR` function `melodyplot()` displays the notes estimated from the fundamental frequency, here the fundamental frequency of the `theremin` sound

reduce the time resolution such that we end up with a few notes only. The function `quantize()` can apply such time binning, its argument `parts` setting the number of notes. The following codes ensure a reduction from more than 1000 notes to only 16 notes:

```
length(notes)
[1] 1223
qnotes <- quantize(notes, p@energy, parts=16)
length(qnotes$notes)
[1] 16
```

We can now plot these notes in a sort of sheet music with the graphical function `quantnote()`. The number of bars is specified with the argument `bars`. We place the 16 notes into four bars with a 4/4 time (Fig. 13.7):

```
quantplot(qnotes, bars=4)
```

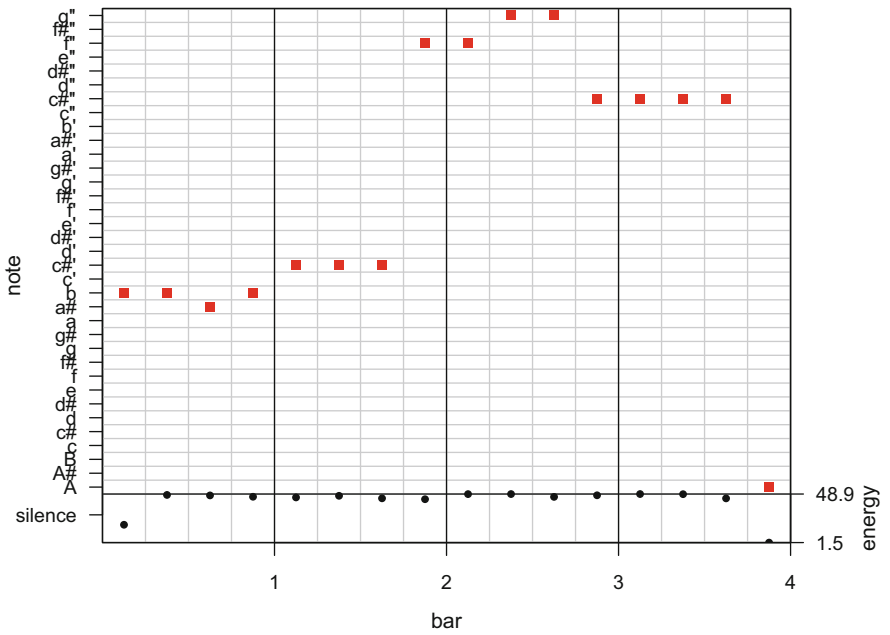


Fig. 13.7 Melody quantization plot. The `tuneR` function `quantplot()` displays the notes estimated from the fundamental frequency after having binned the time scale, here for the `theremin` sound

13.1.2.3 `phonTools` Solutions

The package `phonTools` provides a function, `pitchtrack()`, to detect the fundamental frequency of voice. The process relies on an algorithm which is based, among others, on the autocorrelation (Boersma 1993). The fundamental frequency is estimated over a time delimited window that slides along the signal, making `pitchtrack()` a short-time function. The parameters of the sliding window are set with the arguments `windowlength` and `timestep`, both expressed in ms. These two arguments correspond to `wl` and `ovlp` found in short-time functions of `seewave`, as `spectro()`. The track of the fundamental frequency can be limited between two frequency limits with the arguments `f0range`. The function has also an argument, `periodicity`, to select the autocorrelations which have a coefficient above a specified value. This threshold can be used to exclude voiceless sections from the analysis.

In the following example, we use the default values that are well adapted to the voice data `hello`, that is `windowlength=50`, `timestep=2`, `f0range=c(60, 400)`, and `minacf=0.5`. We only set `show=FALSE` to save the results in an object:

```
res <- pitchtrack(hello@left, fs=hello@samp.rate, show=FALSE)
```

The object returned is a `data.frame` with three columns, the time expressed in ms (`$time`), the estimation of the fundamental frequency expressed in Hz (`$f0`), and the correlation coefficient of the autocorrelation in [0, 1] (`$acf`):

```
str(res)
'data.frame': 248 obs. of  3 variables:
 $ time: num  61 63 65 67 69 71 73 75 77 79 ...
 $ f0  : num  387 384 378 375 375 ...
 $ acf : num  0.525 0.617 0.687 0.744 0.79 ...
```

We can use these data to plot the results on a spectrogram obtained with `spectro()` (Fig. 13.8):

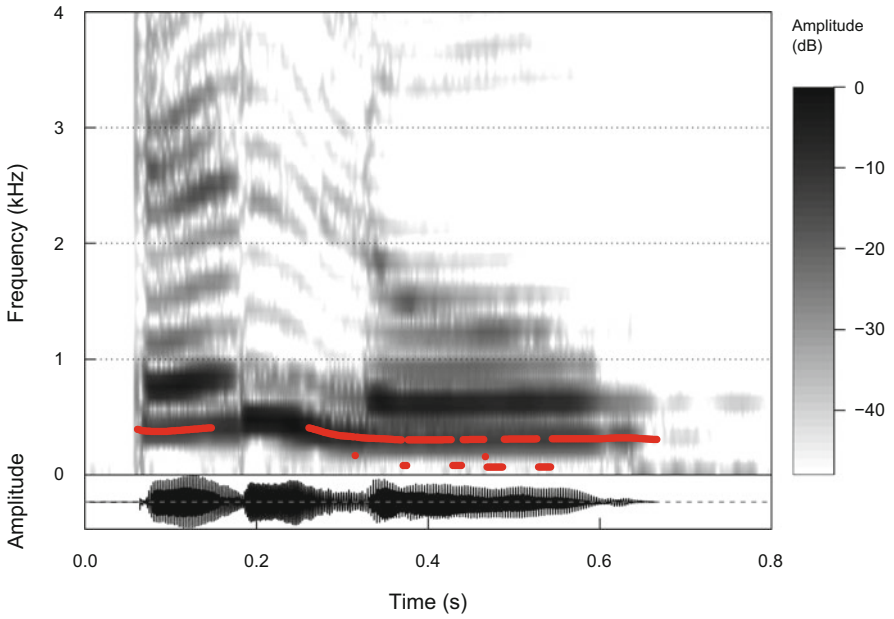


Fig. 13.8 Fundamental frequency tracking with `pitchtrack()`. The fundamental frequency of the voice data `hello` is detected and tracked with the function `pitchtrack()` of `phonTools`. The result is plotted over a spectrogram obtained with `spectro()` of `seewave`

```
spectro(hello, flim=c(0,4), ovlp=87.5,
        palette=reverse.gray.colors.2, collevels=seq(-48,0,1),
        osc=TRUE)
points(res$time/1000, res$f0/1000, col="red", pch=19, cex=0.75)
```

13.1.2.4 `soundgen` Solutions

The package `soundgen` combines four solutions to track the fundamental frequency into a single function, `analyze()`. This function uses the autocorrelation and the cepstrum as previously introduced but also two other methods based on an analysis of the peaks of the frequency spectrum (detection of the lowest-frequency peak and computation of harmonics ratio). The function is rather simple to use, we just need to coerce the data into a numeric vector with `as.numeric()` and store the results in an object (Fig. 13.9):

```
res <- analyze(as.numeric(hello@left), hello@samp.rate)
legend("topright",
       legend=c("autocorrelation", "lowest frequency peak",
                "ratio of harmonics", "cepstrum", "interpolation"),
       pch=c(16,2,3,7,NA), lty=c(rep(NA,4),1),
       col=c("green", "red", "orange", "violet", "blue"),
       bg="white")
```

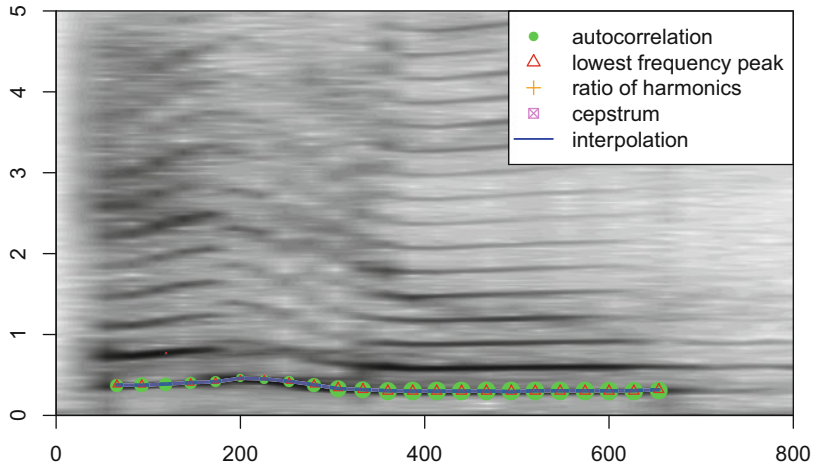


Fig. 13.9 Fundamental frequency tracking with `analyze()`. The fundamental frequency of the voice data `hello` is detected and tracked with the function `analyze()` of `soundgen` following four methods which, here, return almost the same results. The legend was added manually with `legend()`

The function offers several options to control for the parameters of STDFT, the parameters of the four methods used, amplitude frequency, and time thresholds. It also processes an interpolation between the tracks that can also be controlled. A full explanation of `analyze()` is provided in the vignette "Acoustic analysis with soundgen":

```
vignette("acoustic_analysis", package="soundgen")
```

13.1.3 Formants

We have seen in Sect. 12.2.2 how to localize and describe voice formants with the function `findformants()` of `phonTools`. It is easy to imagine that the function can be used with a window sliding along the sound, in the same way as `fund()` and `pitchtrack()`, such that the formants can be tracked along time. This short-time version of `findformants()` is available in the function `formanttrack()` of the same package `phonTools`. The function has the same arguments `windowlength` and `timestep` than `pitchtrack()` and additional arguments specific to the detection of the formants: `minformant` the minimum frequency expressed in Hz, `cutoff` the maximum frequency expressed in Hz, `maxbw` the maximum formant bandwidth expressed in Hz, `formants` the maximum number of formants to detect, and `periodicity` which is a threshold level related to the autocorrelation. We also can choose not to plot the results but to store them in an object by specifying `show=FALSE` and `returnbw=TRUE`:

```
fs <- hello@samp.rate
wl <- 512
res <- formanttrack(hello@left, fs=fs,
                    windowlength=1000*wl/fs,
                    timestep=1000*wl/fs/2,
                    minformant=250, cutoff=4000, maxbw=300,
                    formants=3, periodicity=0.95,
                    show=FALSE, returnbw=TRUE)
```

The value of `formanttrack()` is a `data.frame`, the first column is time in ms (`$time`), the next ith columns are the frequency of the ith formants (`$f#`), and the last ith columns are the bandwidth of the ith formants (`$b#`). As we set `formants=3`, we end here with $1 + 3 + 3 = 7$ columns:

```
str(res)
'data.frame': 115 obs. of  7 variables:
 $ time: num  5.5 32.4 53.9 70 80.8 ...
 $ f1  : num  0 0 1670 2683 745 ...
 $ f2  : num  0 0 2692 0 2554 ...
 $ f3  : num  0 0 0 0 0 ...
 $ b1  : num  0 0 241.7 276.8 61.2 ...
 $ b2  : num  0 0 89.9 0 261.4 ...
 $ b3  : num  0 0 0 0 0 ...
```

The 0 values in `res`, which indicates that the analysis was not run, can be replaced by `NA`:

```
res[res==0] <-  NA
```

so that these values are not used when overlaying the formant frequencies on the spectrogram (Fig. 13.10):

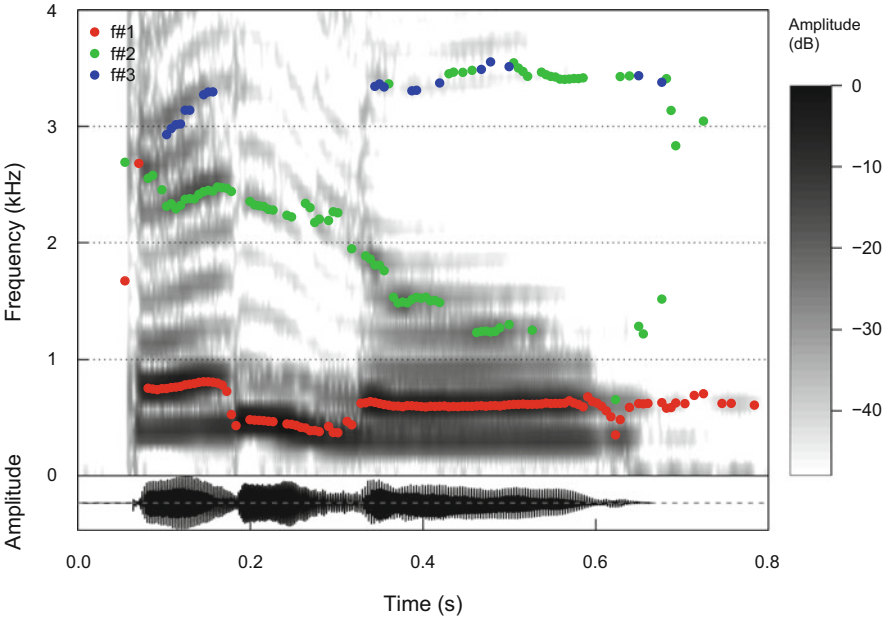


Fig. 13.10 Formant tracking with `formanttrack()`. The formants of the voice data `hello` are detected and tracked with the function `formanttrack()` of `phonTools`. The results, here for three formants, are plotted over a spectrogram obtained with `spectro()` of `seewave`

```
spectro(hello, flim=c(0,4), ovlp=87.5,
        palette=reverse.gray.colors.2, collevels=seq(-48,0,1),
        osc=TRUE)
points(res$time/1000, res$f1/1000, pch=19, col=2)
points(res$time/1000, res$f2/1000, pch=19, col=3)
points(res$time/1000, res$f3/1000, pch=19, col=4)
legend("topleft", legend=paste("f", 1:3, sep="#"),
       col=2:4, pch=19, bty="n")
```

13.1.4 Instantaneous Frequency

13.1.4.1 Hilbert Transform

The analytic signal through the Hilbert transform can give access to both instantaneous amplitude envelope and instantaneous frequency. The Hilbert transform has the great advantage to return a time series for the instantaneous frequency $f(t)$ that has the same sampling rate than the original time signal $s(t)$ such that time resolution is kept. The Hilbert transform provides nice results for monotonal sounds but may go wrong with multi-tonal sounds.

We have seen in Sect. 5.2.1 that the analytic signal $\xi(t)$ can be written as:

$$\xi(t) = s(t) + i\,H(t)$$

where $s(t)$ is the signal, $H(t)$ is the Hilbert transform, and $i^2 = -1$.

We have also seen that this expression can be written under the following form:

$$\xi(t) = a(t)e^{i\varphi(t)}$$

where $a(t)$ is the instantaneous amplitude and the $\varphi(t)$ is the instantaneous phase. $\varphi(t)$ can be therefore obtained by computing the argument of $\xi(t)$:

$$\begin{aligned}\varphi(t) &= \arg(\xi(t)) \\ &= \arctan\left(\frac{H(t)}{s(t)}\right)\end{aligned}$$

and the instantaneous frequency can be derived from the instantaneous phase by computing its time derivative according to:

$$\begin{aligned} f(t) &= \frac{1}{2\pi}\frac{\delta\varphi(t)}{\delta t} \\ &= \frac{1}{2\pi}\frac{\delta \arctan\left(\frac{H(t)}{s(t)}\right)}{\delta t} \\ &= \frac{1}{2\pi}\frac{s(t)\dot{H}(t) - H(t)\dot{s}(t)}{s^2(t) + H^2(t)} \end{aligned}$$

where $\dot{s}(t)$ and $\dot{H}(t)$ are the time derivatives of $s(t)$ and $H(t)$.

The function `ifreq()`, which calls the function `hilbert()` that returns the analytic signal $\xi(t)$, computes and plots the instantaneous phase and the instantaneous frequency. The argument `phase` allows to choose between phase and frequency, and the argument `threshold` is an amplitude threshold expressed in % as in `dfreq()` and `fund()`. Here is an example of the use of `ifreq()` on `tico` (Fig. 13.11):

```
ifr <- ifreq(tico, threshold=6, col="blue")
```

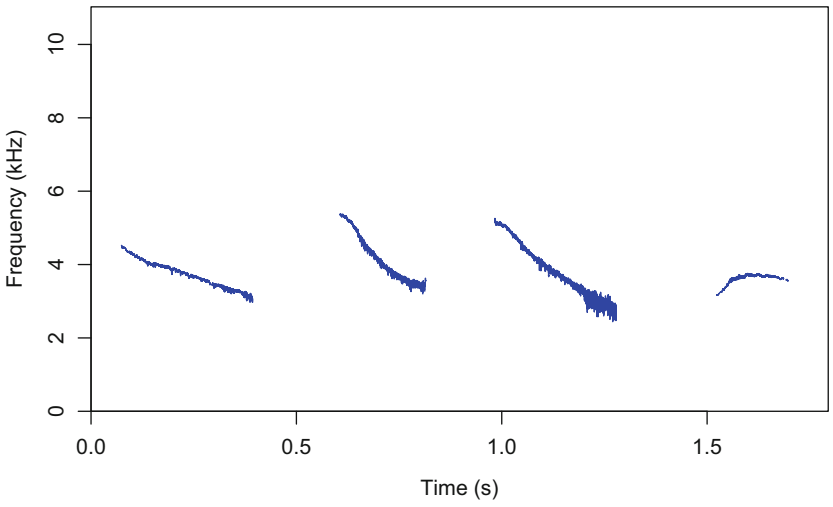


Fig. 13.11 Instantaneous frequency tracking with `ifreq()`. The instantaneous frequency is computed and plotted with the function `ifreq()` on `tico`. An amplitude threshold of 6% was applied to select the notes

The value of `ifreq()` is a two-item list, each item being a two-column matrix. The item `$p` contains the instantaneous phase in rad and `$f` contains the instantaneous frequency in kHz as illustrated for six samples taken into the first note:

```
ifr$p[2000:2005,]
           time          phi
[1,] 0.09065989  0.12905733
[2,] 0.09070524  1.36408262
[3,] 0.09075059  2.59947283
[4,] 0.09079594 -2.44545131
[5,] 0.09084130 -1.20550973
[6,] 0.09088665  0.03293986
ifr$f[2000:2005,]
           time    ifreq
[1,] 0.09065989 4.334156
[2,] 0.09070524 4.335437
[3,] 0.09075059 4.345512
[4,] 0.09079594 4.351409
[5,] 0.09084130 4.346173
[6,] 0.09088665 4.339165
```

Tracking the instantaneous frequency using the Hilbert transform works very well for monotonal sounds but might be altered for multi-tonal sounds. For instance, the bat call stored in `bat` is monotonal in its first parts and then bitonal in its second part. The instantaneous frequency is properly estimated for the first but not second part as illustrated in the following plot that combines the use of `spectro()` and `ifreq()` (Fig. 13.12):

```
spectro(bat, wl=256, ovlp=87.5,
        palette=reverse.gray.colors.2)
ifr <- ifreq(bat, threshold=5, plot=FALSE)
lines(ifr$f, col=rgb(1,0,0,0.75))
```

The function `fma()` opens the possibility to look for periodic frequency modulations by applying the Fourier transform on the instantaneous frequency as we applied the Fourier transform on the Hilbert amplitude envelope to seek for periodicity in the amplitude modulations with the function `ama()` (see Sect. 8.4). The `theremin` sound is regularly modulated in frequency at a rate of approximately of 6 Hz as returned by `fma()` applied on a 2 s section (Fig. 13.13):

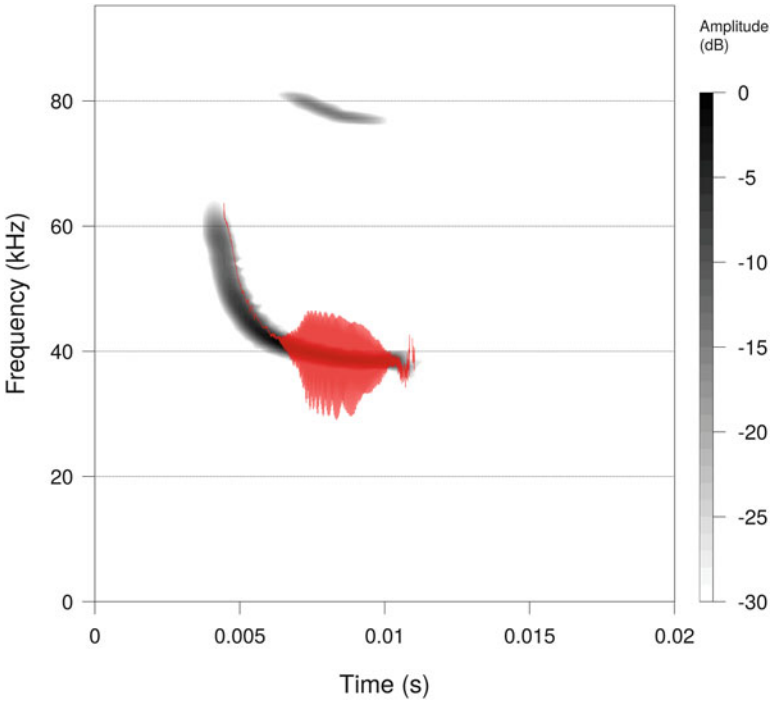


Fig. 13.12 Artifact of instantaneous frequency tracking. The instantaneous frequency is computed and plotted with the function `ifreq()` on `bat`. An amplitude threshold of 5% was applied to select the call. The function can properly estimate the instantaneous frequency when the sound is monotonal but not when an harmonic appears making the sound bitonal

```
theremin.sel <- cutw(theremin, from=10, to=12, output="Wave")
fma(theremin.sel, flim=c(0,0.1))
```

13.1.4.2 Zero-Crossing

The zero-crossing (ZC) is a simple and intuitive method to measure the instantaneous frequency (Mbu Nyamsi et al. 1994). As defined in Sect. 2.2.2, the frequency of a sine wave is the inverse of the period, that is the inverse of the duration of a complete cycle. A cycle can be delimited by successive maxima (or minima) as illustrated in Fig. 2.2 or by positions where the wave crosses the zero line, that is the line where the pressure is null (p_0). The ZC seeks where the wave crosses the zero line and measures the interval T_{zc} between the ith and $(i + 2)$th zero-crossings (Fig. 13.14).

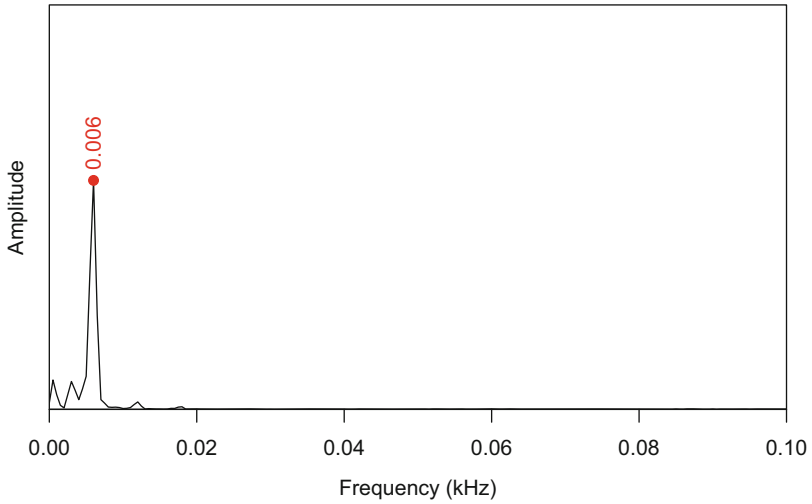


Fig. 13.13 Frequency modulation analysis of the theremin sound. The function `fma()` shows a first peak at 0.006 kHz. This peak was here identified using `identify=TRUE` and then added on the graphic with the low-level plot functions `points()` and `text()` as in Fig. 8.11. Note that the peak can also be automatically identified using `fpeaks()`

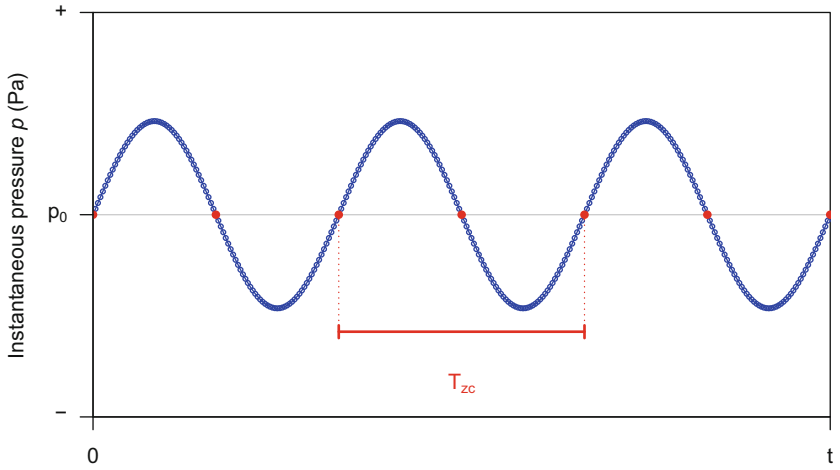


Fig. 13.14 Zero-crossing principle. Positions where the signal crosses the zero line are identified (red points) and used to estimate the instantaneous period T_{zc} and therefore the instantaneous frequency f_{zc}

The ZC method is simple to implement and to use but has some limitations. First, it only works for monotonal sounds. The introduction of overtones can change the shape of the wave so that the positions of the zero-crossings do not correspond to the instantaneous frequency (Fig. 13.15). Second, the accuracy of the ZC method

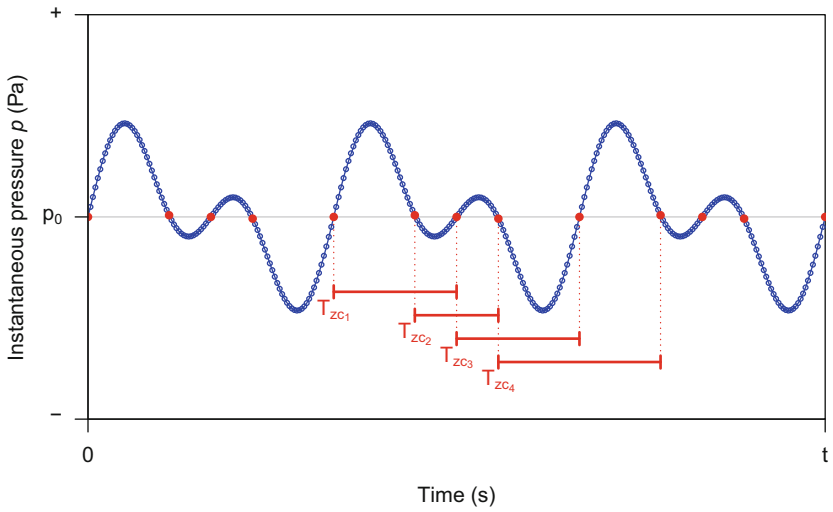


Fig. 13.15 Zero-crossing with a multi-tonal sound. A sound made of different frequencies, here a fundamental and its first harmonic, crosses the zero line several times such that the instantaneous frequency varies around four values

decreases significantly when approaching the Nyquist frequency, the sampling rate being too low to measure correctly the distance between consecutive zero-crossing (Fig. 13.16). A possible solution to reduce errors in ZC method when approaching the Nyquist frequency is to artificially increase the number of samples through an interpolation process as illustrated in the bottom graphic of Fig. 13.16.

The function `zc()` of `seewave` implements the zero-crossing principle with an `interpol` argument to increase the number of samples and hence to increase the accuracy of the results and a `threshold` argument to selection signal section through an amplitude threshold expressed in %. The following code gives an example of `zc()` on the `bat` sound without and with interpolation (Fig. 13.17):

```
par(mfrow=c(2,1))
zc(bat, threshold=7, col="blue", main="no interpolation")
zc(bat, threshold=7, col="blue", interpol=10,
     main=expression(paste("with a ",
                           symbol("\264"), " 10 interpolation")))
```

Deriving from the ZC method, the zero-crossing rate (ZCR) consists in counting the number of times the wave crosses the zero line. The ZCR is often used in speech analysis. For a signal $s[n]$ made of N samples, the ZCR is computed according to:

$$zcr = \frac{1}{2 \times N} \sum_{n=0}^{N-1} |sgn(s[n+1]) - sgn(s[n])|$$

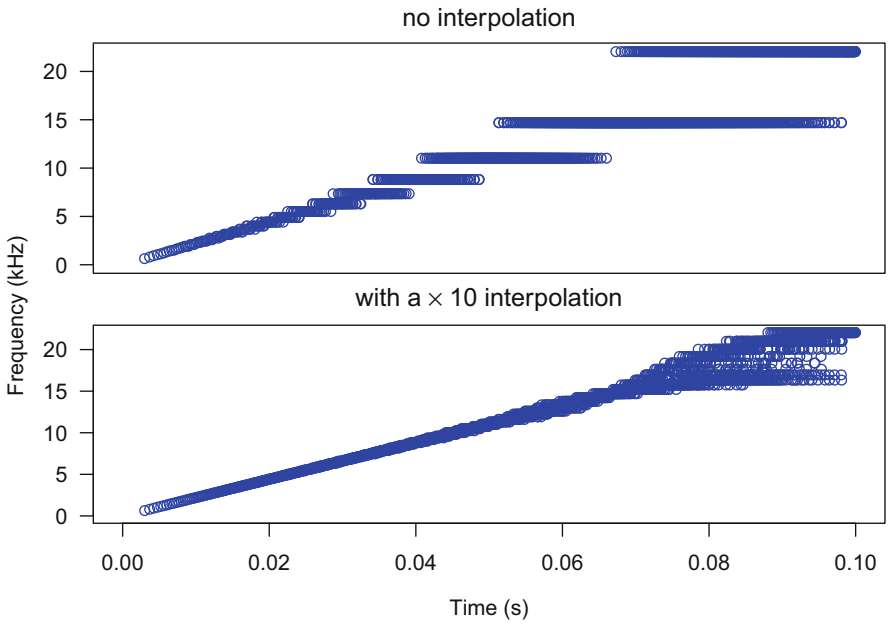


Fig. 13.16 Zero-crossing limitation and interpolation solution. The figure is based on the analysis of a 0.1 s sound sampled at 44,100 Hz with a linear frequency increasing from 0 to 22,050 Hz. Without interpolation the ZC is very inaccurate when getting close to the Nyquist frequency (top). This error can be reduced by interpolating the original signal, here with a $\times 10$ factor (bottom)

where sgn is the signum function defined as follows:

$$sgn(x[n]) = \begin{cases} 1 & \text{if } x[n] \geq 0 \\ -1 & \text{if } x[n] < 0 \end{cases}$$

The `seewave` function `zcr()` can return a single ZCR value for a complete object:

```
zcr(bat, wl=NULL)
[1] 0.4548295
```

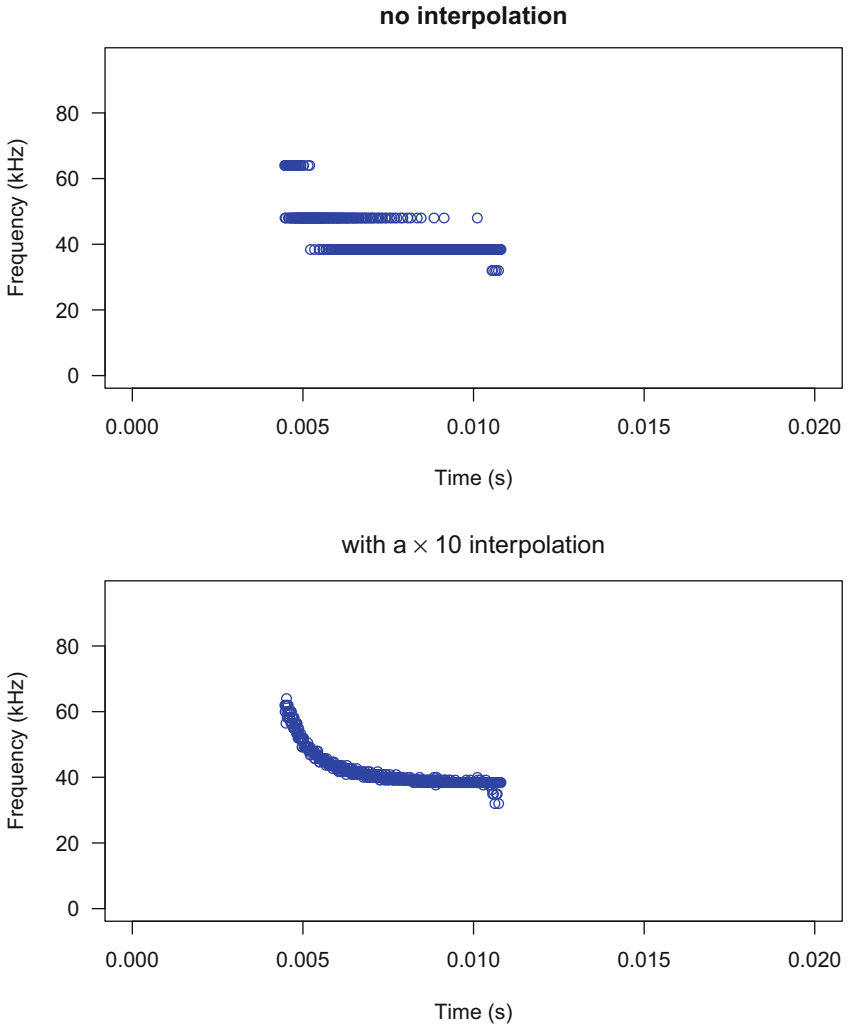


Fig. 13.17 Instantaneous frequency tracking with `zc()`. The instantaneous frequency of the `bat` call is estimated using the zero-crossing principle without (top) and with a tenfold interpolation (bottom)

or can compute successive ZCR values through a short-time process. The usual `wl` and `ovlp` arguments can be used to tune the process (Fig. 13.18):

```
zcr(bat, wl=512, ovlp=87.5)
```

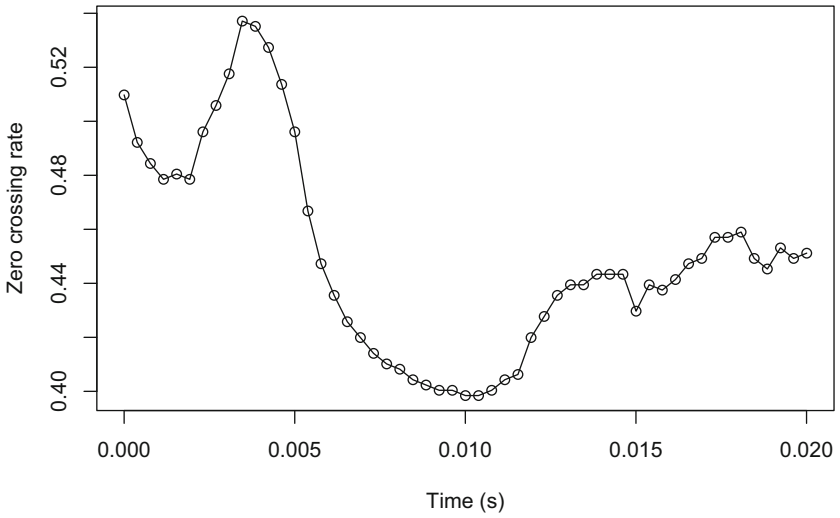


Fig. 13.18 Zero-crossing rate. The zero-crossing rate method is used on `bat` sound by dividing the signal in 53 successive windows by setting the arguments `wl=512` and `wl=87.5`

The zero-crossing rate can also be used to estimate the instantaneous frequency of the sound as demonstrated in DIY box 13.2.

DIY 13.2 — How to derive the instantaneous frequency using zero-crossing rate

The ZCR is another way, probably faster, to estimate the instantaneous frequency as exemplified here on the `hello` recording. The trick is simply to multiply the frequency results of the function `zcr` by half the sampling frequency.

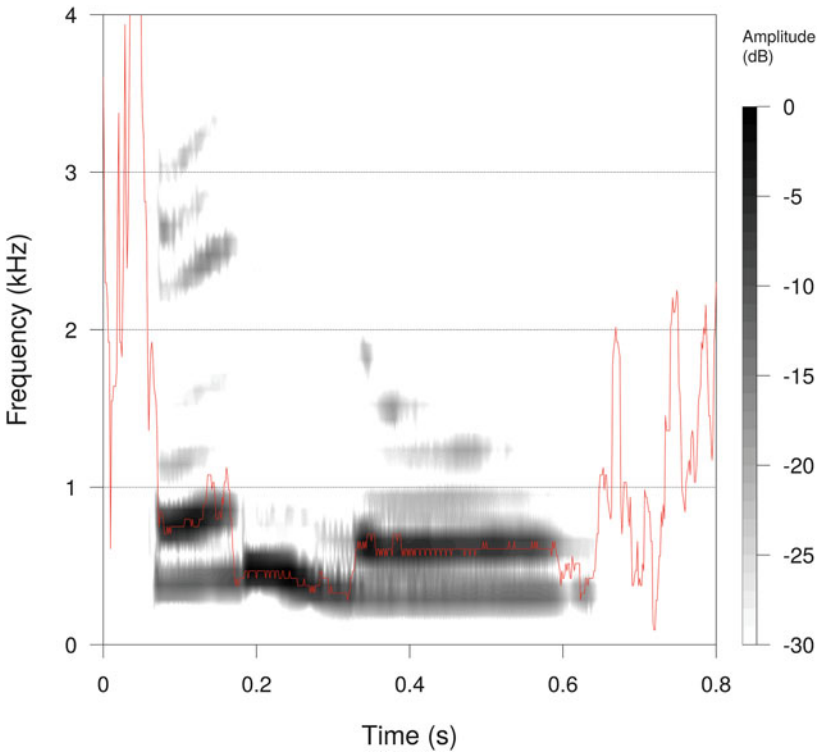
```
zcr(hello)*hello@samp.rate/2
```

It is then possible to overlay the results on the spectrogram with this simple code. Note that the sampling frequency is here divided by 2000 to get results in Hz rather than in kHz:

```
spectro(hello, flim=c(0,4), ovlp=87.5,
        palette=reverse.gray.colors.2)
res <- zcr(hello, ovlp=87.5, plot=FALSE)
lines(res[,1], res[,2]*hello@samp.rate/2000, col="red")
```

(continued)

DIY 13.2 (continued)



13.2 Energy Tracking

The Teager-Kaiser energy operator, abbreviated TKEO, is a transform of the signal that estimates the energy that would be required to a mechanical process—a spring-mass system—to generate a signal made of a single frequency. The idea comes from the fact that the energy E of a mass m suspended by a spring oscillating with an amplitude A and a frequency ω can be written (Kaiser 1990):

$$E = \frac{1}{2} m A^2 \omega^2$$

so that E is proportional to the square of both amplitude and frequency:

$$E \propto A^2 \omega^2$$

The energy of a time series, noted Ω, can be measured by using only three adjacent samples using the following equation (Kaiser 1990):

$$\Omega(t) = \dot{x}^2(t) - x(t) \times \ddot{x(t)}$$

which is written in its discrete version as:

$$\Omega[n] = x[n]^2 - x[n-1] \times x[n+1]$$

The equations can be generalized referring to a lag parameter M and an exponent parameter m (Kvedalen 2003):

$$\Omega[n] = x[n]^{\frac{2}{m}} - (x[n-M] \times x[n+M])^{\frac{1}{m}}$$

This equation is fast to compute and returns a time series with a similar sampling frequency than the original signal. The Teaser-Kaiser operator (TKEO) can be then used to track amplitude modulations (AM) and/or frequency modulations (FM) as illustrated in Fig. 13.19.

The TKEO suffers, however, some limitations. As it is based on the energy equation of a simple mass-spring system oscillating at a single frequency, it can be used for monotonal sounds only (Fig. 13.20). In addition, the expression used is based on the second derivative of the time series such that the TKEO can give an appropriate measurement of the energy only if the sampling frequency f_s is four times the frequency of oscillation, that is four times the main frequency of the sound (Fig. 13.21). The TKEO is also sensitive to noise (Fig. 13.22).

The TKEO is implemented in the `seewave` function `TKEO()`. As usual the data have to be provided in the first argument `wave`, and the lag and exponent parameters can be specified in the arguments `M` and `m` arguments, respectively. These two arguments have a default value of 1 for the computation of a conventional TKEO. By default the function plots the energy against time and saves the results in a two-column matrix which first column is time in s and second column is energy. $m \div 2$ `NA` values are added at the start and end to keep a similar number of samples as in the original input object `wave`. A straight use of TKEO, here on `tico`, is therefore:

```
res <- TKEO(tico)
```

The function `TKEO()` should be employed carefully due to the limitations of the TKEO exposed above. For instance, the TKEO properly works on `tico` because the sound is monotonal and the maximum frequency, around 5500 Hz, is just below

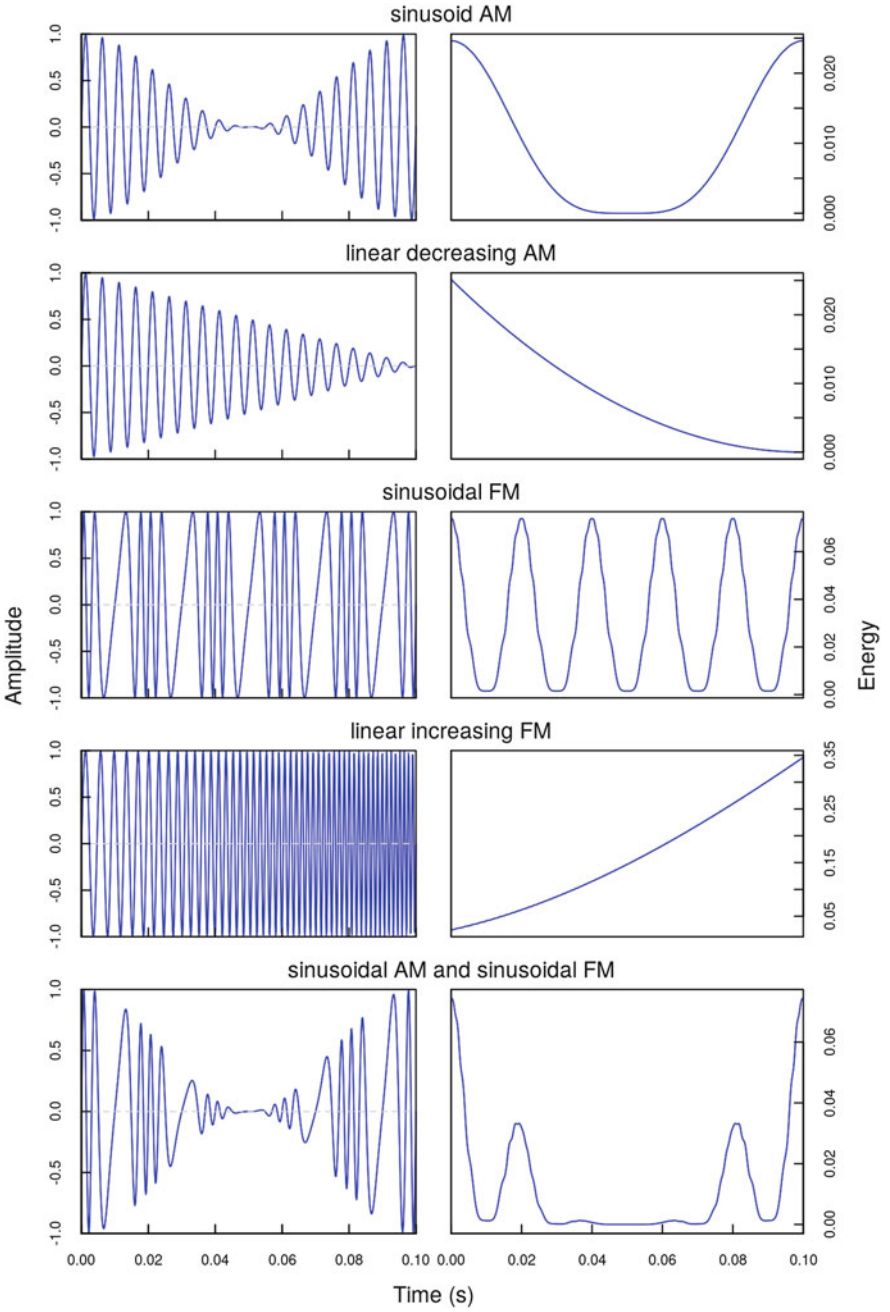


Fig. 13.19 Teager-Kaiser energy operator. Examples of TKEO applied to amplitude modulated (AM) and/or frequency modulated (FM) sounds

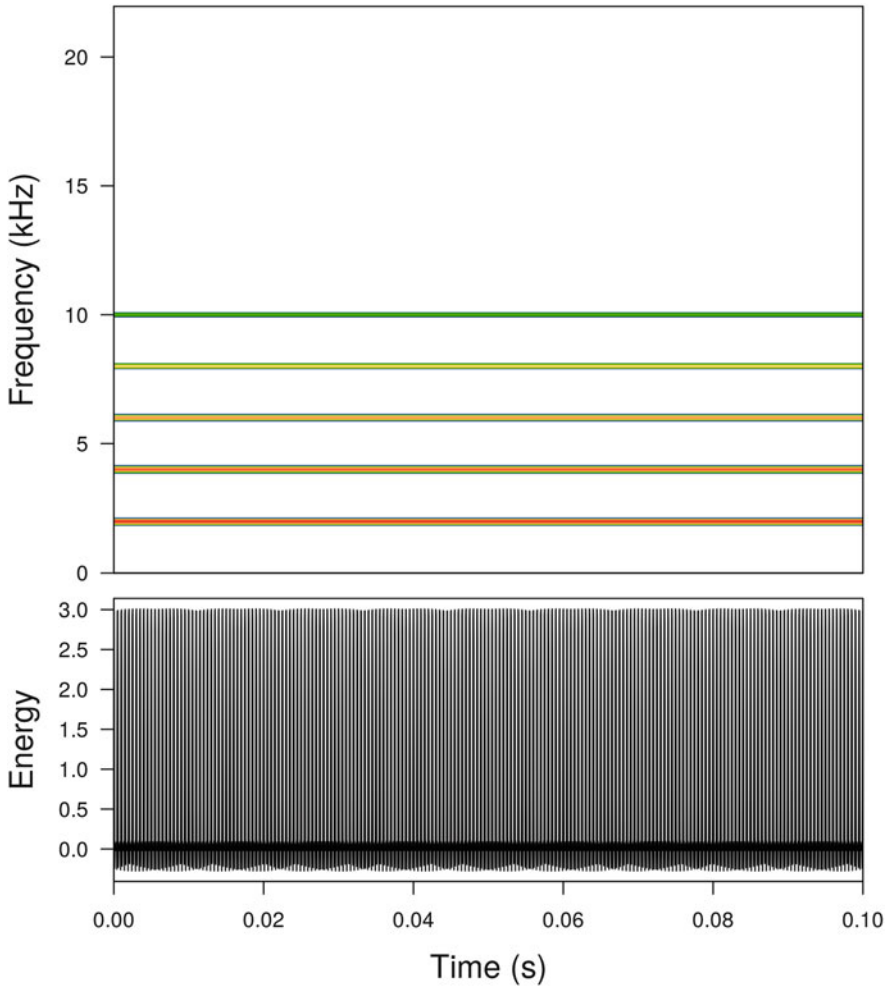


Fig. 13.20 Teager-Kaiser energy operator with multi-tonal sound. The TKEO does not return appropriate results with a multi-tonal sound, as illustrated here with a sound with a carrier frequency at 2000 Hz and four harmonics. Spectrogram (top) and TKEO (bottom)

one fourth of the sampling frequency $22{,}050 \div 4 = 5512.5$ Hz (Fig. 13.23, top). However, the TKEO returns useless results if applied on `sheep` because the bleat sound is a series of harmonics going up 2800 Hz, significantly above one fourth of the sampling frequency $8000 \div 4 = 2000$ Hz (Fig. 13.23, middle). A simple solution to reach this condition is to filter out unwanted frequency bands to focus on a single frequency band of interest. This frequency selection can be processed by employing a band-pass frequency filter as detailed in Chap. 14. If we keep on the `sheep` case,

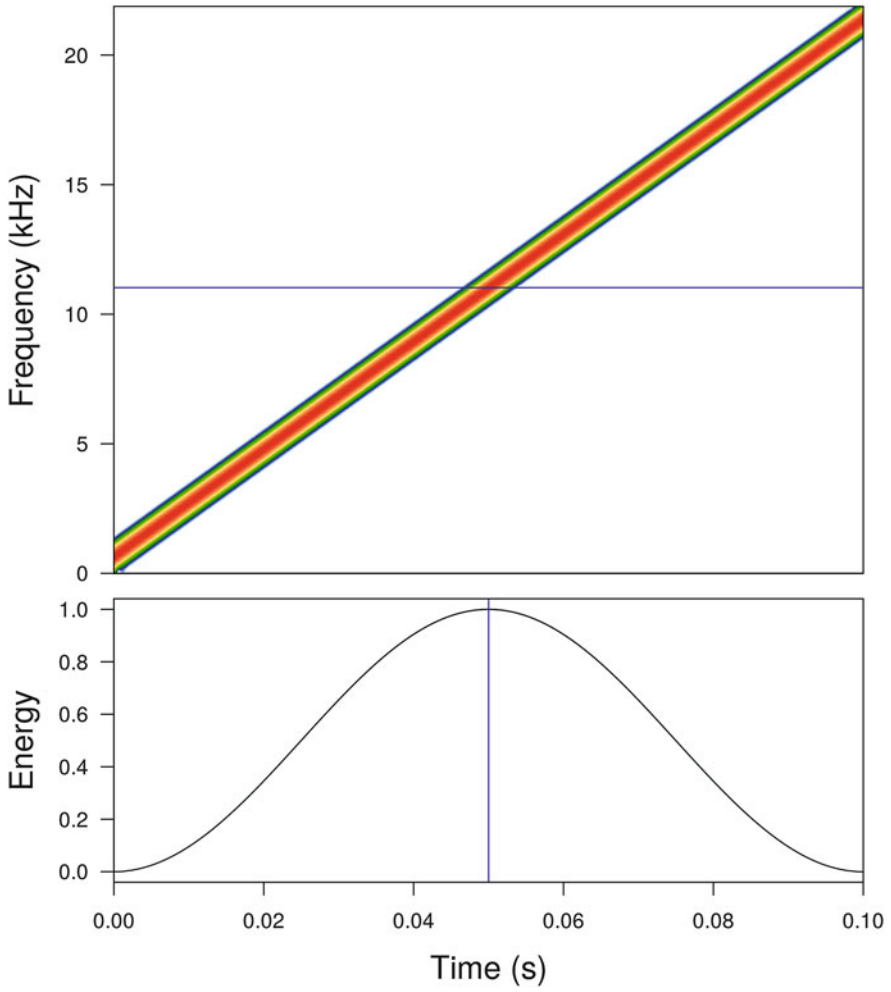


Fig. 13.21 Teager-Kaiser energy operator with high-frequency content. The TKEO does not return appropriate results for frequencies above $f_s \div 4$, as illustrated here with a frequency modulated sound starting at 0 Hz and ending at $f_s \div 2 = 22{,}050$ Hz. The vertical (frequency) or horizontal (vertical) blue line indicates where the TKEO is no more operational. Spectrogram (top) and TKEO (bottom)

we can greatly improve the quality of the results by applying a band-pass filter between 500 and 700 Hz with the function `fir()` which functionality is explained in Sect. 14.6:

```
sheep.f <- fir(sheep, from=500, to=700, output="Wave")
```

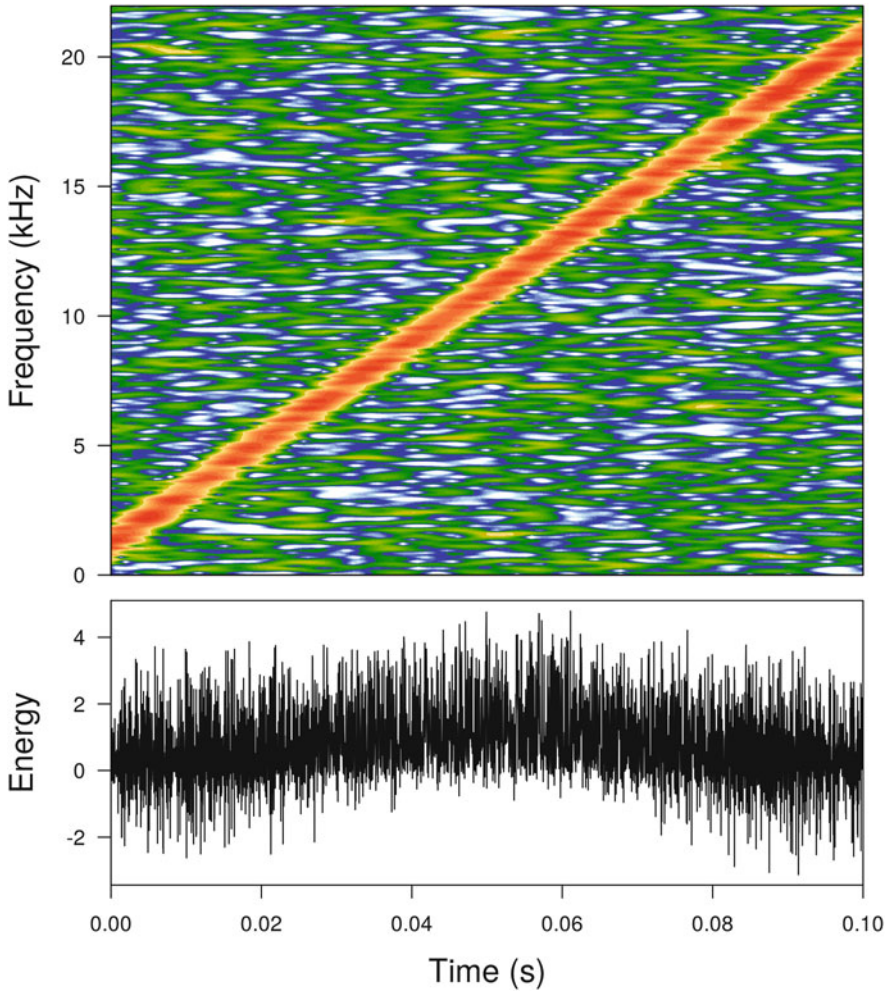


Fig. 13.22 Teager-Kaiser energy operator with noise. The TKEO does not return appropriate results when the system, that is the recording, includes noise as illustrated here with a frequency modulated sound starting at 0 Hz and ending at $f_s \div 2 = 22{,}050$ Hz mixed with white noise. Spectrogram (top) and TKEO (bottom)

The use of TKEO on this filtered signal returns expected results (Fig. 13.23, bottom):

```
par(mfrow=c(3,1))
TKEO(tico, main="tico")
TKEO(sheep, main="sheep")
TKEO(sheep.f, main="sheep with a [500, 700] Hz  band-pass filter")
```

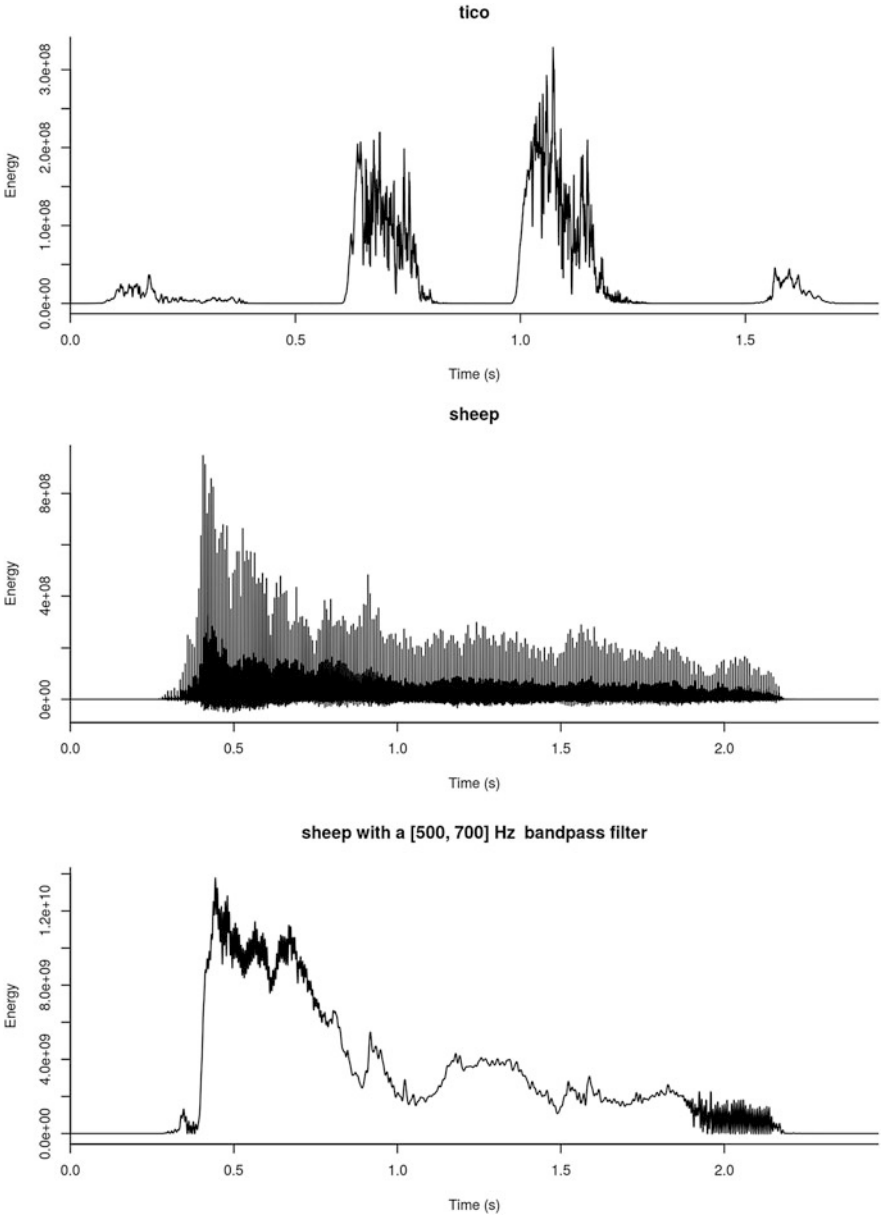


Fig. 13.23 Teager-Kaiser applied on `tico` and `sheep`. The TKEO can be applied directly on `tico` as the conditions of application are met (top). However, the TKEO does not return relevant results if applied on `sheep` that does not meet all conditions of application (middle). A band-pass filter between 500 and 700 Hz can solve the problem by focusing on a single and low-frequency band (bottom)

Chapter 14
Frequency Filters

Audio recordings are far to be perfect, in particular those achieved outdoor where several unwanted sounds may interfere with the sound of interest. It is therefore often necessary to clean up the recordings by removing a part of the frequency spectrum. A section of the frequency content might also be discarded for experimental purposes as in psychoacoustic tests or animal playback experiments where only a part of the signal has to be tested. It is therefore often required to apply a frequency filter on an input signal to produce an output signal with some frequencies removed.

To introduce the functions developed for frequency filtering, we will refer to the vocalizations of two amphibian species. The first species is the South-American dart poison frog *Allobates femoralis* (Fig. 14.1) which male produces during the day a sequence of four frequency-modulated notes. A recording made in French Guiana is available in the file `Allobates_femoralis.wav`:

```
femo <- readWave("sample/Allobates_femoralis.wav")
femo

Wave Object
Number of Samples:      61740
Duration (seconds):     1.4
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```


J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_14

Fig. 14.1 Pictures of soniferous animals: the South-American poison frog *Allobates femoralis* and the European midwife toad *Alytes obstetricans* (Reproduced with the kind permission of pictures by Andrius Pasukonis and Diego Llusia)

The sound includes two sequences of four notes. The recording is of rather good quality, but there is some low-frequency background noise below 500 Hz due to wind, a frequency band at around 3.5 kHz due to distant *A. femoralis* individuals singing in the background, and a frequency band of 8.2 Hz due to an insect stridulation (Fig. 14.2).

The second species is the European midwife toad *Alytes obstetricans* which male produces during the night a delicate, soft, and short call regularly repeated. The file `Alytes_obstetricans.wav` includes three calls or notes with background noise mainly due to wind below 500 Hz and insect stridulation between 9 and 19 kHz (Fig. 14.3):

```
toad <- readWave("sample/Alytes_obstetricans.wav")
toad

Wave Object
Number of samples:      264000
Duration (seconds):     5.5
Sampling rate (hertz):  48000
Channels (mono/stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

A frequency filter is either an electronic device or a digital tool that deletes a specific frequency band. A frequency filter is defined by a transfer function. This transfer function is a frequency function, noted $H(\omega)$ when referring to the angular frequency or $H(f)$ when referring to ordinary frequency, that shows how the filter acts on the input signal to produce an output signal (Fig. 14.4). The transfer function is usually visualized as a Bode magnitude plot which is a *log–log* plot with frequency placed on the *x*-axis and dB relative amplitude on the *y*-axis.

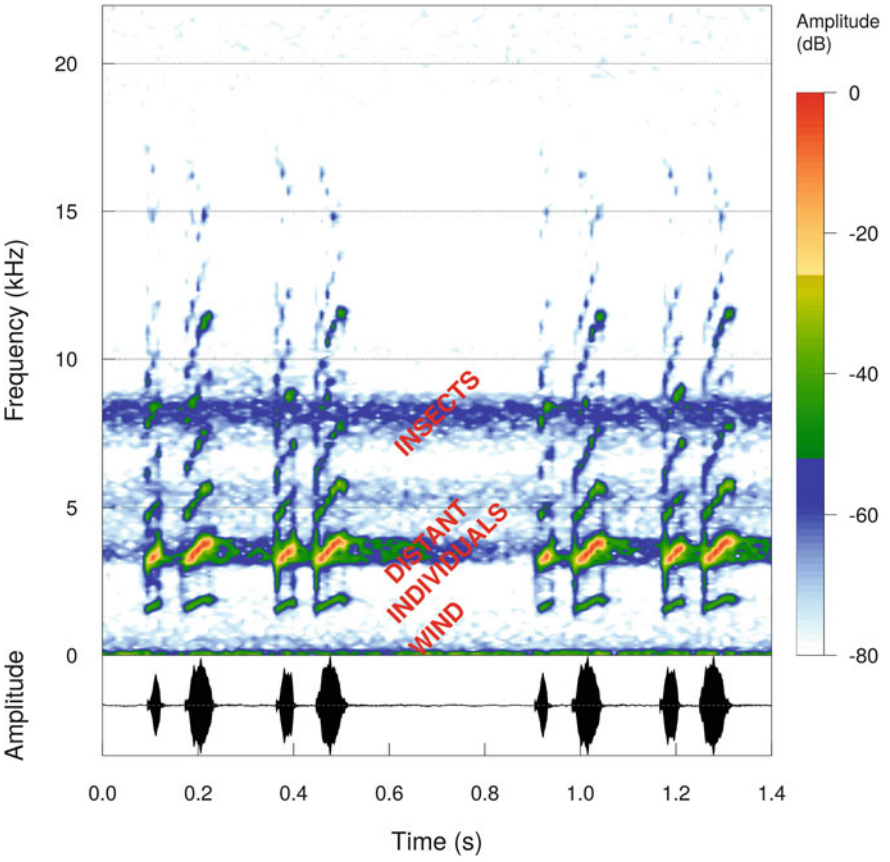


Fig. 14.2 Spectrogram and oscillogram of the vocalization of the dart poison frog *Allobates femoralis*. The recording includes two sequences of four notes and background noise due to wind, distant individuals, and insects. Fourier window size = 512 samples, overlap = 0%, Hanning window

The shape of the transfer function $H(f)$ indicates the frequencies attenuated and unchanged by the filter. There are four main types of frequency filters:

low-pass filter allowing the passage of frequencies lower than a cutoff frequency f_c and attenuating the frequencies higher than f_c.

high-pass filter allowing the passage of frequencies higher than a cutoff frequency f_c and attenuating the frequencies lower than f_c.

band-pass filter allowing the passage of frequencies between a lower cutoff frequency f_l and a upper cutoff frequency f_u and attenuating frequencies lower than f_l and higher than f_u.

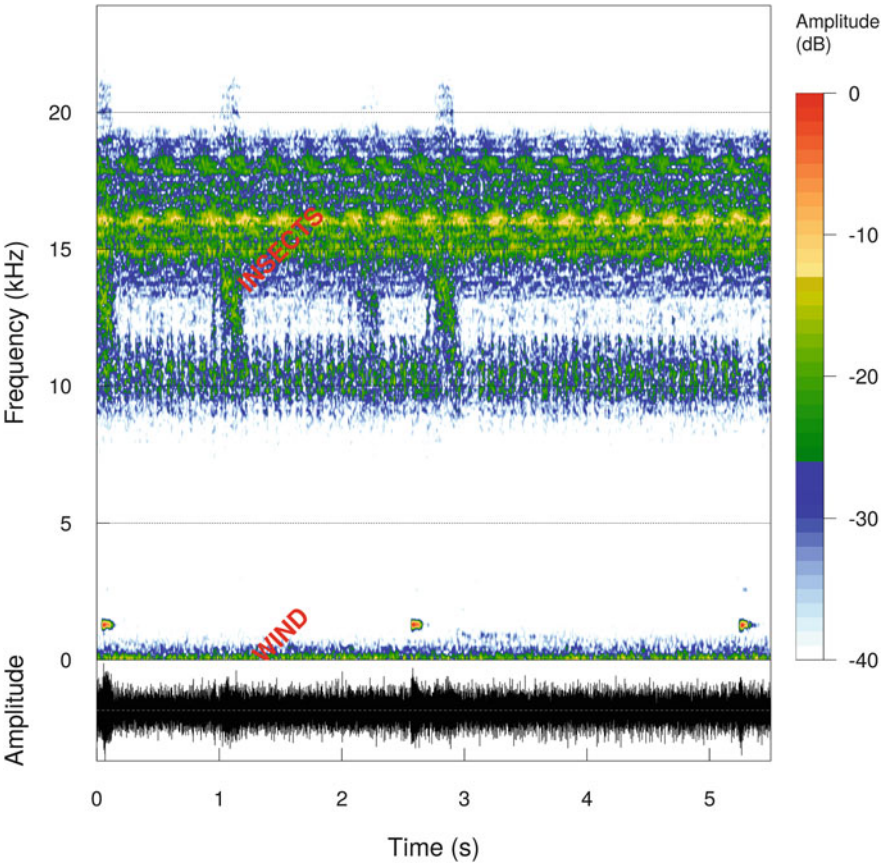


Fig. 14.3 Spectrogram and oscillogram of the vocalization of the European midwife toad *Alytes obstetricans*. The recording includes three notes, wind, and insects. Fourier window size = 512 samples, 0% of overlap, Hanning window

band-stop filter attenuating the passage of frequencies between a lower cutoff frequency f_l and a upper cutoff frequency f_u and allowing the passage of frequencies lower than f_l and higher than f_u.

The attenuation slope(s) of the transfer function can be more or less accentuated. This degree in attenuation is named the roll-off rate or rejection rate. This rate, which estimates how sharp the filter is, is measured in dB per octave or in dB per decade where an octave corresponds to a multiplication of the frequency by 2 and a decade to a multiplication of the frequency by 10.

By definition, a filter operates changes in amplitude as some frequencies are attenuated, and other not but a filter may also introduce a delay in the phase. This delay is frequency-dependent and can then introduce some distortion in the output signal.

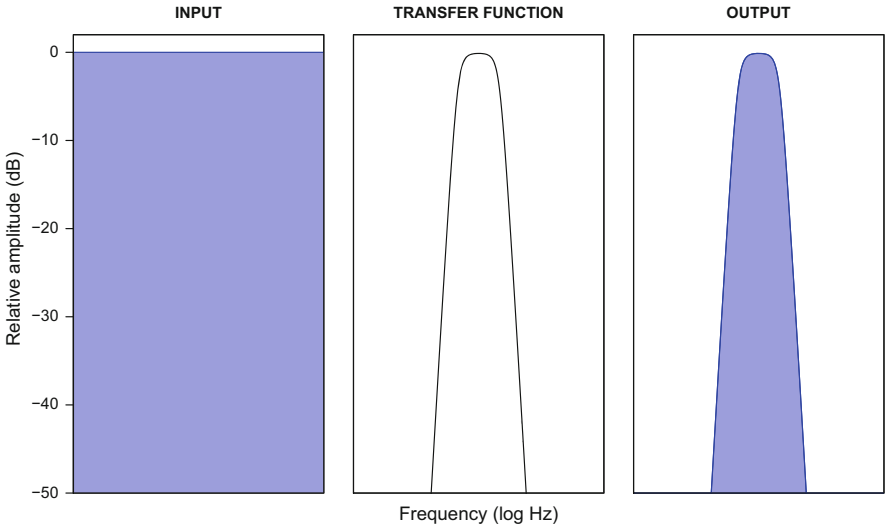


Fig. 14.4 Principle of a frequency filter. The figure sketches how a frequency filter can change the frequency content of a sound. The input sound is a white noise with a flat frequency spectrum (left), the filter is characterized by a transfer function $H(f)$ with a bell-like shape (middle), and the output has a frequency spectrum with a shape similar to the filter transfer function (right). Note that the frequency x-axis follows a logarithmic scale. Inspired from Speaks (1999)

There are seven `seewave` functions to apply a frequency filter as summarized in Table 14.1. These filters will be introduced one after the other one in the next sections.

Table 14.1 Types of frequency filters: short description of the frequency filters found in `seewave`, sorted by alphabetic order

Name	Function	Description	Main use
Butterworth filter	`bwfilter()`	Polynomial coefficients	Enhancement
Comb filter	`combfilter()`	Time derivative	Voice enhancement
DFT/STDFT filter	`ffilter()`	Frequency multiplication	Cleaning
Finite impulse response filter	`fir()`	Time convolution	Cleaning
Preemphasis filter	`preemphasis()`	Time derivative	Voice enhancement
Smoothing spline filter	`rmnoise()`	Cubic smoothing spline	High-frequency cleaning
Smoothing sum filter	`smoothw()`	Sum sliding window	High-frequency cleaning

14.1 Preemphasis Filter

A preemphasis filter is a high-pass filter often used in speech analysis to reinforce the high-frequency part naturally reduced by the voice-system resonators. The preemphasis filter is the first step of the computation of mel-frequency cepstral coefficients (see Sect. 12.1.1). The signal filtered is obtained by a simple time derivative operation which removes through a weighted subtraction small amplitude changes between adjacent samples due to low-frequency components:

$$s_{filtered}[n] = s[n] - \alpha s[n-1]$$

where $s[n]$ is the original signal and α is a constant determining the cutoff frequency f_c of the filter. Increasing α increases the cutoff frequency, hence the attenuation of low frequencies.

The constant α varies usually between 0.9 and 1, with a standard value of 0.97 for speech signals. α can be determined in relation with f_c using the following equation:

$$\alpha = e^{-\frac{2\pi f_c}{f_s}}$$

where f_s is the sampling frequency.

The transfer function of the filter is obtained with (Fig. 14.5):

$$H[f] = 1 + \alpha^2 - 2\alpha \cos\left(\frac{2\pi f}{f_s}\right)$$

A preemphasis filter is available in the `seewave` function `preemphasis()`.[1] To test this function, we first load the data `hello` introduced in Chap. 11:

```
hello <- readWave("sample/hello.wav")
```

We then estimate the time constant α for the cutting frequency $f_c = 150$ Hz:

```
f <- hello@samp.rate
fc <- 150
alpha <- exp(-2*pi*fc/f)
alpha
[1] 0.9805566
```

[1] A preemphasis filter can also be built with the function `filter()` of the package `signal` as implemented in the function `melfcc()`: `filter(x, filter = c(1, -alpha), method = "convolution," sides = 1, circular = FALSE)`.

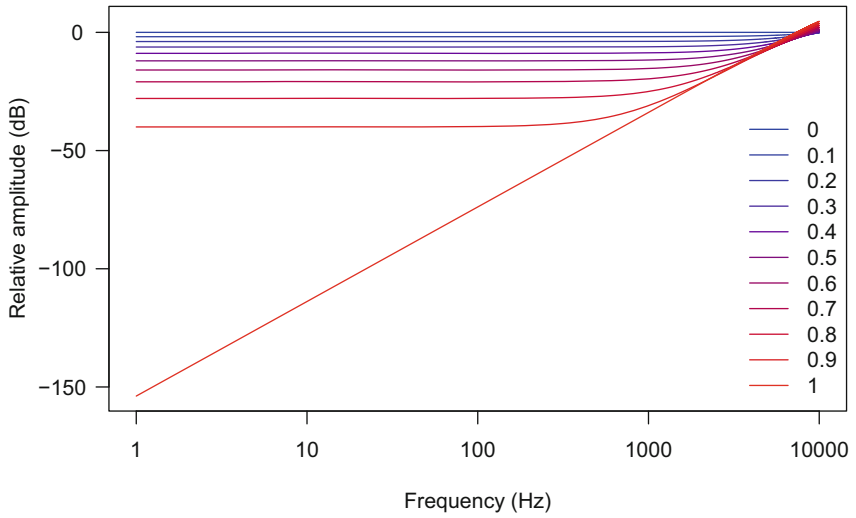


Fig. 14.5 Transfer function of preemphasis filter. The figure shows the Bode plot of the transfer function of preemphasis filters with values of α varying between 0 and 1

We apply the preemphasis filter and save the result in a new `Wave` object with:

```
hello.filt <- preemphasis(hello, alpha=alpha,
                          output="Wave", plot=TRUE)
```

Choosing `plot=TRUE` produces a graphical display where the spectrogram of the new signal and the frequency response of the filter are plotted side-by-side (Fig. 14.6). The result of the filter with different values of α could also be viewed by overlaying the mean spectra of the original ($\alpha = 0$) and filtered signals ($\alpha > 0$) as illustrated in the following code (Fig. 14.7):

```
# alpha values
alpha <- seq(0, 1, by=0.1)
n <- length(alpha)
# mean spectra parameters
wl <- 1024
ovlp <- 87.5
palette <- colorRampPalette(c("blue", "red"))(length(alpha))
# mean spectrum of original signal
meanspec(hello, wl=wl, ovlp=ovlp, dB="max0")
# loop for mean spectra of filtered signals
for(i in 1:n){
```

(continued)

```
    lines(
        meanspec(
            preemphasis(hello, alpha=alpha[i], output="Wave"),
                wl=wl, ovlp=ovlp, dB="max0", plot=FALSE),
                  col=palette[i]
    )
}
# legend
legend("topright",
       legend=alpha, title=expression(alpha), ncol=2,
       col=palette[1:n], lty=1, bty="n")
```

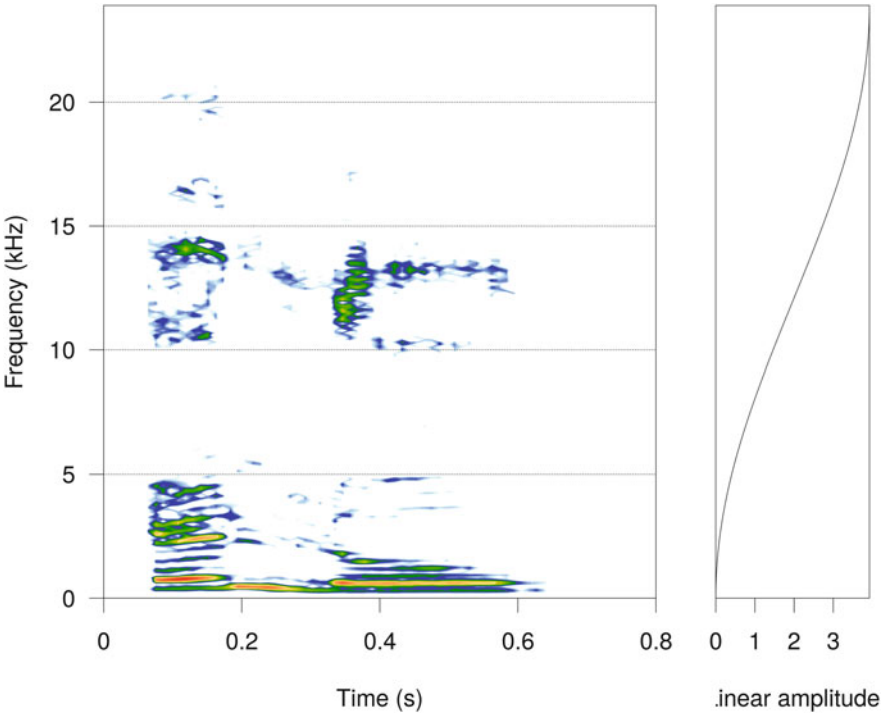


Fig. 14.6 Example of a preemphasis filter. Graphical display of the `seewave` function `preemphasis()` showing side-by-side the spectrogram of the filtered signal, here `hello`, and the frequency response of the filter along a linear amplitude scale

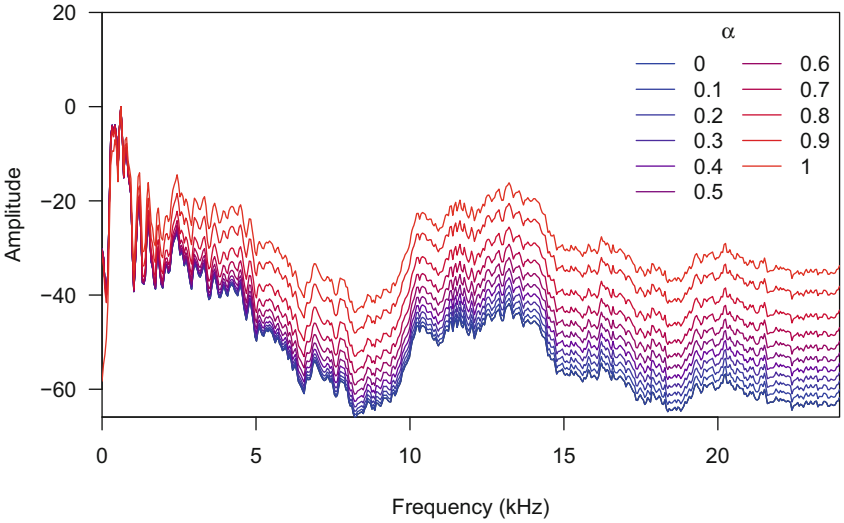


Fig. 14.7 Effect of varying the α time constant of the preemphasis filter. The mean spectra of the original signal ($\alpha = 0$) and filtered signals ($\alpha = \{0.1, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$) using the `seewave` function `preemphasis` are plotted on the same graph. This illustrates how much high-frequency content is enhanced depending on the value of α. Mean spectra parameters: Hanning window, 1024 samples, 87.5% of overlap, no zero-padding

14.2 Comb Filter

A comb filter is a frequency filter which transfer function shows a series of periodic peaks and notches that can remind the shape of a comb. Such filter is quite close to the preemphasis filter (see Sect. 14.1) and is similarly used for speech enhancement. A comb filter is also a particular type of FIR filter (see Sect. 14.6). A comb filter consists in adding a delayed version of a signal to itself resulting in constructive and destructive interference. A feedforward version of a comb filter can be written following:

$$s_{filtered}[n] = s[n] - \alpha s(n - K)$$

where $s[n]$ is the original signal, α is the scaling factor, and K is the delay length. The periodic transfer function is obtained with:

$$H[f] = \sqrt{1 + \alpha^2 + 2\cos\left(\frac{\pi f K}{f_s}\right)}$$

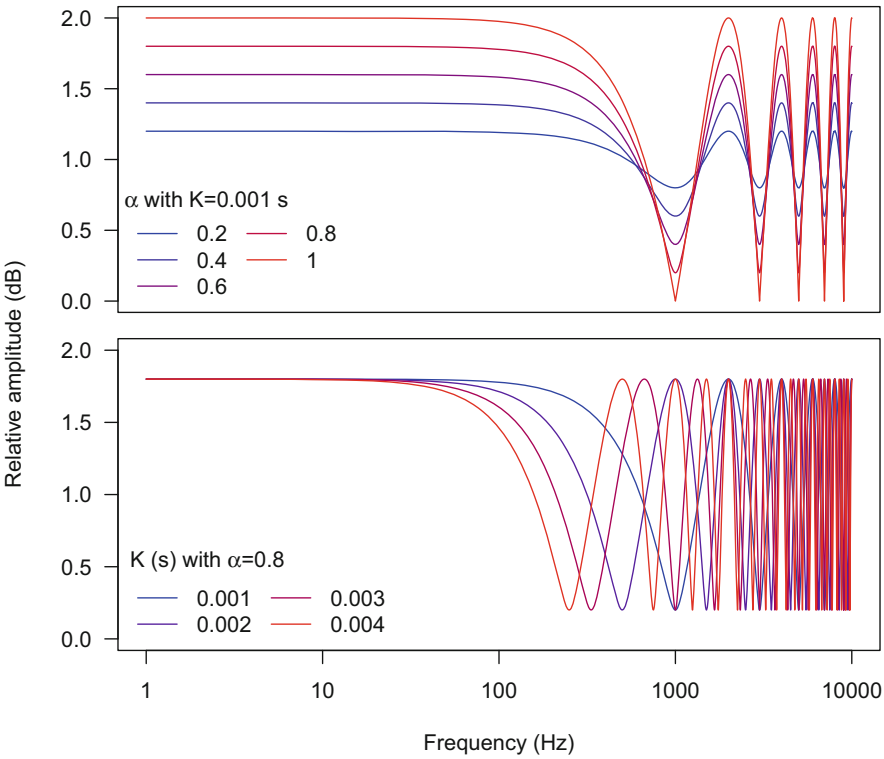


Fig. 14.8 Transfer function of comb filter. The top graphic shows the transfer function H of five comb filters differing in α but not in K ($K = 0.001$). The sharpness of the peak increases with α. The bottom graphic shows the transfer function H of four comb filters differing in K but not in α. The number and position of peaks changes with K

The parameter K controls the number and position of peaks (respectively notches), the peaks being found at $\left\{\frac{2}{K}, \frac{4}{K}, \frac{6}{K}, \ldots\right\}$ and the notches being found at $\left\{\frac{1}{2K}, \frac{3}{2K}, \frac{5}{2K}, \ldots\right\}$ when $\alpha > 0$. Increasing the value α toward 1 increases the sharpness of the peaks (Fig. 14.8).

A comb filter is implemented in the `seewave` function `combfilter()`, the parameters α and K being controlled with the arguments `alpha` and `K`, respectively. Here we apply a comb filter on the dataset `hello` with $\alpha = 0.9$ and $K = 0.001$ s (Fig. 14.9):

```
hello.filt <- combfilter(hello, alpha=0.9, K=0.001, units="seconds",
                         plot=TRUE, output="Wave")
```

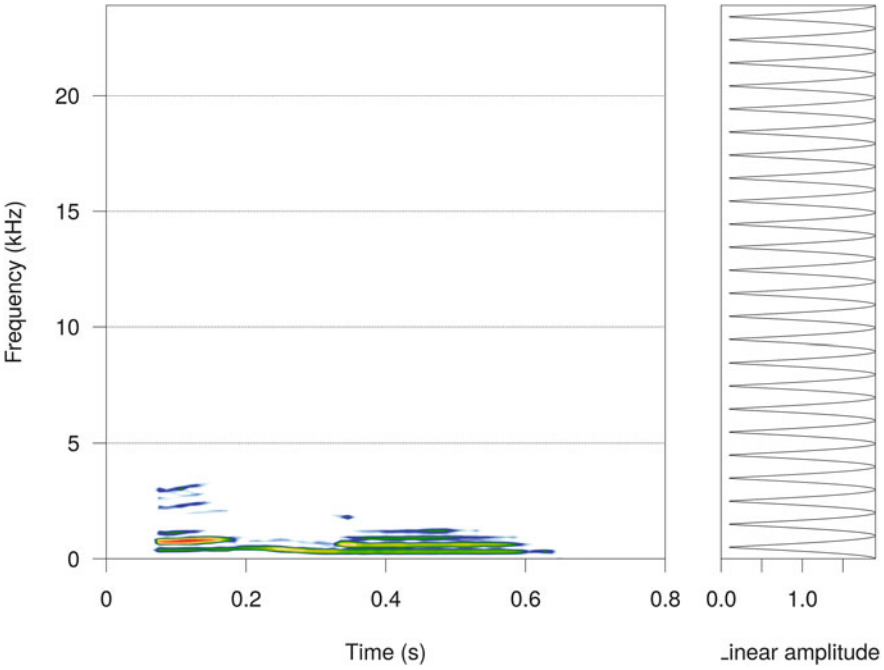


Fig. 14.9 Example of a comb filter. Graphical display of the `seewave` function `comb` showing side-by-side the spectrogram of the filtered signal, here `hello`, and the frequency response of the filter along a linear amplitude scale

14.3 Butterworth Filter

Butterworth filter is a popular frequency filter named after its creator Stephen Butterworth (1885–1958), a British physicist (Butterworth 1930). The main properties of this filter is a roll-off rate of 6 dB per octave, or 20 dB per decade, for a first-order filter. A nth-order filter has a roll-off rate of $n6$ dB per octave, such that an 8th-order filter produces an attenuation effect of $8 \times 6 = 48$ dB per octave. The transfer function being asymmetric, a high-pass and low-pass with the same cutoff frequency do not attenuate in a similar way.

The general transfer function of a Butterworth filter is:

$$H(\omega) = \frac{1}{\sqrt{1 + x^{2n}}}$$

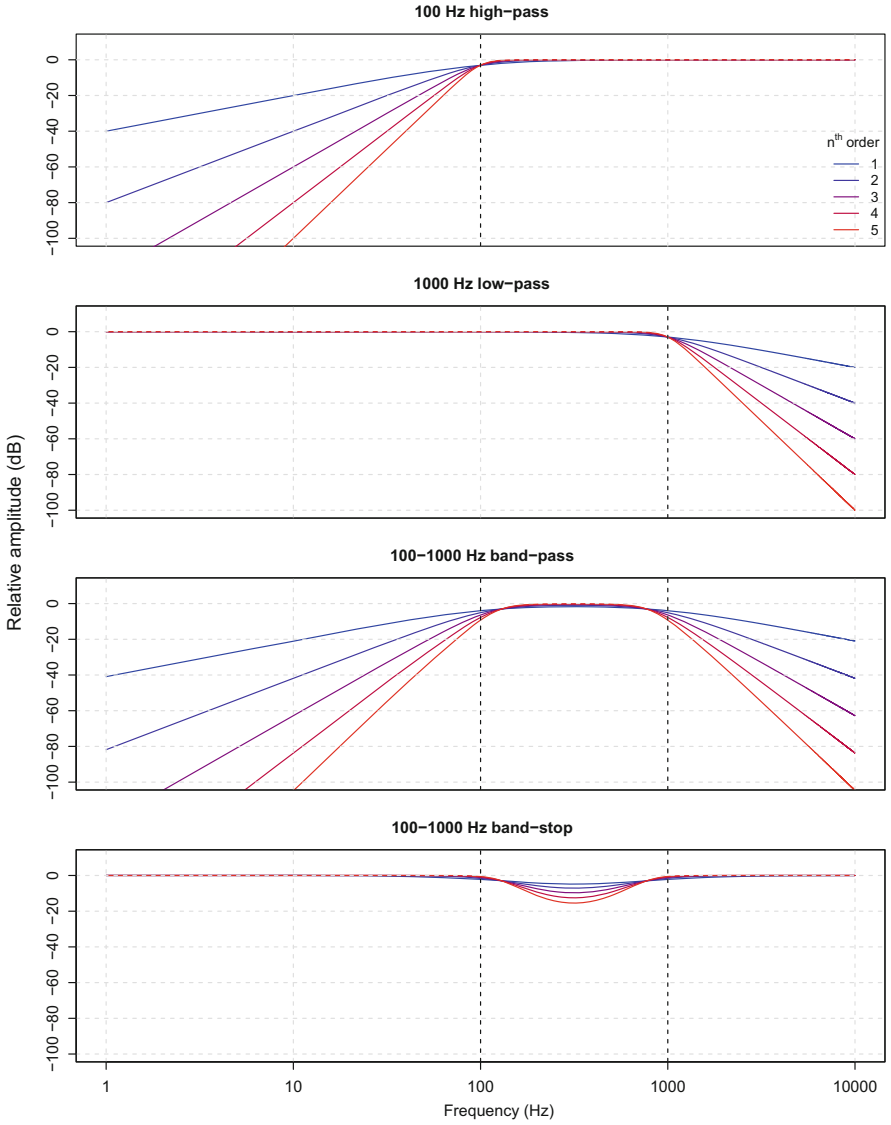


Fig. 14.10 Transfer function of Butterworth filter. The figure shows the Bode plot of the transfer function of a 100 Hz high-pass, a 1000 Hz low-pass, a 100–1000 Hz band-pass, and a 100–1000 Hz band-stop of a 1–5th Butterworth filter. The vertical black dashed-line show the cutoff frequency(ies) and the gray grid underlines the −20 dB roll-off per decade

with:

$$x = \begin{cases} \frac{\omega}{\omega_c} & \text{for a low-pass filter} \\ \frac{\omega_c}{\omega} & \text{for a high-pass filter} \\ \frac{\omega^2+\omega_l\omega_u}{\omega(\omega_u-\omega_l)} & \text{for a band-pass filter} \\ \frac{\omega(\omega_u-\omega_l)}{\omega^2+\omega_l\omega_u} & \text{for a band-stop filter} \end{cases}$$

where $\omega = f \div 2\pi$ are the angular frequencies to be transformed by the transfer function $H(\omega)$, $\omega_c = f_c \div 2\pi$ is the angular cutoff frequency for a low-pass or high-pass filter, and $\omega_l = f_l \div 2\pi$ and $\omega_u = f_u \div 2\pi$ are the lower and upper angular cutoff frequencies for a band-pass or band-stop filter. Examples of amplitude Bode plots are given in Fig. 14.10 and in DIY box 14.1.

DIY 14.1 — How to produce the Bode plot of a Butterworth low-pass or high-pass filter

We have seen that the transfer function $H(\omega)$ or $H(f)$ of a low-pass or high-pass Butterworth filter or order n can be written with two simple equations (see Sect. 14.3). We can write a function, named `butter.H()`, which parameters are the frequencies f for which we want to compute the transfer function, the cutoff frequency f_c, the order of the filter n, and the type of filter, either a low-pass or a high-pass. We first check this type argument with the function `match.arg()`, we then compute $H(f)$ for a low-pass and $H(f)$ for a high-pass only if the condition `type=="high"` is true:

```
butter.H <- function(freq, fc, n, type=c("low", "high"))
{
    type <- match.arg(type)
    if(type=="low") s <- freq/fc
    if(type=="high") s <- fc/freq
    H <- 1/sqrt(1+s^(2*n))
    H <- 20*log10(H)
    return(H)
}
```

We can try the function for frequencies between 1 and 44,100 Hz with a high-pass filter of second order at a cutoff frequency of 2500 Hz:

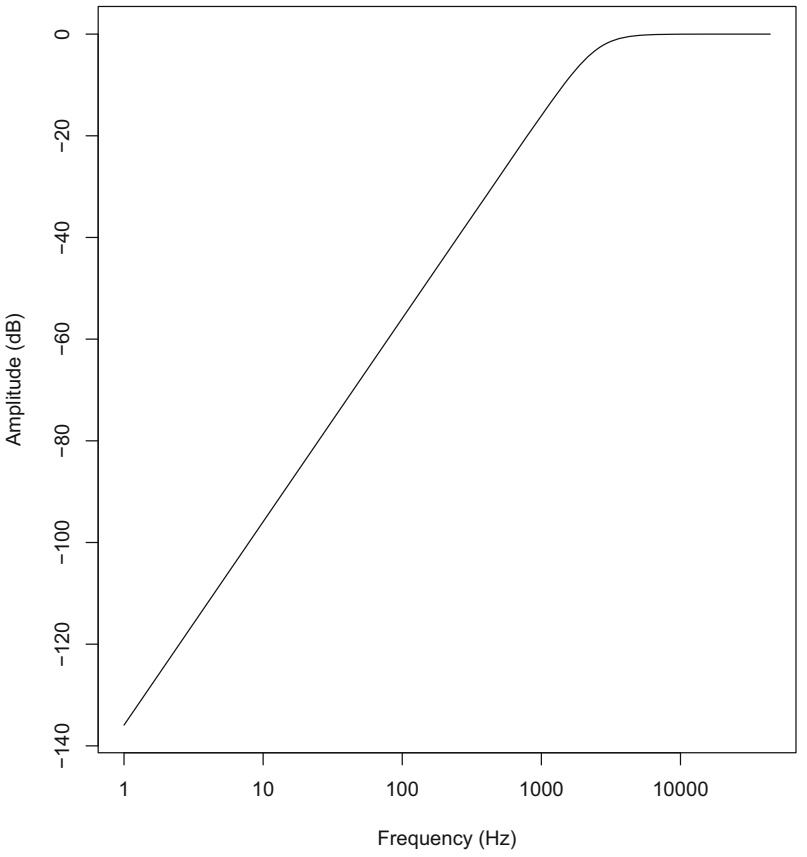
```
freq <- seq(1, 44100, length.out=1000)
H <- butter.H(freq=freq, fc=2500, n=2, type="high")
```

(continued)

DIY 14.1 (continued)

It is now easy to create an amplitude Bode plot with:

```
plot(x=freq, y=H, type="l", log="x",
     xlab="Frequency (Hz)", ylab="Amplitude (dB)")
```



The `seewave` function to apply a Butterworth filter is `bwfilter()`. This function is a rather simple wrapper of the two functions `butter()` and `filtfilt()` of the package `signal`. This later function does a correction of the phase so that distortion is limited. It is first necessary to load `signal`:

```
library(signal)
```

Then, the main arguments of the function `bwfilter()` are:

`n` the order of the filter,
`from` lower cutoff frequency f_l expressed in Hz,
`to` upper cutoff frequency f_u expressed in Hz,
`bandpass` type of filter, either band-pass (if `TRUE`, default) or band-stop (if `FALSE`).

The different types of filters are obtained with the following combinations of these arguments, here with a 3rd-order filter:

```
# low-pass with a 5000 Hz cutoff frequency
res <- bwfilter(femo, n=3, to=5000, output="Wave")
# high-pass with a 5000 Hz cutoff frequency
res <- bwfilter(femo, n=3, from=5000, output="Wave")
# band-pass between 3000 and 5000 Hz
res <- bwfilter(femo, n=3, from=3000, to=5000, output="Wave")
# band-stop from 3000 Hz to 5000 Hz
res <- bwfilter(femo, n=3, from=3000, to=5000,
                bandpass=FALSE, output="Wave")
```

14.4 Wave Smoothing Filter

A solution to remove high-frequency noise is to smooth the time wave. This can be achieved by using a sum sliding window as discussed in Sect. 5.2.3.3. Such operation is available in the `seewave` function `smoothw()`. Its usage is rather straightforward as the main parameter, `wl`, is the window length in number of samples over which the sum is operated. The operation can be repeated so that a nth-order filter can be obtained. The following example uses `smoothw()` twice on `femo` (Fig. 14.11):

```
# first order
res1 <- smoothw(femo, wl=4, output="Wave")
# second order
res2 <- smoothw(res1, wl=4, output="Wave")
```

The function `rmnoise()` also attempts to remove noise through smoothing but this time using cubic smoothing spline as implemented in the function

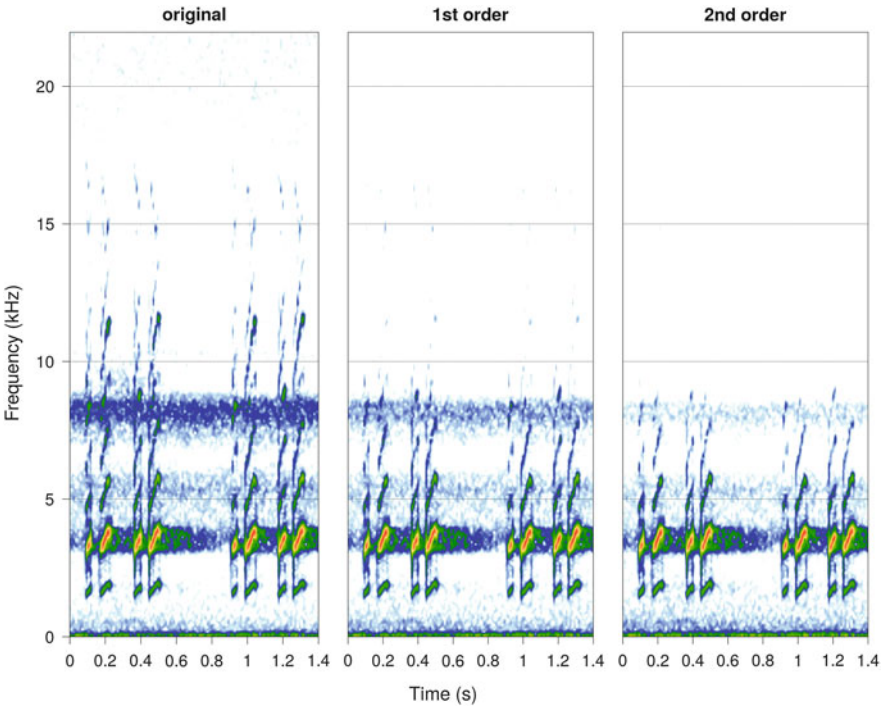


Fig. 14.11 Filter through wave smoothing with `smoothw()`. The original `femo` recording (left) is passed through a wave smoothing a first time (middle) and a second time (right)

`smooth.spline()` of the package `stats`. The parameter that controls the degree of smoothing is the argument `spar` which is typically in (0, 1]. Here is a test with two values of `spar` (Fig. 14.12):

```
res1 <- rmnoise(femo, output="Wave", spar=0.4)
res2 <- rmnoise(femo, output="Wave", spar=0.6)
```

In both cases, the frequency band around 8.2 kHz due to distant insects is reduced. However, such filter may introduce distortion like the uncontrolled amplification of low frequencies. It should be then used carefully and the results should always be visualized with either a spectrogram or a frequency spectrum display.

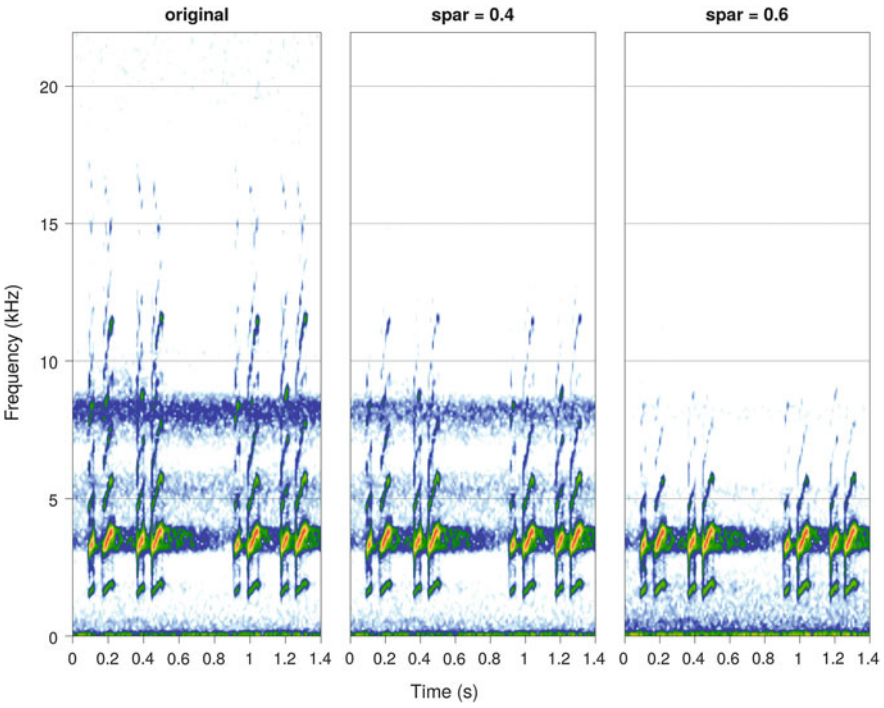


Fig. 14.12 Filter through wave smoothing with `rmnoise()`. The original `femo` recording (left) is passed through a cubic smoothing spline with a smoothing parameter `spar=0.4` (middle) and `spar=0.6` (right)

14.5 DFT and STDFT Filter

14.5.1 Principle

Another way to filter out unwanted frequencies, or to select frequencies of interest, is to use the Fourier transform (FT, DFT). The frequency spectrum $F(f)$ of the signal $s(t)$ is first obtained using the DFT. This frequency spectrum is then multiplied by the frequency transfer function of the filter $H(f)$ so that the frequency domain of the signal is transformed. The filtered signal $s_{filtered}(t)$ is then obtained by using the inverse Fourier transform (IFT, IDFT). This process can be windowed using the short-term Fourier transform (STDFT) and its inverse form (ISTDFT) (Fig. 14.13).

14.5.2 `ffilter()` Function

The `seewave` function `ffilter()` operates a DFT filter by (1) computing the STDFT as `spectro()` does, (2) applying a filter transfer function defined

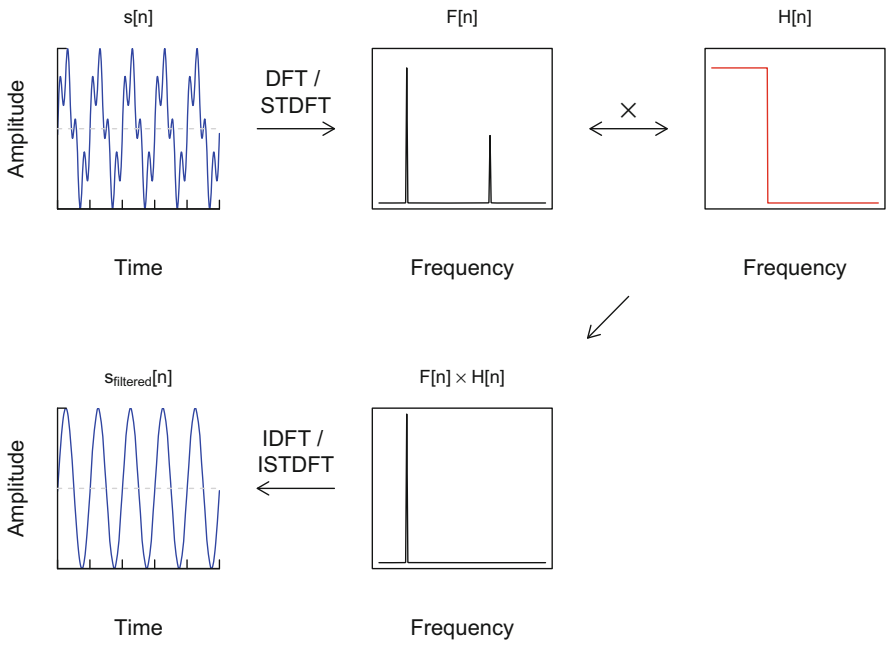


Fig. 14.13 Principle of DFT filter. A DFT filter is based on a return travel between the time and frequency domains: the frequency signal spectrum $F[n]$ of the original signal $s[n]$ is multiplied by the transfer function of the filter $H[n]$, here a low-pass filter, and the filtered signal is obtained through the inverse Fourier transform. Each function is made of n samples

in the arguments, and (3) using the function `istft()` to construct the filtered signal. In addition to the traditional arguments `wl`, `ovlp` and `wn` linked to STDFT computation, the function has the following key arguments:

`from` lower cutoff frequency f_l expressed in Hz,
`to` upper cutoff frequency f_u expressed in Hz,
`bandpass` type of filter, either band-pass (if `TRUE`, default) or band-stop (if `FALSE`)
`custom` optionally a numeric vector of length $wl \div 2$ that set the frequency function transfer $H(f)$ of the filter. The vector can be manually built or designed with the values obtained with `spec()` or `meanspec()`.

14.5.3 Examples

The use of `ffilter()` is in all points similar to the use of the function `fir()` so that the examples given in Sect. 14.6.3 can be then directly translated by replacing `fir()` by `ffilter()`.

A DFT filter can be windowed to give birth to a STDFT filter. Some fancy filters can be designed according to frequency and time. As mentioned above, `ffilter()` calls in background the function `istft()` which processes the ISTDFT. However, this transform is possible only if we provide the matrix of the Fourier coefficients of the STDFT as complex numbers. This can be obtained by extracting the item `$amp` of the value of `spectro()` having specified the arguments `norm=FALSE`, `dB=NULL`, and `complex=TRUE` (see Sect. 11.7.1.4). A dummy example on `femo` shows how to recover the original sound taking care to specify exactly the same parameters (`wl`, `ovlp`, `wn`) for the STDFT (function `spectro()`) and the ISTDFT (function `istft()`):

```
f <- femo@samp.rate ; wl <- 512; ovlp <- 75; wn <- "hanning"
data <- spectro(femo, wl=wl, ovlp=ovlp, wn=wn,
                plot=FALSE, norm=FALSE, dB=NULL, complex=TRUE)
res <- istft(data$amp, wl=wl, ovlp=ovlp, wn=wn,
             f=f, output="Wave")
```

The following line produces the same spectrogram displayed in Fig. 14.2:

```
spectro(res, collevels=seq(-80,0,1), osc=TRUE)
```

We can now try to modify the object `data` to change the time $\times$ frequency content of `femo`. We can, for instance, try to remove the first harmonic of the fourth note. The tricky thing is to identify the time and frequency limits of this harmonic in the matrix `data$amp`. The first step consists in using `locator()` after having displayed the spectrogram with `spectro()` and in identifying the limits with the mouse cursor (see Sect. 11.8.1). We obtained the following limits: 0.44 and 0.51 s and 2.77 and 4.21 kHz. Which columns and rows of `data$amp` do correspond to these values? The time and frequency values of the STDFT matrix are stored in the list items `data$time` and `data$freq`, respectively. The next step is to find the closest values in these vectors to the limits determined previously. This can be achieved by localizing the value $\min(|x - limit|)$:

```
tmin <- which.min(abs(data$time-0.44))
tmax <- which.min(abs(data$time-0.51))
fmin <- which.min(abs(data$freq-2.77))
fmax <- which.min(abs(data$freq-4.21))
```

As we know where the first harmonic of the fourth note is in the `data$amp` matrix, we can replace the Fourier coefficients by 0 values, use the `istft()`

function, and plot the result with `spectro()` so that we apply a band-stop filter (Fig. 14.14, top):

```
data1 <- data$amp
data1[fmin:fmax, tmin:tmax] <- 0
res1 <- istft(data1, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

Alternatively, we can apply a band-pass filter by replacing all the Fourier coefficients except those of the harmonic by 0 values (Fig. 14.14, bottom):

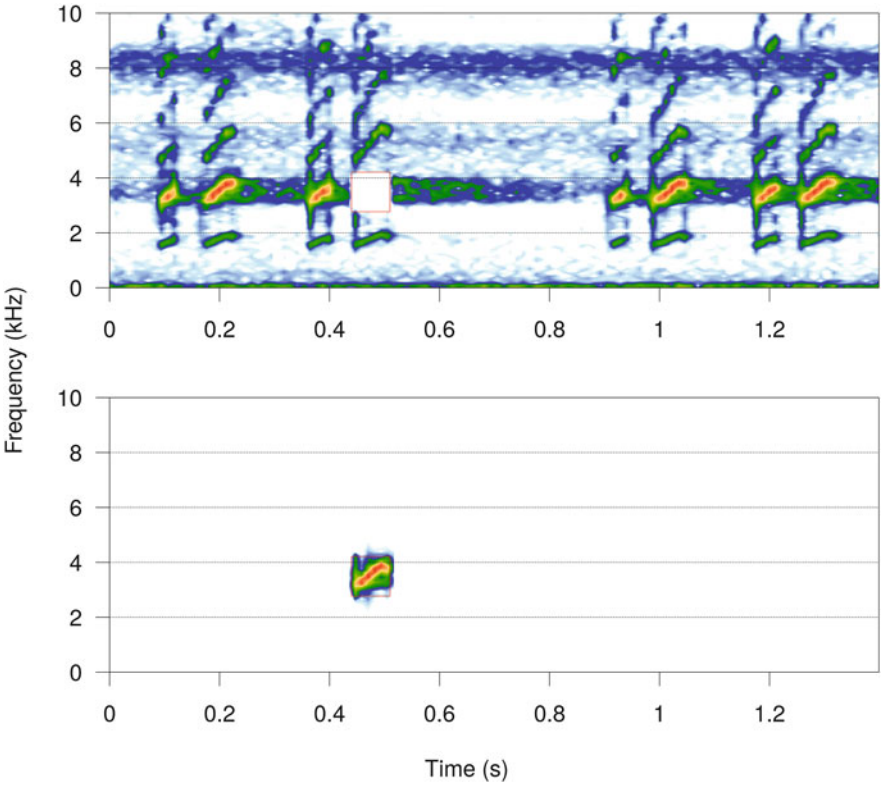


Fig. 14.14 Example of STDFT filter. Two examples of DFT filter based on the function `istft()`. The second harmonic of the first harmonic of the fourth note of `femo` was removed with a band-stop filter (top) or selected with band-pass filter (bottom). The red square was added using the low-level plot function `rect()`

```
data2 <- data$amp
data2[-(fmin:fmax), -(tmin:tmax)] <- 0
res2 <- istft(data2, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

These modifications through the STDFT matrix open the possibility of other changes detailed in Sect. 15.4.

14.6 FIR Filter

14.6.1 Principle

We have seen that the DFT/STDFT filter is partly based on a multiplication between the frequency spectrum of the sound to be filtered and a transfer function expressed in the frequency domain $H(f)$. This multiplication constrains to compute the DFT (or STDFT) and the IDFT (or ISTDFT) to travel between the time and frequency domains. However, a multiplication in the frequency domain is equivalent to a convolution in the time domain. A convolution is a mathematical operation quite similar to cross-correlation (see Sect. 17.1) which returns the amount of overlap of one function as it is shifted over another function. The idea of the finite impulse response (FIR) filter is to use convolution between the original signal and the transfer function to stay in the time domain. Doing so, we avoid the use of the Fourier transforms for a more efficient implementation. The only thing we have to do is to pass the transfer function $H(f)$ from the frequency domain to the time domain. This can be achieved easily with the IDFT.

To summarize, the filtered signal $s_{filtered}[n]$ is obtained by the convolution of the original signal $s[n]$ with a finite transfer function expressed in the time domain $H[n]$ of length $2n + 1$ (Fig. 14.15):

$$s_{filtered}[n] = s[n] * H[n]$$

where $*$ is the convolution operation. This can also be written as:

$$s_{filtered}[n] = \sum_{m=-M}^{M} s[m] \times H[n - m]$$

14.6.2 `fir()` Function

The `fir()` function from `seewave` can be employed to apply a FIR filter. The main arguments of `fir()` are:

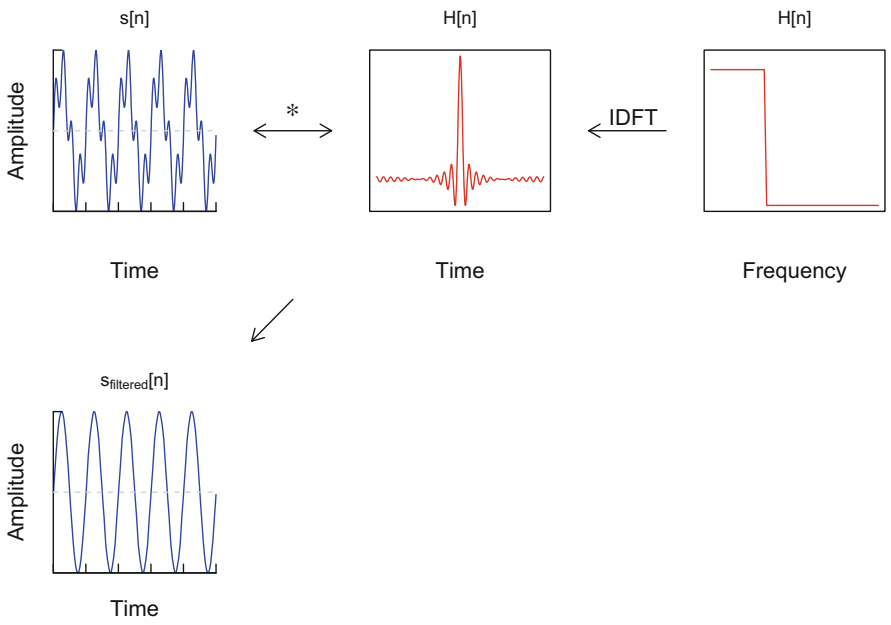


Fig. 14.15 Principle of FIR filter. A FIR filter is based on a convolution ($*$ sign) between the original signal $s[n]$ and the transfer function of the filter $H[n]$ expressed in the time domain. This latter can be obtained from the transfer function in the frequency domain using the inverse of the Fourier transform (IDFT)

`from` lower cutoff frequency f_l expressed in Hz,
`to` upper cutoff frequency f_u expressed in Hz,
`bandpass` type of filter, either band-pass (if `TRUE`, default) or band-stop (if `FALSE`),
`custom` optionally a numeric vector of length $wl \div 2$ that sets the frequency function transfer $H(f)$ of the filter. The vector can be manually built or designed with the value obtained with `spec()` or `meanspec()`.

14.6.3 Examples

14.6.3.1 Simple Use

Here are simple calls of `fir()` applied on `femo`:

```
# low-pass with a 5000 Hz cutoff frequency
res <- fir(femo, to=5000, output="Wave")
# high-pass with a 5000 Hz cutoff frequency
```

(continued)

```
res <- fir(femo, from=5000, output="Wave")
# band-pass between 3000 and 5000 Hz
res <- fir(femo, from=3000, to=5000, output="Wave")
# band-stop from 3000 Hz to 5000 Hz
res <- fir(femo, from=3000, to=5000,
           bandpass=FALSE, output="Wave")
```

14.6.3.2 FIR Band-Pass Filter to Increase Automatic Time Detection

We can explore the usefulness of a FIR filter by using the recording `toad` of the European midwife toad *Alytes obstetricans*. The recording is blurred by the noise of wind and by the stridulation of surrounding katydids such that the toad notes cannot be isolated on the oscillogram (Fig. 14.17, top). If we wish to get automatic time measurements for these notes, we can be quite sure that the function `timer()` shall return irrelevant values (see Chap. 8). However, if we have a look at the spectrogram (Fig. 14.3) and the mean spectrum (Fig. 14.16), we can see that the toad occupies a sharp and isolated frequency band framed by wind and insect sounds.

This suggests that applying a band-pass filter between 0.95 and 1.8 kHz should increase the signal-to-noise ratio of the toad vocalizations (Fig. 14.17, bottom):

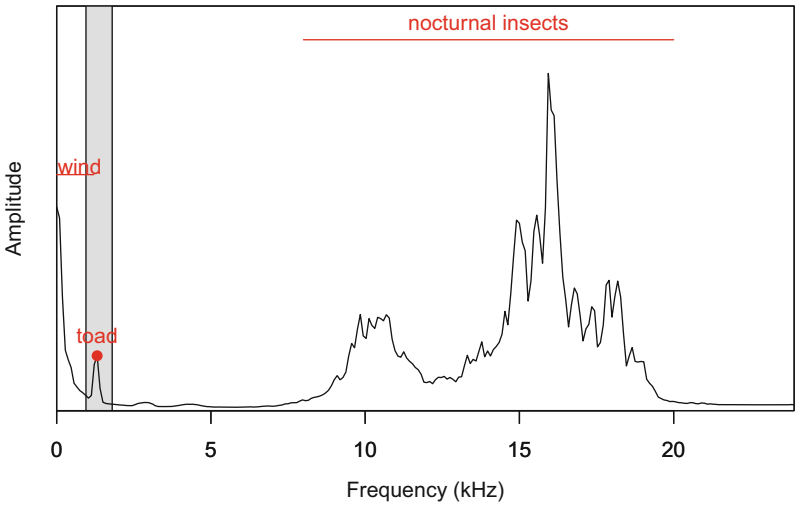


Fig. 14.16 Mean frequency spectrum of `toad`. The recording not only includes vocalizations produced by a male of *Alytes obstetricans* but also wind and nocturnal insect stridulations

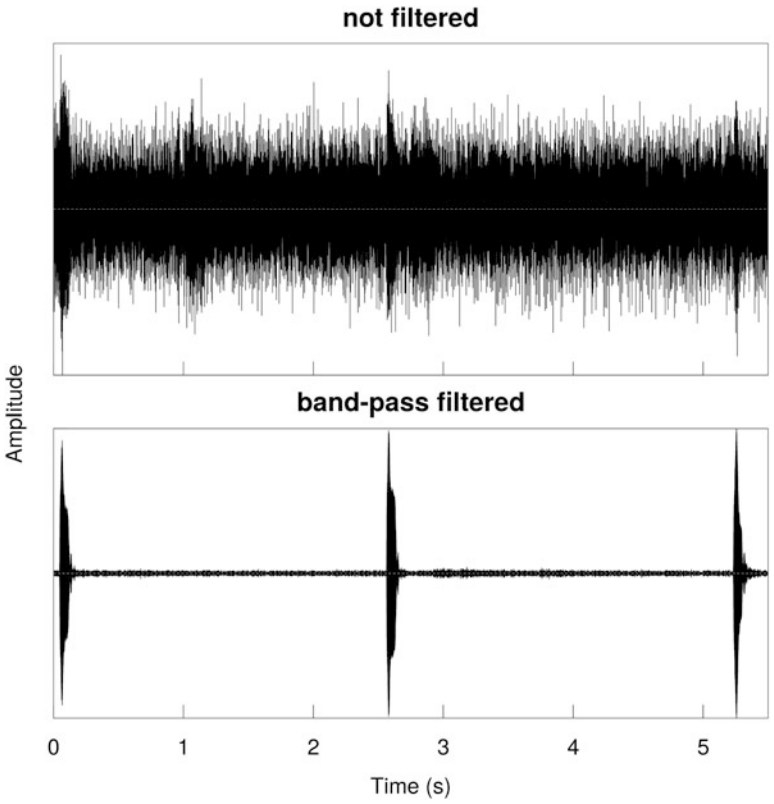


Fig. 14.17 Oscillogram of `toad` before and after FIR filtration. The oscillogram of `toad`, a recording including three vocalizations of *Alytes obstetricans*, wind, and stridulation of nocturnal insects, is shown before (top) and after filtration (bottom)

```
toad.filt <- fir(toad, from=950, to=1800, output="Wave")
```

We can now test `timer()` to measure the note and pause duration taking the precaution to smooth the envelope with the argument `ssmooth()`:

```
res <- timer(toad.filt, envt="hil",
             threshold=4, ssmooth=400, plot=FALSE)
```

We check that `timer()` detected the right number, i.e., 3, of vocalizations:

```
length(res$s)
[1] 3
```

14.6.3.3 FIR Band-Pass Filter to Avoid Aliasing

As explained in Sect. 6.1, undersampling a sound may introduce aliasing artifacts. A solution to avoid the occurrence of unwanted frequencies due to aliasing is to apply and band-pass filter before to resample. Suppose that we had the very bad idea to undersample the recording `peewit` by a factor 4 with the function `resamp()` (see Sect. 6.1):

```
g <- peewit@samp.rate
g
[1] 22050
peewit.u <- resamp(peewit, g=g/4, output="Wave")
```

The undersampling clearly introduced artifact frequencies that are due to frequencies that occur in the original sound above the new Nyquist frequency which is here $g \div 8 = 2756.25$ Hz (Fig. 14.18, left and middle). However, if we take care of applying a FIR low-pass filter with a cutoff frequency corresponding to this new Nyquist frequency then the unwanted frequencies disappear (Fig. 14.18, right):

```
peewit.filtered <- fir(peewit, to=g/8, output="Wave")
peewit.filtered.u <- resamp(peewit.filtered, g=g/4, output="Wave")
```

14.6.4 Setting the Transfer Function

The function `fir()` (and the function `ffilter()` as well) has an argument named `custom` which can be used to give a specific transfer function expressed in the frequency domain as a numeric vector. The values describe the relative amplitude value of the transfer function from 0 to the frequency bin just below the Nyquist frequency, that is, $(f_s \div 2) - (f_s \div wl)$. If we keep on with the `femo` dataset that was sampled at $f_s = 44{,}100$ Hz, then a band-pass filter should be described by 512 values between 0 and $(44{,}100 \div 2) - (44{,}100 - 512) = 21{,}963.87$ Hz. The

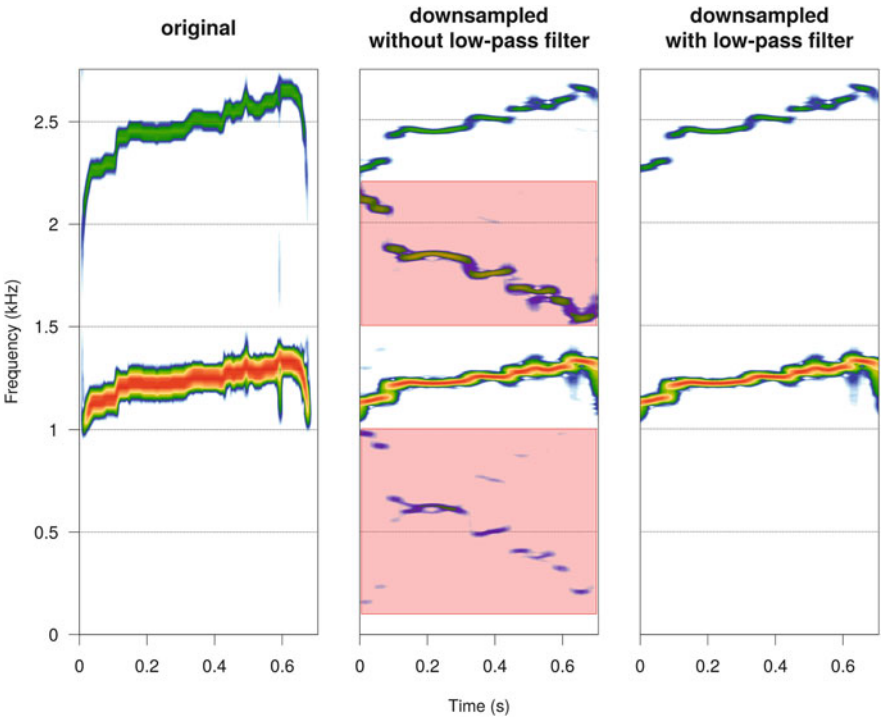


Fig. 14.18 Antialiasing FIR filter. The figure shows the original signal `peewit`, the downsampled and distorted version without any filter process, and the downsampled version with a low-pass FIR filter

following code is a construction of a transfer function for a 3000–5000 Hz band-pass filter. We localize the cutoff frequencies by searching for the minimum absolute difference in the numeric vector containing the frequencies (Fig. 14.19):

```
f <- 44100 ; wl <- 512
freq <- seq(0, f/2-f/wl, length.out=wl)
H1 <- rep(0, wl)
H1[which.min(abs(freq-3000)):which.min(abs(freq-5000))] <- 1
```

The transfer function is then provided to the argument `custom` of `fir()`:

```
res <- fir(femo, wl=wl*2, custom=H1, output="Wave")
```

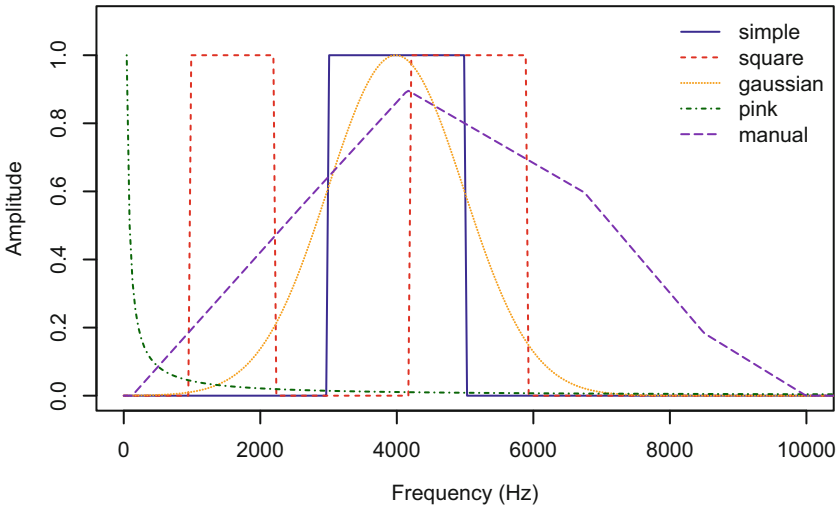


Fig. 14.19 Band-pass frequency transfer functions. Five band-pass transfer functions are displayed on a plot with linear scales. These functions were built manually with basic numeric vectors or with the help of the functions `squarefilter()` and `drawfilter()`

This kind of filter is not very interesting as it is actually equivalent to:

```
res <- fir(femo, from=3000, to=5000, output="Wave")
```

However, we can define any transfer function with 0 and 1 values, as the following one which applies a band-pass filter between 1000 and 2200 and 4200 and 5900 Hz to select the fundamental and the first harmonic only (Fig. 14.19):

```
H2 <- rep(0, wl)
H2[which.min(abs(freq-1000)):which.min(abs(freq-2200))] <- 1
H2[which.min(abs(freq-4200)):which.min(abs(freq-5900))] <- 1
res <- fir(femo, wl=wl*2, custom=H2, output="Wave")
```

Such filter can also be more directly generated with the function `squarefilter()` using the arguments `from` and `to` to set the lower and upper cutoff frequencies in Hz:

```
H2 <- squarefilter(f=f, from=c(1000,4200), to=c(2200,5900))
```

We could also wish to apply a band-pass filter around a specific frequency, say 4000 Hz, with a filter that would have a Gaussian shape. To do that, we can use the density function for the normal distribution `dnorm()`, scale the result between 0 and 1, and then use `fir()` (Fig. 14.19):

```
fc <- 4000
H3 <- dnorm(seq(0, f/2, length=wl), mean=4000, sd=1000)
H3 <- H3/max(H3)
res <- fir(femo, wl=wl*2, custom=H3, output="Wave")
```

Another example of hand-made filter is the generation of pink noise (see Sect. 18.2 for direct synthesis). Pink noise is a noise which frequency spectrum is not flat but favors low frequencies. Such kind of noise better mimics wind or running water noise than white noise that has a flat frequency spectrum. One solution to synthesize pink noise is to first generate white noise and then to apply a frequency filter which transfer function $H(f)$ follows:

$$H(f) \propto \frac{1}{f^{\alpha}}$$

with $\alpha = 1$.

Such transfer function is easy built with the following code which excludes the frequency $f = 0$ because the ratio $1 \div 0$ returns infinity (Fig. 14.19):

```
H4 <- 1/freq[-1]
H4 <- H4/max(H4)
```

The transfer function appears as a regular decrease of amplitude expressed in dB and the logarithm of the frequency with a -3 dB roll-off per octave and 10 dB per decade. The last steps consist in producing a white noise with the function `noisew()` (see Sect. 18.2) and in applying the FIR filter with `fir()`:

```
white <- noisew(d=1, f=44100, output="Wave")
pink <- fir(white, wl=wl*2, custom=c(0,H4), output="Wave")
```

We could also like to manually draw the transfer function $H(f)$. The interactive graphical function `drawfilter()` lets the user the draw either a “continuous”

transfer function or to choose discrete frequencies that will be removed (or kept in the case of a band-pass filter). An empty frequency × amplitude plot is displayed, and the user is invited to choose with the mouse the points that will draw the transfer function (Fig. 14.19):

```
H5 <- drawfilter(f=f, n=wl, continuous=TRUE, discrete=FALSE)
```

The last option could be to recycle the frequency spectrum of another sound as a transfer function. Such operation may be applied when trying to adjust the frequency response of a loudspeaker. In some propagation experiments, a white noise is broadcast in the environment, for instance, a forest, and recorded at specific distances to estimate how sound propagates. However, such experiments rely on a loudspeaker and a microphone with fairly flat frequency response, that is, with an equipment which properties are not frequency dependent. This is rarely the case for low or medium quality loudspeakers used outdoor. A preliminary test is therefore required to determine the frequency response of the loudspeaker and to potentially modify the input to artificially obtain a white noise with a true flat response. Let's walk through an example. Imagine we would like to run such a propagation experiment, we first generate a 1 s noise sampled at $f_s = 44{,}100$ Hz with the function `noisew()` (see Sect. 18.2).

```
noise.in <- noisew(f=44100, d=1, output="Wave")
```

This sound constitutes the input of the system. It is broadcast by the loudspeaker and recorded at a short distance (<1 m) in an anechoic chamber with a microphone that has a flat frequency response. This recording, stored in a `.wav` file named `noise.wav`, is the output of the system:

```
noise.out <- readWave("sample/noise.wav")
```

We compute the mean spectrum of this file to see the frequency profile of the loudspeaker, and we place the result in an object to be used as a transfer function by `fir()`. The argument of `bandpass` of `fir()` is set to `FALSE` so that we apply a band-stop filter as we wish to reduce the importance of the frequency enhanced by the loudspeaker:

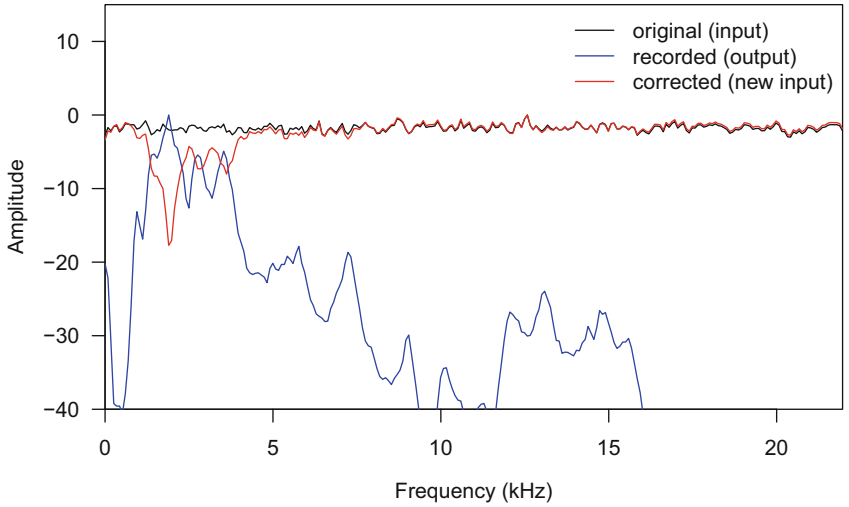


Fig. 14.20 Correction FIR filter for a loudspeaker. Plot of the frequency filter of the original noise (input) given to the loudspeaker, of the noise as recorded after being broadcast by the loudspeaker, and of the noise corrected by the FIR filter

```
H6 <- meanspec(noise.out, plot=FALSE)
noise.cor <- fir(noise.in, custom=H6,
                 bandpass=FALSE, output="Wave")
```

The result is a new object, `noise.cor`, that can be used as new input which should broadcast a noise with a flatter frequency response (Fig. 14.20).

Chapter 15
Other Modifications

We have already seen ways to modify a sound through either edition (see Chap. 6) or filtering (see Chap. 14). Some other changes in amplitude, time, and/or frequency might be required for analysis or to prepare files for broadcast. To go through some functions that change the amplitude, time, and frequency parameters of sound, we will use the song of the bird *Zonotrichia capensis*, the sound of the A-flat tuning fork, the voice of child saying "hello," the vocalizations of the South-American dart poison frog *Allobates femoralis*, and the calling song of the cicada *Cicada orni*:

```
data(tico)
tuningfork <- readWave("sample/tuning-fork.wav")
hello <- readWave("sample/hello.wav")
femo <- readWave("sample/Allobates_femoralis.wav")
data(orni)
```

15.1 Setting the Amplitude Envelope

The amplitude envelope can be changed through edition functions, as detailed in Chap. 6, but a fancy change could be to create a sort of sound chimera by applying the amplitude envelope of one sound to another one, such that the second sound would have the amplitude envelope of the first sound. This kind of mix, which can be useful in psychoacoustic or bioacoustic experiments, can be performed with the `seewave` function `setenv()`.

In the following example, we apply the envelope of `tico` to `tuningfork` so that the original amplitude envelope of `tuningfork` is drastically changed with the four-note amplitude envelope of `tico`. In this particular case, this manipulation could be used in a playback experiment where the frequency properties of the `tico`

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_15

sound would be tested. The code starts with a downsampling of `tuningfork` so that both objects have a similar sampling frequency $f_s = 22{,}050$ Hz:

```
f <- tico@samp.rate
tuningfork.u <- resamp(tuningfork, g=f)
```

We then call `setenv()` which has two main arguments: the first argument, `wave1`, is the input sound, and the second argument, `wave2`, is the sound which amplitude envelope will be used as a reference. The result is visualized with `spectro()` (Fig. 15.1):

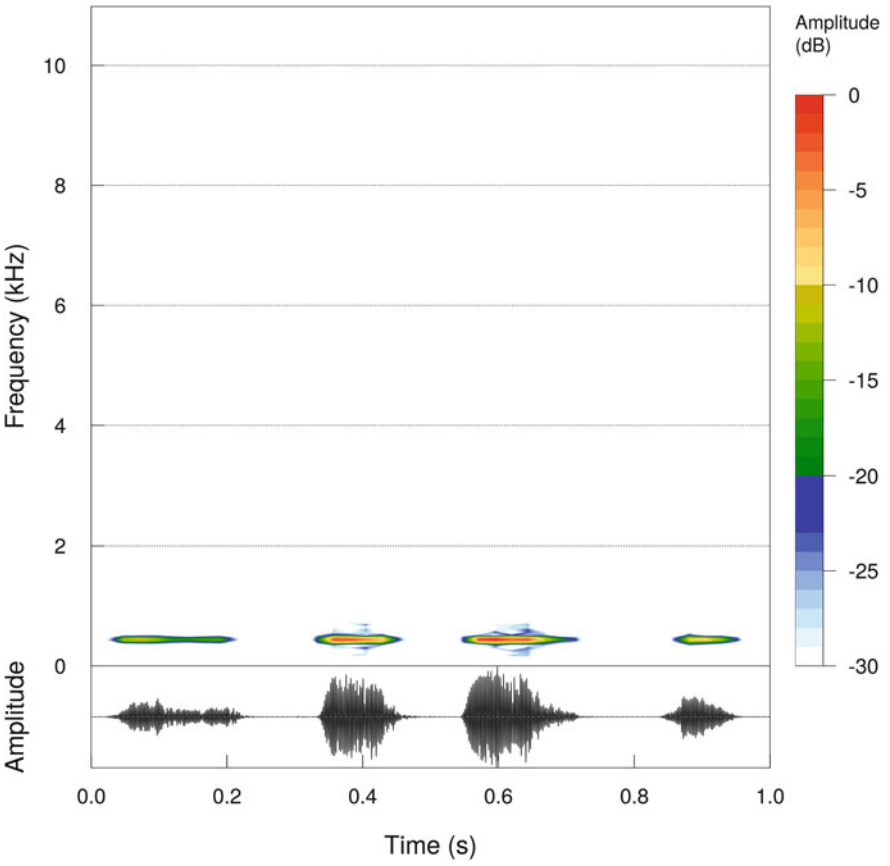


Fig. 15.1 Changing the amplitude envelope with `setenv()`. The amplitude envelope of `tico` was applied to `tuningfork`. Fourier window size = 512 samples, overlap = 0%, Hanning window

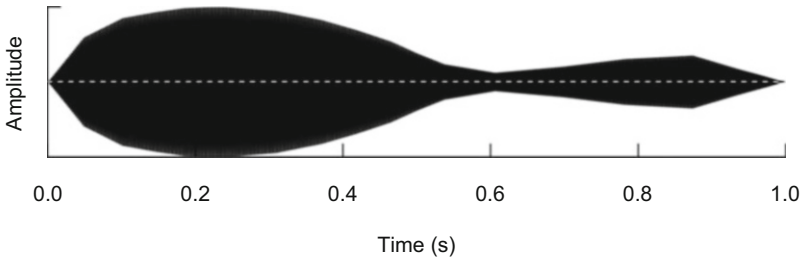


Fig. 15.2 Changing the amplitude envelope with `drawenv()`. The amplitude envelope of `tico` was modified graphically using the mouse cursor

```
res <- setenv(tuningfork.u, tico, f=f,
              plot=FALSE, output="Wave")
spectro(res, osc=TRUE)
```

As we did with the transfer function of a frequency filter (see Sect. 14.6.4), we could need to modify the amplitude envelope by drawing it on the graphical device. The interactive graphical function `drawenv()` displays the oscillogram of the original sound and lets the user draw a new envelope profile by clicking several times on the waveform (Fig. 15.2):

```
res <- drawenv(tuningfork, output="Wave")
```

15.2 Echoes and Reverberation

The exploration of echolocation and reverberation might require the artificial addition of echoes to a signal. The `seewave` function `echo()` adds several echoes to an input sound. This function, which is based on a similar principle to the FIR filter (see Sect. 14.6), processes a convolution between the input wave and a pulse echo filter. There are two parameters defining an echo: its relative amplitude compared to the input signal and its time position or delay in reference to the beginning of the input signal. These parameters are set with the arguments `amplitude` and `delay`, respectively. For instance, choosing `delay=1` and `amp=0.8` results in the addition of an echo starting 1 s after the beginning of the input file with an amplitude scaled by a factor of 0.8 compared to the maximum of the input file.

In the following example, some reverberation is generated on `hello` using a simple echo with a high relative amplitude (0.9) and very brief delay (0.01 s). The

use of `listen=TRUE` allows to listen to the result directly:

```
echo(hello, amp=0.9, delay=0.01, output="Wave", listen=TRUE)
```

In this second example, three non-overlapping echoes decreasing in amplitude by a power of 2 and increasing in duration by a factor of 1, 2, and 3 are added to the same dataset `hello`:

```
echo(hello, amp=1/(2^(2:4)), delay=duration(hello)*1:3,
     output="Wave", listen=TRUE)
```

15.3 Amplitude Filtering

A way to attempt cleaning a sound is to replace any signal sample which amplitude absolute value falls below a threshold by a 0 value. Such amplitude filter is easy to implement and fast because it relies on the test of a simple condition:

$$s_{filtered}[n] = \begin{cases} 0 & \text{if } |s[n]| \leq \theta \\ s[n] & \text{if } |s[n]| > \theta \end{cases}$$

where $s[n]$ is the original signal and θ the threshold.

The `seewave` function `afilter()` does this easy job by referring to a threshold expressed in percentage relative to the maximum of the amplitude envelope. Here is a test with a threshold set with the argument `threshold` at 2% and 5%, respectively (Fig. 15.3):

```
res1 <- afilter(femo, threshold=2,
                output="Wave", plot=FALSE)
res2 <- afilter(femo, threshold=5,
                output="Wave", plot=FALSE)
```

The function `afilter()` is actually parsed by several functions which includes an argument `threshold` (e.g., `dfreq()`, `ifreq()`, `timer()`).

The function `afilter()` is far to be clever since it converts every sample below the threshold into 0, even if these samples are part of the main signal to be analyzed—here the frog vocalizations. It is thereby necessary to check whether the

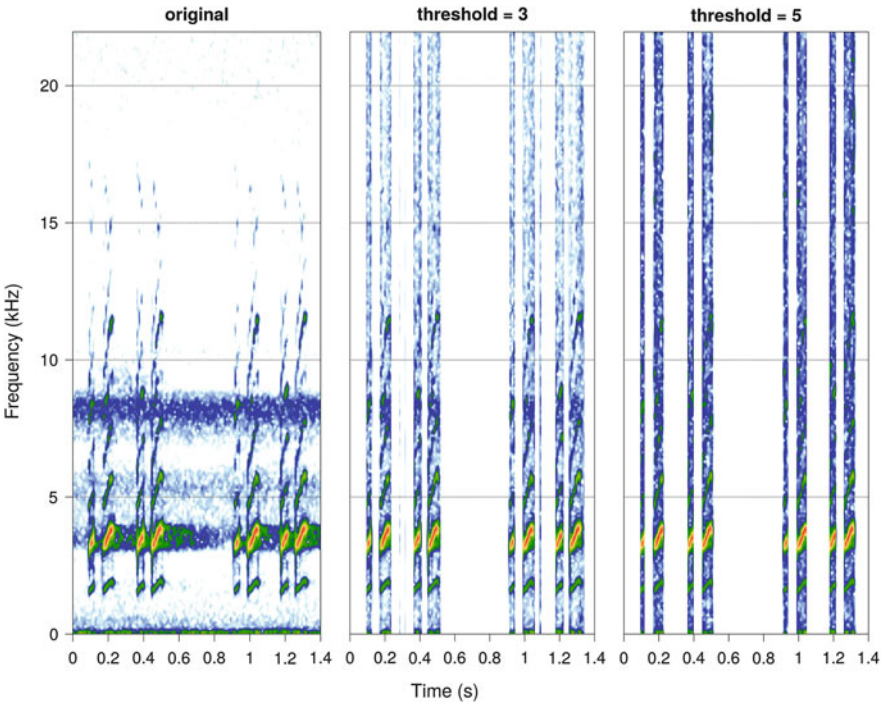


Fig. 15.3 Amplitude filter with `afilter()`. The original `femo` recording (left) was passed through an amplitude filter with a threshold of 3% (middle) and 5% (right). Fourier window size = 512 samples, overlap = 0%, Hanning windows

modified sound was not too much altered in its amplitude and frequency content. A simple test consists in listening to the results:

```
listen(res1)
listen(res2)
```

Here, the sounds appear slightly distorted. Such distortion may introduce some artifacts when tracking the frequency feature, as the dominant frequency. This was the case for the `sheep` dataset as shown in Fig. 13.2. Here, in the case of `femo`, the impact on frequency tracking is negligible as illustrated in Fig. 15.4:

```
df1 <- dfreq(femo, plot=FALSE)
df2 <- dfreq(res1, plot=FALSE)
df3 <- dfreq(res2, plot=FALSE)
```

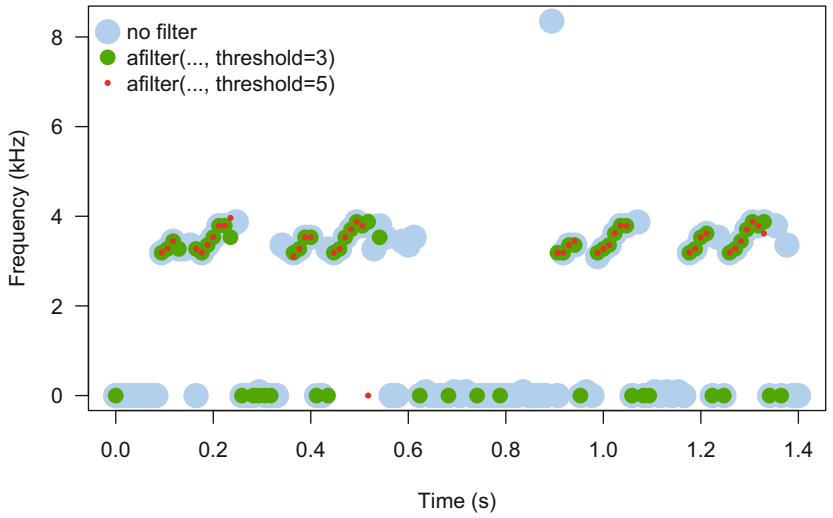


Fig. 15.4 Use of `afilter()` on dominant frequency tracking. The graphic shows the results of tracking the dominant frequency of `femo` after having filtered the sound using `afilter()` with different settings

15.4 Modifications Using the ISTDFT

We have seen in Sect. 14.5.3 that we can use the inverse short-time Fourier transform (ISTFT/ISTDFT) to apply a band-pass or a band-stop filter to a particular time × frequency section of sound. The principle was based on the production a STDFT matrix, the inclusion of 0 values in the STDFT matrix, and then a return in the time domain using the ISTDFT. The changes of the STDFT matrix are not limited to the introduction of 0 values. Any modification can be applied to a time × frequency section. For instance, we can increase the amplitude of the first harmonic of the fourth note of `femo` by a simple multiplication by 20 (Fig. 15.5, top-left). We first compute the STDFT matrix and get time and frequency limits of the harmonic to be manipulated as presented in Sect. 14.5.3:

```
# STDFT matrix
f <- femo@samp.rate ; wl <- 512; ovlp <- 75; wn <- "hanning"
data <- spectro(femo, wl=wl, ovlp=ovlp, wn=wn,
                plot=FALSE, norm=FALSE, dB=NULL, complex=TRUE)
# time and frequency limits of the harmonic
tmin <- which.min(abs(data$time-0.44))
tmax <- which.min(abs(data$time-0.51))
fmin <- which.min(abs(data$freq-2.77))
fmax <- which.min(abs(data$freq-4.21))
```

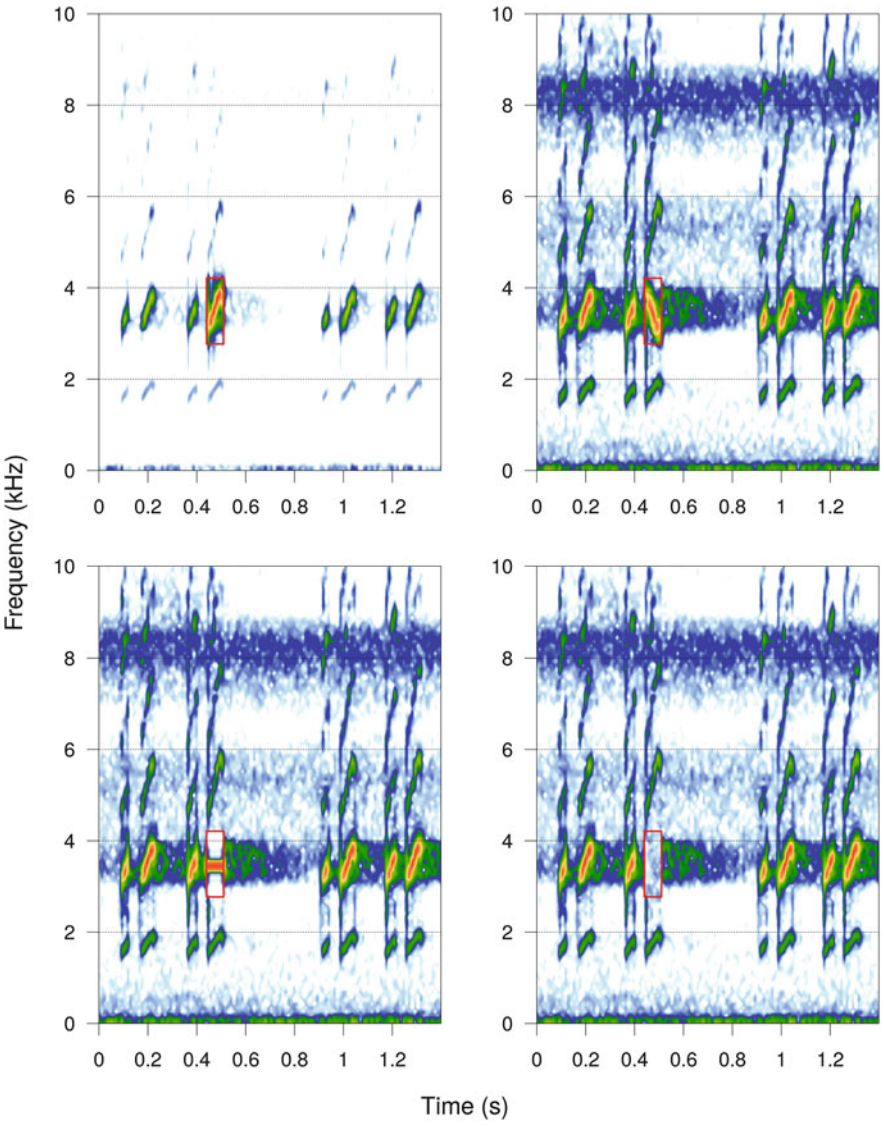


Fig. 15.5 Modifications using the ISTDFT. Four examples of sound modifications on `femo` based on the function `istft()`. The second harmonic of the first harmonic of the fourth note was amplified (top-left), reversed in frequency (top-right), replaced by a pure tone (bottom-left), and replaced by noise (bottom-right). Fourier window size = 512 samples, overlap = 0%, Hanning windows

We then apply the modification with (Fig. 15.5, top-left):

```
data1 <- data$amp
data1[fmin:fmax, tmin:tmax] <- 20*data1[fmin:fmax, tmin:tmax]
res1 <- istft(data1, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

We can also invert the harmonic in frequency (Fig. 15.5, top-right):

```
data2 <- data$amp
data2[fmin:fmax, tmin:tmax] <- data2[fmax:fmin, tmin:tmax]
res2 <- istft(data2, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

We may also like to replace the harmonic by a pure tone (Fig. 15.5, bottom-left):

```
data3 <- data$amp
data3[fmin:fmax, tmin:tmax] <- 0
data3[fmin+(fmax-fmin)/2, tmin:tmax] <-
    complex(real=max(Re(data3)))
res3 <- istft(data3, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

Eventually we can replace the harmonic with white noise (Fig. 15.5, bottom-right)

```
data4 <- data$amp
re <- im <- rnorm(length(fmin:fmax)*length(tmin:tmax))
data4[fmin:fmax, tmin:tmax] <- complex(real=re, imaginary=im)
res4 <- istft(data4, wl=wl, ovlp=ovlp, wn=wn, f=f, output="Wave")
```

Figure 15.5 was produced with the following code. The graphic function named `plot.istft()` is designed to facilitate the repetition of the plots:

```
collevels <- seq(-80,0,1)
plot.istft <- function(x){
    spectro(x, collevels=collevels, flim=c(0,10),
            scale=FALSE, tlab="", flab="")
    rect(xleft=0.44, ybottom=2.77, xright=0.51, ytop=4.21,
         border="red", lwd=3)
    }
par(mfrow=c(2,2), mar=c(3,3,1,1), oma=c(2,2,0,0))
plot.istft(res1)
```

(continued)

```
plot.istft(res2)
plot.istft(res3)
plot.istft(res4)
mtext("Time (s)", side=1, outer=TRUE)
mtext("Frequency (kHz)", side=2, outer=TRUE, las=0)
```

We can also take advantage of the ISTDFT to shift positively or negatively the frequencies of a sound to be used, for instance, in playback experiments. The function `lfs()`—for **l**inear **f**requency **s**hift—of `seewave` can apply such drastic change. The main argument is `shift` that controls the frequency shift in Hz. If we take the `orni` song, we first apply a high-pass FIR filter to remove the wind noise that could generate undesired frequency bands:

```
orni.filtered <- fir(orni, from=150, output="Wave")
```

We afterward use `lfs()` to apply a positive and a negative frequency shift of 1000 Hz (Fig. 15.6):

```
orni.higher <- lfs(orni.filtered, shift=1000, output="Wave")
orni.lower <- lfs(orni.filtered, shift=-1000, output="Wave")
```

The process of frequency shift can be easily included in a `for` loop to generate a succession of modified sounds as illustrated in the DIY box 15.1.

DIY 15.1 — How to generate a series of sounds with different linear frequency shifts

It can be necessary for the purpose of an experiment to generate a series of stimuli that result of a linear frequency shift. We can, for instance, think that in the case of the cicada *Cicada orni*, a playback experiment could aim at testing the importance of frequency in the encoding-decoding process of species recognition. To run such a test, it could be necessary to apply a series of regular shift between −2000 and +2000 Hz with a step of 500 Hz. We first prepare a numeric vector containing the shift values:

```
shift <- seq(-2000, 2000, by=500)
```

(continued)

DIY 15.1 (continued)

We remove the 0 value:

```
shift <- shift[shift!=0]
```

so that we obtain the vector:

```
shift
[1] -2000 -1500 -1000  -500   500  1000  1500  2000
```

We use a FIR filter to remove low-frequency noise due to wind:

```
orni.filtered <- fir(orni, from=150, output="Wave")
```

We write a `for` loop that calls the function `lfs()` to apply the frequency change and the function `savewav()` to export the modified sounds into individual `.wav` file files named `orni_lfs_-2000.wav`, `orni_lfs_-1500.wav`, `orni_lfs_-1000.wav`, etc.:

```
for(i in 1:length(shift)){
        tmp <- lfs(orni.filtered, shift=shift[i],
                   output="Wave")
        savewav(tmp,
                file=paste("orni_lfs_", shift[i], ".wav",
                           sep=""))
        }
```

15.5 Modifications Using the Hilbert Transform

We have seen in Sects. 5.2.1 and 13.1.4.1 that the Hilbert transform can be used to obtain the amplitude envelope and the instantaneous frequency of monotonal sounds. If we call the `seewave` functions `env()` and `ifreq()`, we can store the Hilbert amplitude envelope and the instantaneous frequency of `tico` in two

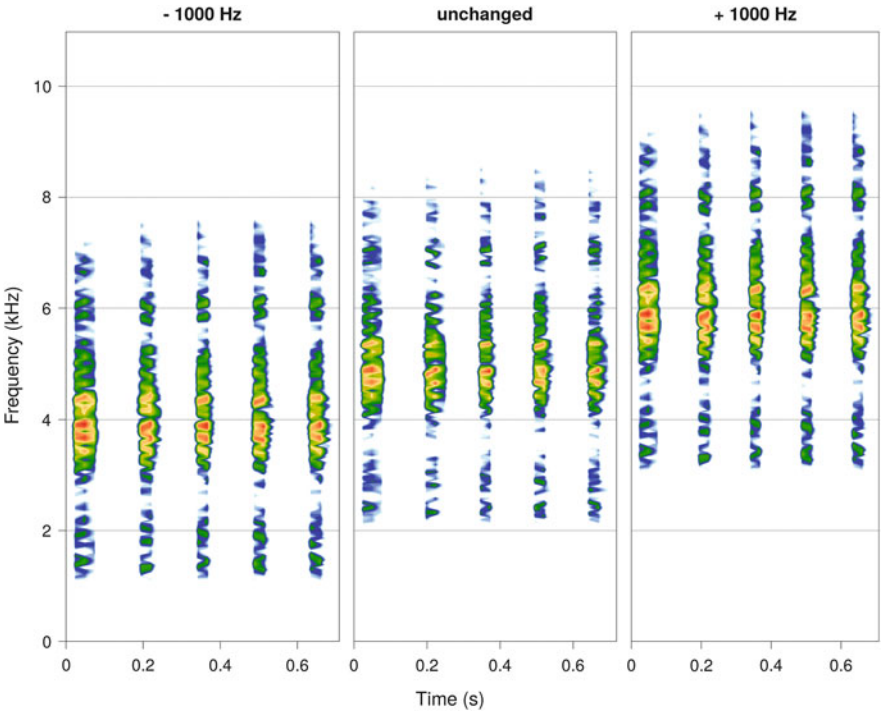


Fig. 15.6 Linear frequency shift using the ISTDFT. The song of `orni` was shifted toward low or high frequencies with the function `lfs()` that uses the ISTDFT in background. Fourier window size = 512 samples, overlap = 0%, Hanning window

distinct objects:

```
env.tico <- env(tico, plot=FALSE)
ifreq.tico <- ifreq(tico, plot=FALSE)
```

`seewave` includes a function to synthesize sound, `synth2()`, that uses as input the Hilbert amplitude envelope and the instantaneous frequency. This function, which is detailed in Sect. 18.5, can be used to recover the original sound of `tico`. We just extract and multiply by 1000 the second column of `ifreq.tico` that contains the instantaneous frequency in Hz:

```
f <- tico@samp.rate
ifreq.tico <- ifreq.tico$f[,2]*1000
```

We use these parameters to feed the function `synth2()`:

```
res <- synth2(env=env.tico, ifreq=ifreq.tico,
              f=f, output="Wave")
```

The object `res` is similar to `tico` so that the manipulation is not that interesting. However, we can play with envelope and instantaneous frequency independently to modify the sound in a more fancy way. First we could inverse the amplitude envelope but not the frequency modulations[1]:

```
res <- synth2(env=rev(env.tico), ifreq=ifreq.tico,
              f=f, output="Wave")
```

Second, we may reverse the instantaneous frequency but not the envelope (Fig. 15.7, left):

```
res <- synth2(env=env.tico, ifreq=rev(ifreq.tico),
              f=f, output="Wave")
```

We can apply any arithmetic operation as squaring the amplitude envelope:

```
res <- synth2(env=env.tico^2, ifreq=ifreq.tico,
              f=f, output="Wave")
```

shifting the instantaneous frequency by adding 1000 Hz:

```
res <- synth2(env=env.tico, ifreq=ifreq.tico+1000,
              f=f, output="Wave")
```

[1] A simple time reversion of the sound can be obtained with the function `revw()` as seen in Sect. 6.3.5.

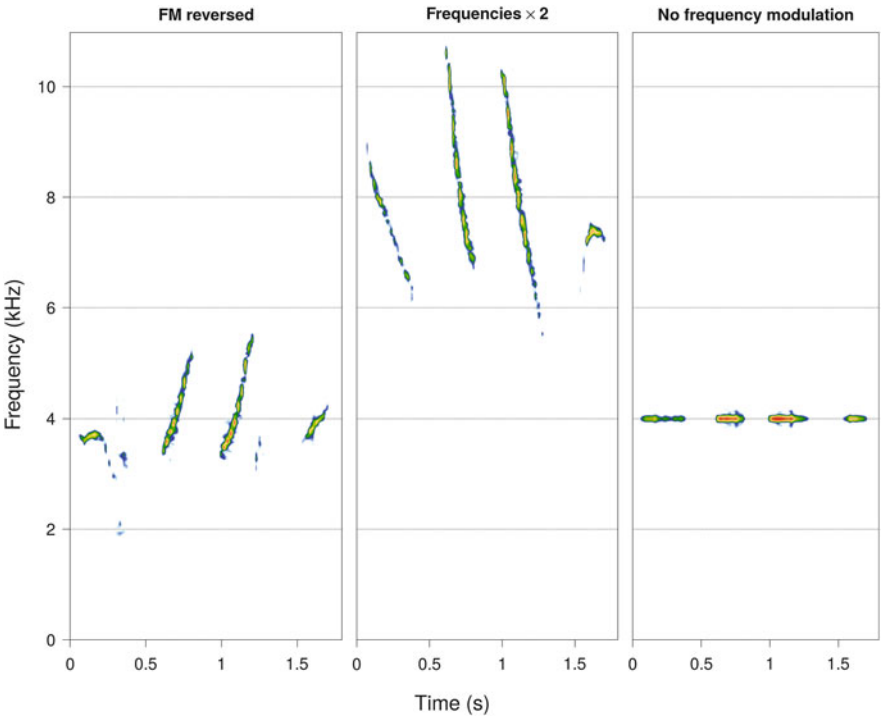


Fig. 15.7 Modifications using the Hilbert transform. Three examples of sound modifications on `tico` based on the function `synth2()`. The frequency modulation was inverted according to time (left), the frequencies were multiplied by 2 (middle), and the frequency modulation was replaced by 4000 Hz pure tone (right). Fourier window size = 512 samples, overlap = 0%, Hanning window

or multiplying the instantaneous frequency by a factor of 2 (Fig. 15.7, middle):

```
res <- synth2(env=env.tico, ifreq=ifreq.tico*2,
              f=f, output="Wave")
```

We can also generate slightly more complex modifications. For instance, we can selectively change the second note which position can be estimated automatically using the function `timer()` (see Sect. 8.3):

```
# timer() call
tres <- timer(tico, threshold=5, msmooth=c(50,0), plot=FALSE)
# start position of the second note
```

(continued)

```
start <- tres$s.start[2]
# end position of the second note
end <- tres$s.end[2]
# assignation of env.tico object in a new object
env.tico.mod <- env.tico
```

We replace the sample values of the second note by the maximum of the amplitude envelope with:

```
env.tico.mod[floor(start*f):floor(end*f)] <- max(env.tico)
```

that we use with `synth2()`:

```
res <- synth2(env=env.tico.mod, ifreq=ifreq.tico,
              f=f, output="Wave")
```

The second note now appears as a rectangular signal but still containing the original frequency modulation. This suggests that the amplitude modulation could be removed without touching to the frequency modulation, a manipulation often carried out in animal playback experiments. Such manipulation is also directly accessible with the `seewave` function `rmam()`:

```
res <- rmam(tico, plot=FALSE, output="Wave")
```

At the opposite, we might intend to keep the amplitude modulation but to remove any frequency modulation. Here is a possible solution by replacing the frequency modulation by a 4000 Hz pure tone (Fig. 15.7, right):

```
res <- synth2(env=env.tico, ifreq=rep(4000,
              times=length(ifreq.tico)),
              f=f, output="Wave")
```

Chapter 16
Indices for Ecoacoustics

The recent development of ecoacoustics, a discipline that aims at tackling ecology questions through the lens of acoustics (Sueur and Farina 2015), has been accompanied by the development of acoustic indices. These indices derive from historical biodiversity indices that are mathematical functions designed to evaluate some aspects of biodiversity (Magurran and McGill 2011). Acoustic indices aspire at estimating the complexity of the sound emerging from animal populations, animal communities, and landscapes.

There is now a long list of acoustic indices that can be split into two main groups: (1) α acoustic indices that are designed to evaluate the acoustic diversity of a single unit, defined as a population, a community, or a landscape at a specific time, and (2) β acoustic indices that are developed to compare the acoustic diversity of two units, for instance, two communities recorded from two different sites or the same community recorded at two different periods of the year (Sueur et al. 2014).

This chapter is a review of the α and β indices that can be computed with the packages `seewave` and `soundecology`, but their "quality" to infer ecological patterns and processes will not be treated here. Usage and test of the indices can be found in several papers (Fuller et al. 2015; Kendrick et al. 2016; Gasc et al. 2015; Lellouch et al. 2014; Towsey et al. 2014).

The acoustic indices will be introduced by referring to the file `forest.wav`, a sound of the tropical forest recorded in French Guiana previously introduced in Sect. 11.7.2:

```
forest <- readWave("sample/forest.wav")
```

We will also analyze a series of 24 files lasting each 20 s recorded as well in French Guiana in the Nouragues wildlife sanctuary at the cross of the trails labeled "M" and "XV." These stereo files were obtained every hour from 00:00

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_16

Fig. 16.1 Recording the French Guiana tropical acoustic communities. Twelve autonomous recorder SM2 of the company Wildlife Acoustics© were settled in the Nouragues reserve in French Guiana to record both understory and canopy acoustic communities. For each recorder, one microphone was installed at 1.5 m (understory recording), and another one was set at a height of 20 m (canopy recording). The hanging microphone on the right of the picture is ready to be sent up to the canopy. Picture by Jérôme Sueur and Amandine Gasc

to 23:00 on the 25th of November 2010 with a Song Meter SM2 of the company Wildlife Acoustics©. One microphone was installed at a height of 1.5 m to record the understory soundscape (right channel), and a second microphone was settled at a height of 20 m to capture the canopy soundscape (left channel) (Rodriguez et al. 2014) (Fig. 16.1). These 24 files are stored in a subdirectory named `guiana` included in the directory `sample`. The names of the files can be listed using the function `dir()` as seen in Sect. 4.2.5:

```
files <- dir("sample/guiana", pattern="wav$")
files
 [1] "M-XV_20101125_000000.wav" "M-XV_20101125_010000.wav"
 [3] "M-XV_20101125_020000.wav" "M-XV_20101125_030000.wav"
 [5] "M-XV_20101125_040000.wav" "M-XV_20101125_050000.wav"
 [7] "M-XV_20101125_060000.wav" "M-XV_20101125_070000.wav"
 [9] "M-XV_20101125_080000.wav" "M-XV_20101125_090000.wav"
[11] "M-XV_20101125_100000.wav" "M-XV_20101125_110000.wav"
[13] "M-XV_20101125_120000.wav" "M-XV_20101125_130000.wav"
```

(continued)

```
[15] "M-XV_20101125_140000.wav" "M-XV_20101125_150000.wav"
[17] "M-XV_20101125_160000.wav" "M-XV_20101125_170000.wav"
[19] "M-XV_20101125_180000.wav" "M-XV_20101125_190000.wav"
[21] "M-XV_20101125_200000.wav" "M-XV_20101125_210000.wav"
[23] "M-XV_20101125_220000.wav" "M-XV_20101125_230000.wav"
```

The names of the files were automatically generated by the Song Meter device so that we can use the `seewave` function `songmeter()` to get details about the context of the recordings (see Sect. 4.2.7):

```
head(songmeter(files))
    model prefix mic year month day hour min sec
1 SM2/SM4   M-XV  NA 2010    11  25    0   0   0
2 SM2/SM4   M-XV  NA 2010    11  25    1   0   0
3 SM2/SM4   M-XV  NA 2010    11  25    2   0   0
4 SM2/SM4   M-XV  NA 2010    11  25    3   0   0
5 SM2/SM4   M-XV  NA 2010    11  25    4   0   0
6 SM2/SM4   M-XV  NA 2010    11  25    5   0   0
                 time geo
1 2010-11-25 00:00:00  NA
2 2010-11-25 01:00:00  NA
3 2010-11-25 02:00:00  NA
4 2010-11-25 03:00:00  NA
5 2010-11-25 04:00:00  NA
6 2010-11-25 05:00:00  NA
tail(songmeter(files))
     model prefix mic year month day hour min sec
19 SM2/SM4   M-XV  NA 2010    11  25   18   0   0
20 SM2/SM4   M-XV  NA 2010    11  25   19   0   0
21 SM2/SM4   M-XV  NA 2010    11  25   20   0   0
22 SM2/SM4   M-XV  NA 2010    11  25   21   0   0
23 SM2/SM4   M-XV  NA 2010    11  25   22   0   0
24 SM2/SM4   M-XV  NA 2010    11  25   23   0   0
                 time geo
19 2010-11-25 18:00:00  NA
20 2010-11-25 19:00:00  NA
21 2010-11-25 20:00:00  NA
22 2010-11-25 21:00:00  NA
23 2010-11-25 22:00:00  NA
24 2010-11-25 23:00:00  NA
```

We can check the audio specifications by reading the header of each file with `readWave(..., header=TRUE)` (see Sect. 4.2.1) and looping the process with the base function `sapply()` (see Sect. 3.3.5.2):

```
files.spec <- sapply(X=files,
                     FUN=function(x)
                     readWave(paste("sample/guiana/", x, sep=""),
                        header=TRUE)
                     )
```

The specifications of all the files can be explored by just printing `files.spec`:

```
head(files.spec)
```

16.1 α Indices

16.1.1 Functions

There are 11 α indices available in `R` (Table 16.1). We will review sequentially several indices on the dataset `forest`. It is indeed often advised to compute and compare several indices at the same time.

The **bioacoustics index**, BI, was coined to assess relative avian abundance. The index consists in computing the dB mean spectrum and in calculating the area under the curve between two frequency limits, originally between 2000 and 8000 Hz. We can use the function `bioacoustic_diversity()` of `soundecology` tuning the arguments `min_freq` and `max_freq` to increase the frequency bandwidth. As `forest` is a mono sound, the main result is stored in the item `$left_area`:

```
res <- bioacoustic_index(forest, min_freq=500, max_freq=16000)

 This is a mono file.

 Calculating index. Please wait...

  Bioacoustic Index: 42.21748
  res$left_area
  [1] 42.21748
```

The **amplitude index**, M, is an amplitude index that computes the median of the amplitude envelope, either the absolute or Hilbert amplitude envelope, scaled by the

Table 16.1 α acoustic indices: name, function, package, and main literature reference

Name	Function	Package	Reference
Acoustic Complexity Index	`ACI()`	`seewave`	Pieretti et al. (2011)
Acoustic diversity index	`acoustic_complexity()`	`soundecology`	Villanueva-Rivera et al. (2011)
	`acoustic_diversity()`	`soundecology`	
Acoustic entropy index	`H()`	`seewave`	Sueur et al. (2008b)
Acoustic evenness index	`acoustic_evenness()`	`soundecology`	Villanueva-Rivera et al. (2011)
Acoustic richness index	`AR()`	`seewave`	Depraetere et al. (2012)
Bioacoustic index	`bioacoustic_index()`	`soundecology`	Boelman et al. (2007)
Frequency peaks number	`fpeaks()`	`seewave`	Gasc et al. (2013b)
Amplitude index	`M()`	`seewave`	Depraetere et al. (2012)
Normalized difference soundscape index	`NDSI()`	`seewave`	Kasten et al. (2012)
Spectral entropy	`ndsi()`	`soundecology`	Sueur et al. (2008b)
	`sh()`	`seewave`	
Temporal entropy	`th()`	`seewave`	Sueur et al. (2008b)

The indices are sorted out by alphabetic order

digitization depth of the recording. The formula is:

$$M = \text{median}(a_i) \times 2^{1-\text{depth}}$$

with $M \in [0, 1]$.

The function `M()` is easy to use and is by default based on the Hilbert amplitude envelope so that no argument needs to be changed:

```
res <- M(forest)
res
[1] 0.01798825
```

The **temporal entropy index**, H_t, estimates the Shannon evenness of the amplitude envelope. The amplitude envelope, usually the Hilbert amplitude envelope, is scaled by its sum, so that the sum of the sample values equals to 1. This is equivalent to transform the amplitude envelope into a probability mass function.

For an amplitude envelope made of n samples, the index is therefore computed according to:

$$H_t = -\frac{\sum_{i=1}^{n} a_i \log(a_i)}{\log(n)}$$

with $\sum_{i=1}^{n} a_i = 1$ and $H_t \in [0, 1]$.

The computation of H_t with `R` consists in a two-step process including the computation of the amplitude envelope with `env()` (see Sect. 5.2.1) and then the calculation of the index with `th()`:

```
forest.env <- env(forest, plot=FALSE)
res <- th(forest.env)
res
[1] 0.9889863
```

This can also be written with a single line code:

```
res <- th(env(forest, plot=FALSE))
```

The **acoustic richness index**, AR, is based on the ranks of the indices M and H_t obtained for a set of n files. The indices M and H_t are first computed for each file and then sorted into ascending order. The position, or rank, of each file in this forward sort is then used to compute AR. The index, which is scaled between 0 and 1, depends therefore on the set of the files considered. The expression of AR is:

$$AR = \frac{\text{rank}(M) \times \text{rank}(H_t)}{n^2}$$

with $AR \in [0, 1]$.

The function `AR()` returns the AR index and its companions M and H_t indices for either a set of `R` objects or for a series of files stored in the working directory. This choice between objects and files is set with the argument `datatype`. In the following code, the current working directory is saved in an object, a new working directory is chosen to point out to the directory containing the `.wav` files, the AR index is computed specifying `datatype="files"`, the results are stored and displayed with an object of class `data.frame`, and lastly the original working

directory is restored:

```
oldwd <- getwd()
setwd("sample/guiana/")
res <- AR(getwd(), datatype="files")
res
                                  M         Ht           AR
M-XV_20101125_000000 0.043846503 0.9888873 0.659722222
M-XV_20101125_010000 0.037427357 0.9886404 0.562500000
M-XV_20101125_020000 0.034101580 0.9888316 0.560763889
M-XV_20101125_030000 0.026792128 0.9885353 0.413194444
M-XV_20101125_040000 0.023252848 0.9884119 0.361111111
M-XV_20101125_050000 0.029294027 0.9882671 0.388888889
M-XV_20101125_060000 0.020471720 0.9895142 0.458333333
M-XV_20101125_070000 0.014155508 0.9851849 0.038194444
M-XV_20101125_080000 0.011916565 0.9882846 0.208333333
M-XV_20101125_090000 0.009461191 0.9873620 0.060763889
M-XV_20101125_100000 0.009815709 0.9873231 0.060763889
M-XV_20101125_110000 0.009706416 0.9873461 0.062500000
M-XV_20101125_120000 0.009216894 0.9879034 0.062500000
M-XV_20101125_130000 0.008761356 0.9876396 0.041666667
M-XV_20101125_140000 0.007850435 0.9880655 0.041666667
M-XV_20101125_150000 0.007297304 0.9853045 0.005208333
M-XV_20101125_160000 0.013712839 0.9896097 0.399305556
M-XV_20101125_170000 0.028655706 0.9879145 0.260416667
M-XV_20101125_180000 0.012190863 0.9694965 0.015625000
M-XV_20101125_190000 0.083331516 0.9882367 0.541666667
M-XV_20101125_200000 0.057486511 0.9869541 0.159722222
M-XV_20101125_210000 0.050075458 0.9902678 0.833333333
M-XV_20101125_220000 0.052779448 0.9879280 0.420138889
M-XV_20101125_230000 0.052096677 0.9889630 0.765625000
setwd(oldwd)
```

The **spectral entropy index**, H_f, follows the same principle as H_t but works in the frequency domain. H_f is actually the Shannon evenness of the frequency spectrum (see Sect. 10.1.6.3), usually the mean spectrum (see Sect. 11.14). The index is constrained between 0 and 1 by transforming the frequency spectrum into a probability mass function as done for the Hilbert amplitude envelope with H_t. The equation used to compute H_f for a frequency spectrum made of n frequency bins is:

$$H_f = -\frac{\sum_{i=1}^{n} f_i \log f_i}{\log(n)}$$

with $\sum_{i=1}^{n} f_i = 1$ and $H_f \in [0, 1]$.

The function `sh()` calculates H_f for a spectrum obtained upstream, such that, as in the case of H_t, the index is obtained by calling two functions:

```
mspec <- meanspec(forest, plot=FALSE)
res <- sh(mspec)
res
[1] 0.8503514
```

This code can be compressed into:

```
res <- sh(meanspec(forest, plot=FALSE))
```

The **acoustic entropy index**, H, is the multiplication of H_t and H_f:

$$H = H_t \times H_f$$

with $H \in [0, 1]$.
The function `H()` can do the computation in a direct way:

```
res <- H(forest)
res
[1] 0.8409859
```

This is equivalent to the multiplication of H_t and H_f obtained with `th()` and `sh()`, respectively:

```
res.t <- th(env(forest, plot=FALSE))
res.f <- sh(meanspec(forest, plot=FALSE))
res <- res.t*res.f
res
[1] 0.8409859
```

The **acoustic diversity index**, ADI, uses, as H_f does, the Shannon entropy on the spectral content. The ADI index computes the STDFT, cuts the STDFT into a determined number of bins, selects the relative amplitude of each bin that is above a dB threshold, and applies the Shannon entropy index on these selected values. By default the frequency bandwidth between 0 and 10 kHz is split into 10 bins, and the dB threshold is set to -50 dB. The function `acoustic_diversity()` of `soundecology` is dedicated to this index:

```
res <- acoustic_diversity(forest)

 This is a mono file.

 Calculating index. Please wait...

  Acoustic Diversity Index: 2.110825
```

All these parameters can be modified using the arguments `max_freq` for the upper frequency limit of the analysis in Hz, `freq_step` for the width of each frequency in Hz, and `db_threshold` for the dB threshold. In the following, ADI covers the full frequency range ($f_s = 44{,}100 \div 2 = 22{,}050$ Hz) with frequency bins of 500 Hz and a threshold of -40 dB:

```
res <- acoustic_diversity(forest, max_freq=forest@samp.rate/2,
                          db_threshold=-40, freq_step=500)
```

The function `acoustic_diversity()` prints the result in the console, but the object returned is a list of items containing the results of each channel. Here, `forest` being a mono sound, the main result is stored in the item `$adi_left`:

```
res$adi_left
[1] 2.447162
```

A quick representation of the frequency bins used by ADI can be obtained using two other items that contain the amplitude value of each frequency bin (`$left_band_values`) and their frequency limits (`$left_bandrange_values`). Some general graphical parameters can be changed with the function `par()` to improve the display of the x-axis based on `barplot()` as in (Fig. 16.2):

```
par(las=2, cex.axis=0.75, oma=c(1,0,0,0))
barplot(height=res$left_band_values,
        names=res$left_bandrange_values,
        ylab="Proportion")
```

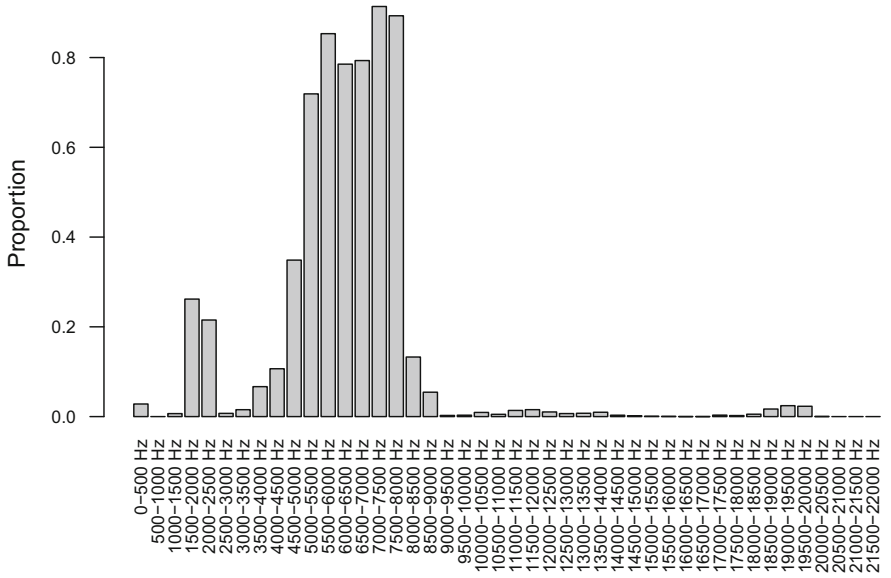


Fig. 16.2 Barplot of the values used by the acoustic diversity index (ADI). The relative amplitude values of the frequency bins used to compute ADI are plotted as a barplot. The values were obtained with a maximum frequency of 22,050 Hz and a frequency step of 500 Hz

The **acoustic evenness index**, AEI, proceeds the same first step as ADI but computes the Gini coefficient which is a measure of distribution inequality. The function `acoustic_evenness()` works as `acoustic_diversity()` but uses the function `ineq()` of the package `ineq` to compute the Gini index G. The `ineq()` function uses the following equation:

$$G = \frac{2\sum_{i=1}^{n} i a_i}{n\sum_{i=1}^{n} a_i} - \frac{n+1}{n}$$

where the values a_i are sorted into ascending order, such that $a_i \leq a_{i+1}$.

The default use of `acoustic_evenness()` is rather simple:

```
res <- acoustic_evenness(forest)
```

This can be tuned with the same arguments as in `acoustic_diversity()`:

```
res <- acoustic_evenness(forest, max_freq=forest@samp.rate/2,
                         db_threshold=-40, freq_step=500)
```

The results are stored this time in a two-item list, each item containing the result for each channel, here only for the left channel as `forest` is a mono sound:

```
res
$aei_left
[1] 0.796779

$aei_right
[1] NA
```

The **acoustic complexity index**, ACI, aims at measuring the complexity of STDFT matrix giving more importance to sounds that are modulated in amplitude and, hence, reducing the importance of sound with a rather constant amplitude as anthropogenic noise may have. The main principle of the ACI is to compute the average absolute amplitude difference between adjacent cells of the STDFT matrix in each frequency bin, that is, in each row of the STDFT matrix. If we refer to the construction of the STDFT matrix as explained in Sect. 11.1.1, the STDFT matrix is a collection of Fourier coefficients a_{kj}, with K the number of frequencies and J the number of Fourier windows computed along the signal:

$$\begin{array}{c} \\ \omega_1 \\ \vdots \\ \omega_k \\ \vdots \\ \omega_K \end{array} \begin{array}{c} \begin{array}{ccccc} n_1 & \dots & n_j & \dots & n_J \end{array} \\ \begin{pmatrix} a_{11} & \dots & a_{1j} & \dots & a_{1J} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{k1} & \dots & a_{kj} & \dots & a_{kJ} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{K1} & \dots & a_{Kj} & \dots & a_{KJ} \end{pmatrix} \end{array}$$

In the original description of ACI, long audio files were split into I time frames so that I STDFT matrices were obtained. If we consider a single frequency bin that is a single row k of one STDFT matrix, the ACI computes the derivative of the coefficients scaled by the sum of the coefficients:

$$ACI_j = \sum_{j=1}^{J-1} \left(|a_{j+1} - a_j| / \sum_{j=1}^{J} a_j \right)$$

The computation is processed for each frequency bin k, and the total is summed up, so that we have for a single time frame i:

$$\begin{aligned} ACI_{kj} &= \sum_{k=1}^{K} ACI_k \\ &= \sum_{k=1}^{K} \sum_{j=1}^{J-1} \left(|a_{j+1} - a_j| / \sum_{j=1}^{J} a_j \right)_k \end{aligned}$$

If we now generalize for all i time frames, we end up with:

$$\begin{aligned} ACI_{kji} &= \sum_{i=1}^{I} ACI_{kj} \\ &= \sum_{i=1}^{I} \sum_{k=1}^{K} \sum_{j=1}^{J-1} \left(|a_{j+1} - a_j| / \sum_{j=1}^{J} a_j \right)_{ik} \end{aligned}$$

If the final and complete equation of ACI seems to be complex, its computation is rather simple. There are two `R` functions to compute the ACI, the eponymous function `ACI()` of `seewave` and the function `acoustic_complexity()` of `soundecology`. `ACI()` was checked by Nadia Pierreti and Almo Farina and seems to be faster and will only be considered here.[1]

A basic computation of ACI, without time splitting is:

```
ACI(forest)
[1] 151.5964
```

The classic STDFT parameters can be changed as in:

```
ACI(forest, wl=1024, ovlp=50, wn="hanning")
[1] 294.7536
```

[1]See the `soundecology` vignette that explains the homology between `ACI()` and `acoustic_complexity()`. The vignette can be consulted from `R` with: `vignette ("ACIandSeewave", package="soundecology")`.

A frequency selection using the argument `flim` expressed in Hz can be applied to limit the computation to a specific frequency band, here a [4, 8] kHz band:

```
ACI(forest, flim=c(4,8))
[1] 25.96505
```

The number of time frames I can also be set with the argument `nbwindows`. In the following, the sound is divided into four frames of equal duration:

```
ACI(forest, nbwindows=4)
[1] 606.6612
```

The **number of frequency peaks**, NP, is the number of major peaks appearing in a frequency spectrum, usually the mean spectrum of the sound of interest. A high diversity of sounds is supposed to generate an important number of frequency peaks. We have seen in Sect. 10.1.3.1 that `fpeaks()` works as a peak detection function with several threshold parameters to select major frequency peaks and discard residual frequency peaks due to background noise. `fpeaks()` returns a two-column matrix where each line corresponds to the amplitude and frequency of each peak. The number of rows of this matrix is the index NP. To compute NP on `forest`, we do:

```
mspec <- meanspec(forest, plot=FALSE)
res <- nrow(fpeaks(mspec, amp=c(0.04,0.04), plot=FALSE))
res
[1] 6
```

Changing the arguments of `fpeaks()` that are used to select the frequency peaks—`amp`, `freq`, or `threshold`—will of course change the results. It is obviously recommended to set carefully these parameters and to keep them unchanged when comparing different sounds.

The **normalized difference soundscape index**, $NDSI$, aims at estimating the level of anthropogenic disturbance on the soundscape by computing the ratio of human-generated (anthropophony) to biological (biophony) acoustic components. The index computes the following ratio:

$$NDSI = \frac{(b-a)}{(b+a)}$$

where $b = biophony$, $a = anthropophony$, $NDSI \in [-1, 1]$, and $NDSI = 1$ indicate a sound containing no anthropophony. In terms of frequency, the anthropophony was originally defined as the $[1 - 2[$ kHz frequency bin and the biophony as the $[2 - 8[$ kHz frequency bins, but these values are supposed to be modified according to the soundscape explored and to the sampling frequency of the recording. The `seewave` function `NDSI()`, written with the control of Eric Kasten, computes the above ratio for a frequency spectrum obtained with the function `soundscapespec()` which returns a Welch's frequency spectrum binned into 1 kHz frequency bands (see Sect. 11.15). The computation of $NDSI$ follows then a two-step process:

```
sdspec <- soundscapespec(forest, plot=FALSE)
NDSI(sdspec)
[1] 0.9452764
```

The arguments `anthropophony` and `biophony` can be used to change the kHz frequency limits of the anthropophony and biophony, with here almost no changes on the result:

```
NDSI(sdspec, anthropophony=1, biophony=1:18)
[1] 0.9476306
```

16.1.2 Batch Processing: How to Obtain a List of α Indices for a Set of Sounds

The `soundecology` function `multiple_sounds()` allows the computation of one index on several files, but it is often necessary to compute several indices on several objects or files. The goal is here to compute a series of indices on the 24 files recorded in French Guiana. We retrieve the list of the `.wav` file names to analyze:

```
oldwd <- getwd()
setwd("sample/guiana")
files <- dir(pattern = "wav$")
```

We then write a new function, named `indices()`, that reads a `.wav` file and computes a selection of four indices, namely, H_f, AEI, ACI, and $NDSI$:

```
indices <- function(x){
    x <- readWave(x)
    return(c(sh(meanspec(x, plot=FALSE)),
             acoustic_evenness(x)$aei_left,
             ACI(x),
             NDSI(soundscapespec(x, plot=FALSE))
             )
           )
}
```

We prepare an object where the results will be written in:

```
n <- length(files)
num <- rep(NA, n)
res <- data.frame(Hf=num, AEI=num, ACI=num, NDSI=num,
                  row.names=files)
```

We use the new function on each mean spectrum with a `for` loop:

```
for(i in 1:n) res[i,] <- indices(files[i])
```

We print the results:

```
head(res)
                                  Hf       AEI       ACI
M-XV_20101125_000000.wav 0.8222687 0.055606 152.6392
M-XV_20101125_010000.wav 0.8245113 0.088373 152.4629
M-XV_20101125_020000.wav 0.8318868 0.096916 151.8242
M-XV_20101125_030000.wav 0.8410839 0.110988 151.6884
M-XV_20101125_040000.wav 0.8402450 0.074288 151.5751
M-XV_20101125_050000.wav 0.8172957 0.142642 152.3881
                                NDSI
M-XV_20101125_000000.wav 0.9834654
M-XV_20101125_010000.wav 0.9838402
M-XV_20101125_020000.wav 0.9820926
M-XV_20101125_030000.wav 0.9762023
M-XV_20101125_040000.wav 0.9602881
M-XV_20101125_050000.wav 0.9770428
```

And we finally restore the original working directory to get back where we were before to run the analysis:

```
setwd(oldwd)
```

16.2 β Indices

When can we say that two sounds differ or not? β indices consist in computing a distance that estimates how much two sounds are dissimilar. Finding an appropriate β acoustic dissimilarity index is actually very delicate as sound is by essence a multivariate object and two sounds may differ for one parameter, say their amplitude envelope, but not another one, say their fundamental frequency. It might be tricky as well to conclude whether a difference of a few Hz or a few ms is significant or not. In addition, to compare two sounds requires to identify homologous parameters. For instance, we need to be sure that we measure the same frequency band (for instance, the fundamental frequency) or that we measure the duration of the same note. This homology is not always trivial and might require a detailed exploration of the sounds structure.

Most of the β acoustic dissimilarity indices simplified these issues by working on the frequency domain only. Two frequency spectra computed with the same DFT or STDFT parameters can be considered as homologous objects where the frequency bins computed for one frequency spectrum can be paired with the frequency bins of another frequency spectrum. Nonetheless this homology does not solve everything. The comparison of two spectra, deriving from a physical or chemical analysis, is not an easy task. The difficulty of finding a proper metric probably explains why several indices have been coined (Table 16.2). These spectral distances often derive from classical geometric, probability, or statistic distances.

16.2.1 Functions

We will review the dissimilarity acoustic indices available in `R` one after one. The list of β indices could probably be lengthened with other distances that could be easy adapted to sound, as the Hellinger distance or the Bhattacharyya distance or any other measure proposed in the package `proxy` (Meyer and Buchta 2016). We could also consider the methods reviewed in Chap. 17 as potential indices.

Table 16.2 β acoustic indices: name, function, package, and main literature reference

Name	Function	Package	Reference
Temporal dissimilarity	`diffenv()`	`seewave`	Sueur et al. (2008b)
Cumulative spectral dissimilarity	`diffcumspec()`	`seewave`	Lellouch et al. (2014)
Spectral dissimilarity	`diffspec()`	`seewave`	Sueur et al. (2008b)
Wave dissimilarity	`diffwave()`	`seewave`	Sueur et al. (2008b)
Itakura-Saito distance	`itakura.dist()`	`seewave`	–
Kullback-Leibler distance	`kl.dist()`	`seewave`	Gasc et al. (2013a)
1-Mutual information	`1-symba()`	`seewave`	Cazelles (2004)
Kolmogorov-Smirnov distance	`ks.dist()`	`seewave`	Gasc et al. (2013a)
Log-spectral distance	`logspec.dist()`	`seewave`	–
Relative frequency dissimilarity	`100-simspec()`	`seewave`	Deecke and Janik (2006)
Correlation-based dissimilarity	`1-cor()`	`stats`	Lellouch et al. (2014)
RV dissimilarity	`1-coeffRV()`	`FactoMineR`	Gasc et al. (2013a)

Examples of use of most of these indices can be found as well in Depraetere et al. (2012), Gasc et al. (2013b), and Rodriguez et al. (2014)

In the following, f_i and g_i are two frequency spectra containing each n frequency bins. These frequency spectra can be scaled by their sum that is converted into probability mass functions x_i and y_i with:

$$x_i = \frac{f_i}{\sum_{i=1}^{n} f_i}$$

$$y_i = \frac{g_i}{\sum_{i=1}^{n} g_i}$$

so that $\sum_{i=1}^{n} x_i = 1$ and $\sum_{i=1}^{n} y_i = 1$.

The cumulative probability mass functions X_i and Y_i of x_i and y_i, respectively, are obtained with:

$$X_i = \sum_{j=1}^{i} x_j$$

$$Y_i = \sum_{j=1}^{i} y_j$$

To test the `R` functions dedicated to acoustic dissimilarity, we will use two sounds from the set of 24 files recorded in French Guiana, one corresponding to midnight and the other one corresponding to midday:

```
night <- readWave("sample/guiana/M-XV_20101125_000000.wav")
day <- readWave("sample/guiana/M-XV_20101125_120000.wav")
```

We select the left channel that caught the canopy acoustic communities by applying the `tuneR` function `mono()`:

```
night.left <- mono(night, which="left")
day.left <- mono(day, which="left")
```

We compute the mean spectrum of each sound with:

```
night.mspec <- meanspec(night.left, plot=FALSE)
day.mspec <- meanspec(day.left, plot=FALSE)
```

The **spectral dissimilarity index**, D_f, is probably the simplest way to compare f_i and g_i by computing the absolute difference by pair of frequency bin of x_i and y_i and by summing up these point-wise differences:

$$D_f = \frac{\sum_{i=1}^{n} |x_i - y_i|}{2}$$

with $D_f \in [0, 1]$

The function `diffspec()`, which computes D_f, waits two frequency spectra as input:

```
diffspec(night.mspec, day.mspec)
[1] 0.472607
```

Note that the function of D_f is symmetric so that the two following commands are equivalent:

```
diffspec(night.mspec, day.mspec)
[1] 0.472607
diffspec(day.mspec, night.mspec)
[1] 0.472607
```

The result can be visualized by turning the argument `plot` to `TRUE` (Fig. 16.3, top):

```
diffspec(night.mspec, day.mspec, plot=TRUE)
```

The **cumulative spectral dissimilarity** index, D_{cf}, proceeds the same way as D_f, but the pairwise absolute difference between frequency bins is computed on the cumulative probability mass functions X_i and Y_i:

$$D_{cf} = \frac{\sum_{i=1}^{n} |X_i - Y_i|}{n}$$

with $D_{cf} \in [0, 1]$.

The difference between D_f and D_{cf} might appear very subtle, but they may actually return contrasted results, as it is in the case for `night.left` and `day.left`. D_f is a point-wise metric that can return high values between two frequency spectra with a similar shape but shifted in frequency by a few Hz only (Fig. 16.4 top-left). Comparable values could actually be obtained for two frequency spectra totally differing in frequency (Fig. 16.4 top-right). The index D_{cf} does not have this drawback as it is sensitive to the spectral overlap between f_i and g_i but also to the mean frequency distance between the different frequency peaks of f_i and g_i (Fig. 16.4, bottom-left and bottom-right).

The function `diffcumspec()` works as `diffspec()` with a similar option to display the result (Fig. 16.3, middle). The input arguments may be swapped as well as the index is symmetric:

```
diffcumspec(night.mspec, day.mspec)
[1] 0.08755972
```

The **Kolmogorov-Smirnov** distance, D_{KS}, is a statistic metric used in the Kolmogorov-Smirnov nonparametric rank test. The distance is the maximal distance between the cumulative probability mass functions:

$$D_{KS} = \max_i |X_i - Y_i|$$

with $D_{KS} \in [0, 1]$.

The function `ks.dist()` computes and optionally displays the result. The advantage of the Kolmogorov-Smirnov metric is that the distance can be associated with a frequency. Here a distance of 0.386 is found at a frequency of 7.235 kHz (Fig. 16.3 bottom):

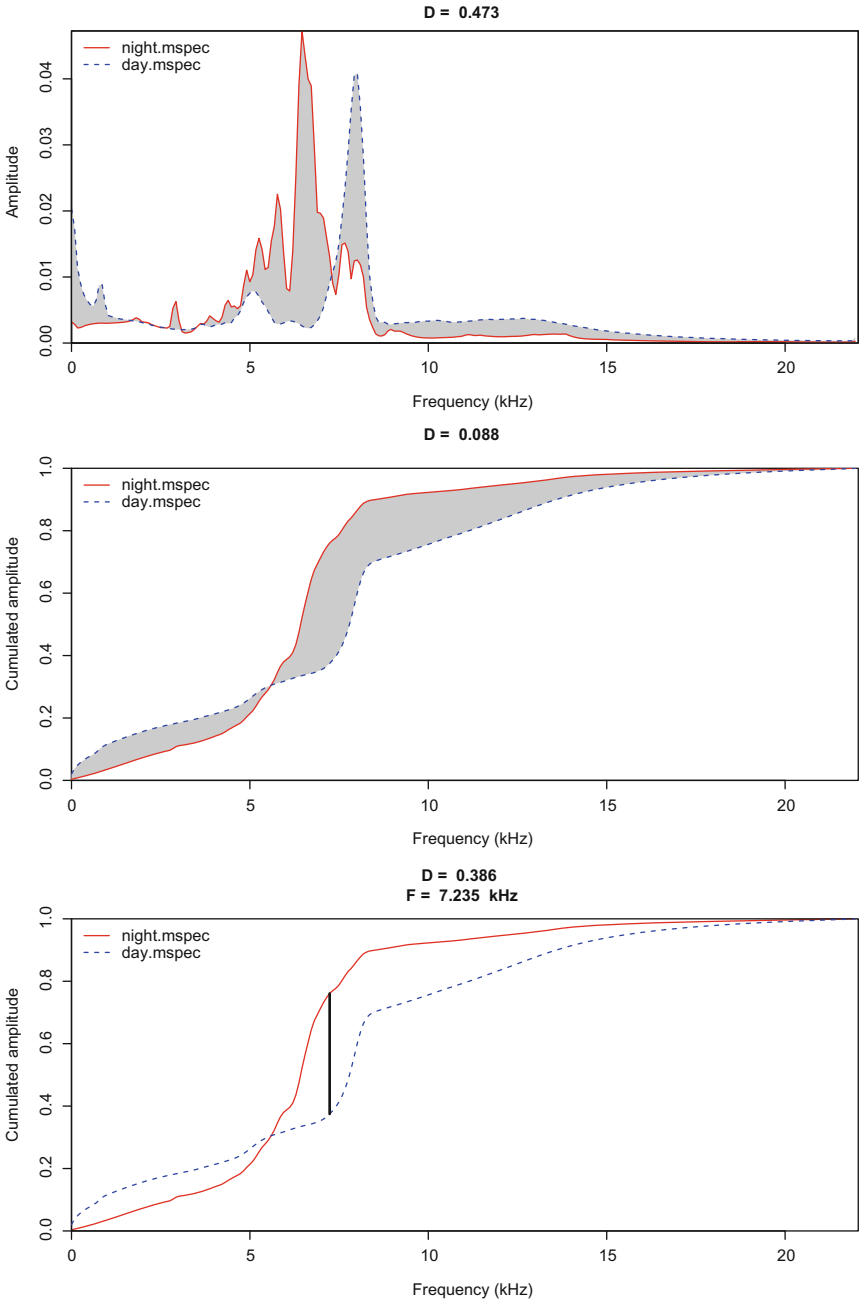


Fig. 16.3 Visualization of three β indices. Graphical output of the function `diffspec()` (top), `diffcumspec()` (middle), and `ks.dist()` (bottom) for the indices D_f, D_{cf}, and D_{KS}, respectively. In each case the mean spectra of the two sounds `night` and `day` were provided to the functions. The gray area or the segment indicates the dissimilarity index

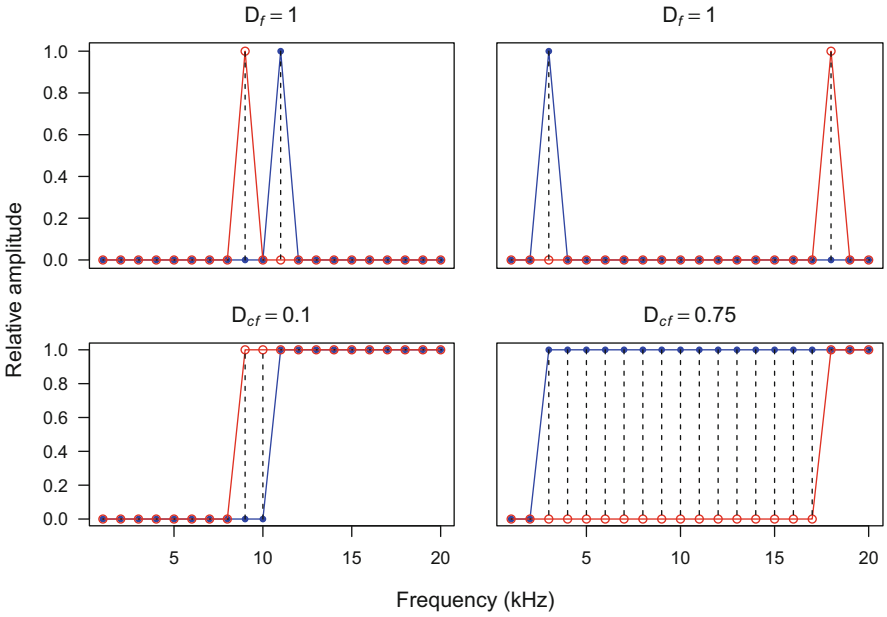


Fig. 16.4 Comparison between spectral dissimilarity index and cumulative spectral dissimilarity index. The indices D_f and D_{cf} return different values for spectra of similar shapes but with different frequency. The examples are here for pure-tone theoretic sounds. The index D_f returns the same value in the two cases (probability mass functions of two frequency spectra, top-left and top-right) when D_{cf} returns expected low and high values (cumulated probability mass functions of two frequency spectra, bottom-left and bottom-right)

```
ks.dist(night.mspec, day.mspec)
$D
[1] 0.3861037

$F
[1] 7.235156
```

Note that the distance returned by `ks.dist()` differs from the D statistics that could be obtained with the function `ks.test()` that applies the Kolmogorov-Smirnov test. The function `ks.test()` should not be fed with cumulative probability mass functions as `night.spec[,2]` and `day.spec[,2]` are but with raw data. This command is therefore a nonsense:

```
ks.test(night.mspec[,2], day.mspec[,2])$statistic
        D
0.3242188
```

The **Itakuro-Saito** distance, D_{IS}, is a nonsymmetric measure of the difference between two probability mass functions calculated following:

$$D_{IS}(x\|y) = \sum_{i=1}^{n} \frac{x_i}{y_i} - \log\left(\frac{x_i}{y_i}\right) - 1$$

A symmetrization of the distance can be operated by calculating the mean between the distances of each direction:

$$D_{IS} = \frac{D_{IS}(x\|y) + D_{IS}(y\|x)}{2}$$

with $D_{IS} \in [0, 1]$.

The Itakuro-Saito distances are provided by the function `itakura.dist()` as a list of three items. In the following example, the function returns: (1) `$D1` the distance of `day.mspec` with respect to `night.mspec`, (2) `$D2` the distance of `night.mspec` with respect to `day.mspec`, and `$D` the symmetric distance:

```
itakura.dist(night.mspec, day.mspec)
$D1
[1] 0.7338433

$D2
[1] 0.7871226

$D
[1] 0.7604829
```

The **Kullback-Leibler** divergence, D_{KL}, is another nonsymmetric measure of the difference between two probability mass functions. This distance has the advantage to be linked to the measurement of entropy, such that D_{KL} values can be therefore expressed in number of bits. The formula to obtain D_{KL} of f_i with respect to g_i is expressed as:

$$D_{KL}(f\|g) = \sum_{i=1}^{n} x_i \times \log\left(\frac{x_i}{y_i}\right)$$

A symmetry can be obtained by calculating the mean between the two directions:

$$D_{KL} = \frac{D_{KL}(f\|g) + D_{KL}(f\|g)}{2}$$

The Kullback-Leibler divergences are provided by the function `kl.dist()` as a list of three items in the same way as `itakura.dist()`: (1) `$D1` the divergence of `day.mspec` with respect to `night.mspec`, (2) `$D2` the divergence of `night.mspec` with respect to `day.mspec`, and `$D` the symmetric divergence:

```
kl.dist(night.mspec, day.mspec)
$D1
[1] 1.236742

$D2
[1] 0.9032989

$D
[1] 1.07002
```

The **mutual information**, I, is the result of a symbolic analysis as detailed in Sect. 10.1.5.1. The frequency spectra f_i and g_i are transformed into two series of symbols, S_{f_i} and S_{g_i}, respectively, following an alphabet of five letters encoding for an increase, a decrease, a peak, a trough, or a flat region in the frequency spectrum. The absolute frequency of each symbol is then computed and used to compute a level of entropy referring to the usual formula $H = -\sum_{i=1}^{n} p_i \log p_i$. The mutual information combines the entropy of each frequency spectrum and the joint entropy S_{fg_i} obtained when concatenating the two frequency spectra:

$$I = H_{S_f} + H_{S_g} - H_{S_{fg}}$$

We have seen in Sect. 10.1.5.1 that the function `symba()` computes the entropy of one frequency spectrum transformed into a series of symbols. If we provide two frequency spectra to `symba()`, then the function calculates the entropy corresponding to each frequency spectrum and the mutual information between the two frequency spectra, stored in the element `$I`. To obtain a dissimilarity measure, we only need to compute $1 - I$ as shown here between `night.left` and `day.left` sounds:

```
res <- 1-symba(round(night.mspec[,2],2), round(day.mspec[,2],2))$I
res
[1] 0.8144348
```

The **log-spectral** distance, D_{LS}, is a symmetric distance based on the logarithm of the ratio of x_i and y_i:

$$D_{LS} = \sqrt{\sum_{i}^{n} 10 \times \log_{10}\left(\frac{x_i}{y_i}\right)^2}$$

The use of the function `logspec.dist()` is elementary:

```
logspec.dist(night.mspec, day.mspec)
[1] 76.72361
```

The distance can be scaled by the length of the spectra with the argument `scale`:

```
logspec.dist(night.mspec, day.mspec, scale=TRUE)
[1] 4.795226
```

The **relative frequency similarity**, S, is an index expressed in % based on minimum and maximum of each f_i and g_i frequency bin:

$$S = \frac{100}{n} \times \sum_{i=1}^{n} \frac{\min_i(f_i, g_i)}{\max_i(f_i, g_i)}$$

The same process can be applied to the probability mass functions, x and y:

$$S = \frac{100}{n} \times \sum_{i=1}^{n} \frac{\min_i(x_i, y_i)}{\max_i(x_i, y_i)}$$

A dissimilarity metric can be obtained by computing $100 - S$. Here, with the function `simspec()`, we compute such a dissimilarity on the raw spectra and then on the probability mass functions with `PMF=TRUE`:

```
100-simspec(night.mspec, day.mspec)
[1] 60.11749
100-simspec(night.mspec, day.mspec, PMF=TRUE)
[1] 56.98964
```

The **Pearson correlation coefficient**, r, can be used to estimate how much two frequency spectra are correlated, that is, similar. The equation of the coefficient for x_i and y_i is:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

However, x_i and y_i are probability mass functions so that we have:

$$\bar{x} = \bar{y} = \frac{1}{n}$$

We can then simplify the equation of r_{xy}:

$$r_{xy} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

A dissimilarity based on Pearson correlation can be derived by computing $\sqrt{1 - r_{xy}}$. There is no dedicated function to compute such a dissimilarity, but this can be solved using the base function `cor()`:

```
sqrt(1-cor(night.mspec[,2]/sum(night.mspec[,2]),
           day.mspec[,2]/sum(day.mspec[,2])))
[1] 0.8513756
```

So far, we worked only on the (mean) frequency spectrum, but distances can also work in the time domain. In that case, the sounds to be compared should be perfectly synchronized to ensure a perfect homology between the items to be compared. This is the case between the left and right channel of a stereo recording as found in each recording stored in the directory `guiana` but not between `day.left` and `night.left` as the recordings were not taken at the same time.

The **temporal dissimilarity index**, D_t, is similar to the spectral dissimilarity index D_f except that it compares the absolute difference between Hilbert envelopes instead of the absolute difference between mean spectra. If we have two signals, s_1 and s_2, the D_t is computed according to:

$$D_t = \frac{\sum_{i=1}^{n} |s_{1_i} - s_{2_i}|}{2}$$

with $D_t \in [0, 1]$

The function `diffenv()`, which computes D_t, accepts sounds as input and has arguments to optionally smooth the envelope (see Sect. 5.2.3). Here is the command to compare the left and right channels of `day`:

```
diffenv(day@left, day@right, f=day@samp.rate)
[1] 0.3592248
```

Both spectral and temporal indices can be combined in a single index, D:

$$D = D_f \times D_t$$

with $D_t \in [0, 1]$. The function to obtain D is `diffwave()`:

```
diffwave(day@left, day@right, f=day@samp.rate)
[1] 0.05598203
```

The time domain can also be considered as the component of a STDFT matrix. Two STDFT can be compared through a matrix distance method. For instance, the RV correlation coefficient, developed by Robert and Escoufier (1976), can be defined for two matrices X and Y centered by columns as (Josse et al. 2008):

$$RV = \frac{tr(XX^\intercal YY^\intercal)}{\sqrt{tr(XX^\intercal{}^2)tr(YY^\intercal{}^2)}}$$

where $tr(X)$ is the trace of the matrix X, that is, the sum of the elements found on the main diagonal of X, and where $X^\intercal$ denotes the transpose of X. The RV coefficient can take values between 0 for total uncorrelation between X and Y and 1 when X and Y are identical or when X and Y only differ by a scaling factor.

The RV coefficient can be computed, thanks to the function `coeffRV()` of the package `FactoMineR`:

```
library(FactoMineR)
```

The STDFT matrices including linear real coefficients normalized to 1 are obtained with `spectro()` as explained in Sect. 11.7.1.4. Finally, the RV coefficient is retrieved by selecting the element `$rv`. The dissimilarity value is obtained by computing $1 - RV$:

```
f <- day@samp.rate
left.stft <- spectro(day@left, f=f, dB=NULL, plot=FALSE)$amp
right.stft <- spectro(day@right, f=f, dB=NULL, plot=FALSE)$amp
res <- 1-coeffRV(left.stft, right.stft)$rv
res
[1] 0.1351937
```

16.2.2 Batch Processing: How to Obtain and Analyze a Matrix of β Indices

Obtaining an index for a single pair of sounds is obviously not satisfactory as audio databases usually include hundreds or thousands of files to be compared. Files have to be compared pair by pair in such a way that a dissimilarity matrix can be built. A dissimilarity matrix for n sounds s_i contains $n \times n$ D distances. The matrix is a square and symmetric matrix. The main diagonal is the repetition of a single value, the minimum of the dissimilarity matrix, in most cases 0:

$$\begin{array}{c} \\ s_1 \\ \vdots \\ s_i \\ \vdots \\ s_n \end{array} \begin{array}{c} \begin{array}{ccccc} s_1 & \dots & s_i & \dots & s_n \end{array} \\ \begin{pmatrix} D_{11} & \dots & D_{1i} & \dots & D_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ D_{i1} & \dots & D_{ii} & \dots & D_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ D_{n1} & \dots & D_{ni} & \dots & D_{nn} \end{pmatrix} \end{array}$$

with $\{D_{11}, \dots, D_{ii}, \dots, D_{nn}\} = 0$

In this section, we will see how to generate such a matrix and how to visualize and treat it with a clustering method and a statistical procedure. We will refer to the 24 files recorded in French Guiana and stored in the subdirectory `guiana` of the directory `sample`:

```
oldwd <- getwd()                # save the current directory path
setwd("sample/guiana")          # change the current directory
files <- dir(pattern = "wav$") # get the .wav file names
n <- length(files)              # number of files
n
[1] 24
```

The time of recording (hour) is saved in an object `hour` to be used as a factor:

```
hour <- songmeter(files)$hour
hour
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
[19] 18 19 20 21 22 23
```

16.2.2.1 Generation of the Distance Matrix

The prelude to the matrix generation consists in obtaining the raw data, that is, n frequency spectra, one for each of the n files. Here we will compute the mean frequency spectrum of each file using the function `meanspec()` with a Hanning window (default), a window including 512 samples (`wl=512`) and no overlap between successive windows (`ovlp=0`, default) (see Sect. 11.14). To do that we first create a matrix of dimension $n \times wl \div 2$ containing only NA values. We then apply a `for` loop calling `meanspec()` and `readWave()`. Only the second column of the value of `meanspec()` is kept as it contains the amplitude value of the frequency spectrum:

```
wl <- 512
mspectra <- matrix(NA, nrow=wl/2, ncol=n)
for(i in 1:n) mspectra[,i] <- meanspec(readWave(files[i]),
                                       wl=wl, plot=FALSE)[,2]
```

We can check that everything went well using `str()`:

```
str(mspectra)
 num [1:256, 1:24] 0.0678 0.0602 0.0482 0.0499 0.0539 ...
```

Now the challenge is to compare these 24 mean spectra by pair. It is therefore necessary to compare the mean spectrum 1 with the mean spectra 2, 3, ..., 24, then the mean spectrum 2 with the mean spectra 3, 4, ..., 24, then the mean spectrum 3 with the mean spectra 4, 5, ..., 24, and so on up to have $((24 \times 23) \div 2) - 24 = 276$ pair-wise combinations.

The number defining each pair, that is, $\{(1, 2), (1, 3), \ldots, (23, 24)\}$, can be obtained with the base function `combn()`:

```
comb <- combn(1:n, 2)
ncol(comb)
[1] 276
comb[, 1:25]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    1    1    1     1
[2,]    2    3    4    5    6    7    8    9   10    11
     [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19]
[1,]     1     1     1     1     1     1     1     1     1
[2,]    12    13    14    15    16    17    18    19    20
     [,20] [,21] [,22] [,23] [,24] [,25]
[1,]     1     1     1     1     2     2
[2,]    21    22    23    24     3     4
```

We now open a new matrix `m` of dimension $(n \times n)$ where the dissimilarity distances will be stored. We compute the cumulative spectral difference D_{cf} index between each pair of mean spectra with a loop `for`:

```
m <- matrix(NA, nrow=n, ncol=n)
for(i in 1:ncol(comb)){
m[comb[2,i], comb[1,i]] <- diffcumspec(mspectra[,comb[1,i]],
                                       mspectra[,comb[2,i]],
                                       plot=FALSE)
}
```

For a sake of clarity, we use the names of the audio files as column and row names. The function `strplit()` is called to remove the useless file extension `.wav`:

```
colnames(m) <- rownames(m) <- unlist(strsplit(files, split=".wav"))
```

We check the first 16 elements of the matrix `m`:

```
m[1:4,1:4]
                     M-XV_20101125_000000
M-XV_20101125_000000                   NA
M-XV_20101125_010000          0.007904314
```

(continued)

```
M-XV_20101125_020000          0.009130246
M-XV_20101125_030000          0.026090174
                     M-XV_20101125_010000
M-XV_20101125_000000                   NA
M-XV_20101125_010000                   NA
M-XV_20101125_020000          0.004061756
M-XV_20101125_030000          0.018414709
                     M-XV_20101125_020000
M-XV_20101125_000000                   NA
M-XV_20101125_010000                   NA
M-XV_20101125_020000                   NA
M-XV_20101125_030000           0.01770413
                     M-XV_20101125_030000
M-XV_20101125_000000                   NA
M-XV_20101125_010000                   NA
M-XV_20101125_020000                   NA
M-XV_20101125_030000                   NA
```

Only the lower triangle of the matrix is filled so we need to make the matrix symmetric. To achieve this, we use the base function `upper.tri()` to select the upper triangle that we fill with the upper triangle of the transposed matrix:

```
m[upper.tri(m)] <- t(m)[upper.tri(m)]
m[1:4,1:4]
                     M-XV_20101125_000000
M-XV_20101125_000000                   NA
M-XV_20101125_010000          0.007904314
M-XV_20101125_020000          0.009130246
M-XV_20101125_030000          0.026090174
                     M-XV_20101125_010000
M-XV_20101125_000000          0.007904314
M-XV_20101125_010000                   NA
M-XV_20101125_020000          0.004061756
M-XV_20101125_030000          0.018414709
                     M-XV_20101125_020000
M-XV_20101125_000000          0.009130246
M-XV_20101125_010000          0.004061756
M-XV_20101125_020000                   NA
M-XV_20101125_030000          0.017704135
                     M-XV_20101125_030000
M-XV_20101125_000000           0.02609017
M-XV_20101125_010000           0.01841471
M-XV_20101125_020000           0.01770413
M-XV_20101125_030000                   NA
```

We end up by replacing the NA values on the main diagonal by 0 values so that the matrix is ready for further analyses:

```
diag(m) <- 0
m[1:5,1:5]
                      M-XV_20101125_000000
M-XV_20101125_000000          0.000000000
M-XV_20101125_010000          0.007904314
M-XV_20101125_020000          0.009130246
M-XV_20101125_030000          0.026090174
M-XV_20101125_040000          0.037985776
                      M-XV_20101125_010000
M-XV_20101125_000000          0.007904314
M-XV_20101125_010000          0.000000000
M-XV_20101125_020000          0.004061756
M-XV_20101125_030000          0.018414709
M-XV_20101125_040000          0.030688195
                      M-XV_20101125_020000
M-XV_20101125_000000          0.009130246
M-XV_20101125_010000          0.004061756
M-XV_20101125_020000          0.000000000
M-XV_20101125_030000          0.017704135
M-XV_20101125_040000          0.029312122
                      M-XV_20101125_030000
M-XV_20101125_000000           0.02609017
M-XV_20101125_010000           0.01841471
M-XV_20101125_020000           0.01770413
M-XV_20101125_030000           0.00000000
M-XV_20101125_040000           0.01422946
                      M-XV_20101125_040000
M-XV_20101125_000000          0.03798578
M-XV_20101125_010000          0.03068820
M-XV_20101125_020000          0.02931212
M-XV_20101125_030000          0.01422946
M-XV_20101125_040000          0.00000000
```

16.2.2.2 Visualization

A solution to visualize the results is to plot the dissimilarity matrix as a false color plot, or heatmap, where the dissimilarity distance obtained for each pair of sound is plotted as a square of color following a linear color scale. We utilize the function `image()` as already done when building by hand a spectrogram (see Sects. 11.4 and 12.1.2.1) together with the `seewave` function `dBscale()` initially implemented to add a dB scale to a spectrogram. For fancy color output, we use the palette `viridis` from the eponymous package. The result seems to indicate three parts according to time that can be grouped into two main periods: a

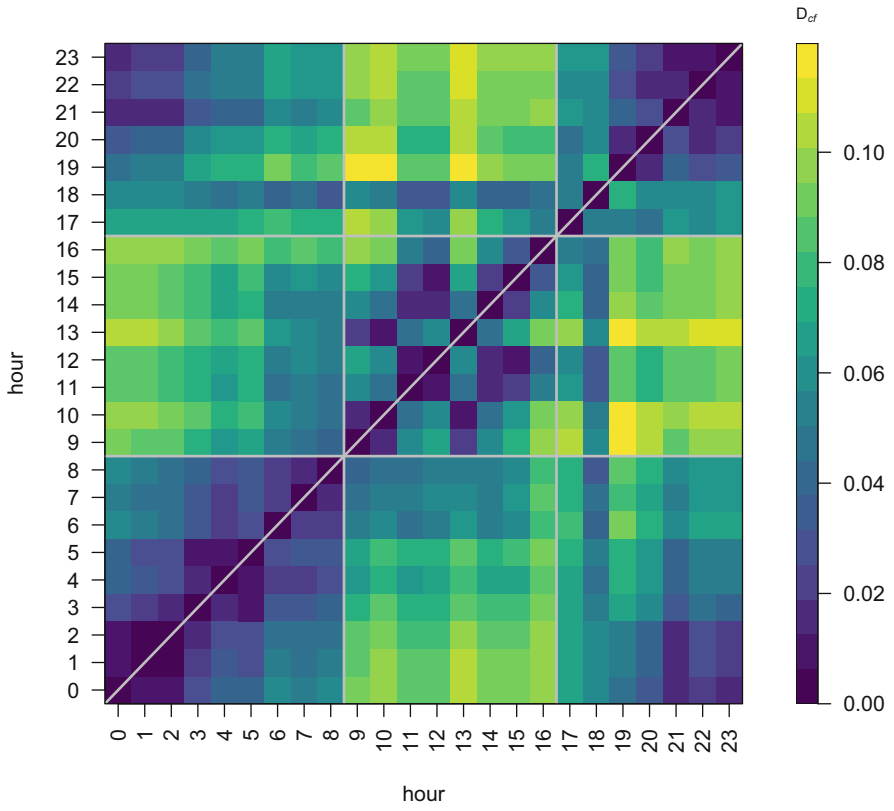


Fig. 16.5 Visualization of a β index matrix with a heatmap. The dissimilarity matrix obtained with the cumulative spectral difference D_{cf} index was plotted as a heatmap using the function `image()`. The scale on the left was produced by taking advantage of the function `dBscale()` used to add a dB scale to a spectrogram. Gray lines were added manually with `abline()`

night period between 17:00 and 08:00 and a day period between 09:00 and 16:00 (Fig. 16.5):

```
library(viridis)
col <- viridis(20)
layout(matrix(1:2, nrow=1), width=c(8,1))
image(x=1:n, y=1:n, z=m, col=col, axes=FALSE,
      xlab="hour", ylab="hour")
axis(side=1, at=1:n, labels=hour, las=2)
axis(side=2, at=1:n, labels=hour, las=2)
abline(v=c(8.5,16.5)+1, h=c(8.5,16.5)+1, col="grey", lwd=2)
abline(a=0, b=1, col="grey", lwd=2)
```

(continued)

```
box()
par(mar=c(5.1,0,4.1,3))
dBscale(collevels=seq(0,max(m),length.out=20),
        textlab=bquote(D[italic(cf)]), palette=viridis)
```

16.2.2.3 Clustering

Hierarchical cluster analysis (HCA) is an unsupervised clustering method that groups observations according to an agglomeration method, the most popular being the Ward distance that minimizes the intra-cluster inertia (variance) and maximizes the intercluster inertia. The result is visualized as a dendrogram where closely related observations are grouped by branches. The topology of the dendrogram can help in defining clusters. In R, HCA is made available, thanks to the base function `hclust()` which requires an object of class `dist`. The function `hclust()` has a dedicated `plot()` function that displays the dendrogram. The dendrogram can be cut in a specific number of clusters k by using the function `rect.hclust()`. Here the original dissimilarity matrix m containing the D_{cf} values between the 24 sounds is coerced into a `dist` object with the function `as.dist()`, the HCA is processed with the Ward distance and displayed. Three tries of tree cutting are attempted, with $k = \{2, 3, 4\}$ (Fig. 16.6):

```
d <- as.dist(m)
hc <- hclust(d, method="ward.D")
plot(hc, labels=hour, main="", sub="", xlab="Hour", hang=-1)
rect.hclust(hc, k=2, border="green")
rect.hclust(hc, k=3, border="red")
rect.hclust(hc, k=4, border="blue")
legend("topright", legend=paste("k", 2:4, sep="="),
       lty=1, col=c("green","red","blue"), bty="n")
```

For $k = 2$ we obtain a night period between 19:00 and 8:00 and a day period between 09:00 and 18:00, slightly different from what we could conclude by observing the heatmap (Fig. 16.5). For $k = 3$, we have a first night period between 19:00 and 02:00, a second night period extending to the morning between 03:00 and 08:00, and a day period between 09:00 and 18:00. With $k = 4$, the day period is split into two periods including discontinuous hours: 09:00, 10:00, and 13:00 in one and 11:00, 12:00, 14:00, 15:00, 16:00, 17:00, and 18:00 in the other one.

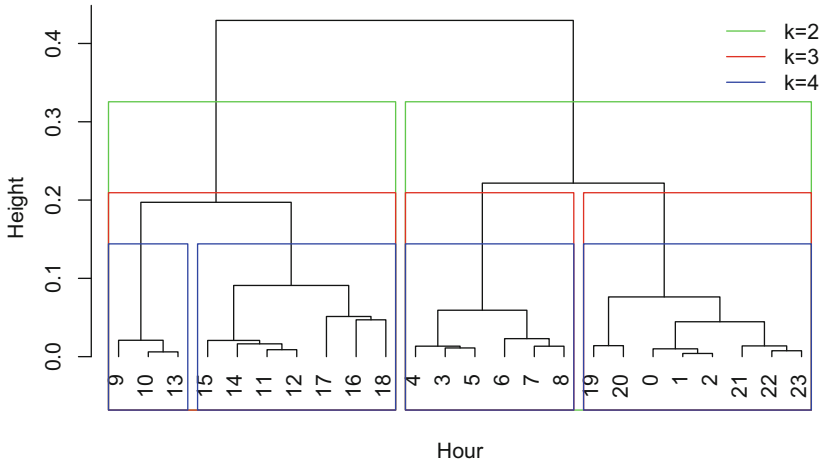


Fig. 16.6 Visualization of a β index matrix with a hierarchical cluster analysis dendrogram. The dissimilarity matrix obtained with the cumulative spectral difference D_{cf} was treated with hierarchical cluster analysis, the result being plotted as a dendrogram. The color rectangles show how to cut the dendrogram in 2, 3, or 4 clusters

16.2.2.4 Ordination Analysis

Ordination methods are descriptive analyses that consist in extracting the main trends of a set of variables in the form of continuous and orthogonal axes. Unconstrained ordination methods include, among others, the very popular principal component analysis (PCA) and correspondence analysis (CA). These unconstrained methods are mainly graphical methods based on scatterplots that are not supported with a statistical test. Constrained, or canonical, ordination procedures compute the axes under the constraint of explanatory variables. These constrained methods are also visual methods but can be accompanied with a statistical test.

The redundancy analysis (RDA) is a constrained ordination technique which aims at modeling a multivariate response data, e.g., a matrix **Y** made of several observations (lines) described by several variables (columns) by a set of explanatory variables organized in a matrix **X**. The RDA is a kind of PCA which axes are constrained by a linear combination of the explanatory variables. In practice, a RDA consists in a multivariate multiple linear regression (MLR) followed by a PCA applied on the table of the fitted values returned by the regression (Borcard et al. 2011). The null hypothesis H_0 of absence of linear relationship between **Y** and **X** can be tested.

The RDA has been extended to the analysis of distance metrics such that the response matrix **Y** takes the form of a distance matrix (Legendre 1999). The very first step of the method, named distance-based redundancy analysis (db-RDA), is a principal coordinate analysis (PCoA), an ordination procedure that takes a distance matrix as input. The relationship of the principal coordinates with the explanatory

variables stored in $\mathbf{X}$ is then treated with a usual RDA. The importance of the explanatory variables in modeling the response variables can be tested through a permutation test. The db-RDA has proved to be useful in the analysis of tropical acoustic communities and soundscapes (Gasc et al. 2013b; Rodriguez et al. 2014). In `R`, the db-RDA is available in the package `ade4` with the sequential use of the functions `dudi.pco()` for computing the PCoA and `pcaiv()` for the application of a PCA with respect to instrumental variables.

We first load `ade4` and its graphical companion `adegraphics`:

```
library(ade4)
library(adegraphics)
```

We compute the PCoA taking care of transforming the dissimilarity matrix as quasi-Euclidean distance matrix with the help of the function `quasieuclid()`. The argument `scannf` is turned to `FALSE` so that the barplot of the eigenvalues is not produced:

```
d <- as.dist(m)
pcoa <- dudi.pco(quasieuclid(d), scannf=FALSE)
```

We prepare two sets of factors to be tested with the db-RDA: (1) a first factor named `hour` corresponding to each hour of recording and (2) a second factor, named `period`, corresponding to the time periods of the night and day cycle as highlighted by the HCA with $k = 3$ (Fig. 16.6), i.e., with a first night period between 19:00 and 02:00, a second night period extending to the morning between 03:00 and 08:00, and a day period between 09:00 and 18:00.

So for the hour, the factor is obtained as above with:

```
hour <- songmeter(files)$hour
```

The three periods can be built by filling a vector of length $n = 24$ according to the object `hour` we have just created:

```
period <- rep(NA, n)
period[hour >= 19 | hour < 3] <- "night"
period[hour >= 3 & hour < 9] <- "morning"
period[hour >= 9 & hour < 19] <- "day"
```

Both objects need to be coerced as factor objects:

```
hour <- as.factor(hour)
hour
 [1] 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
[19] 18 19 20 21 22 23
24 Levels: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 23
period <- as.factor(period)
period
 [1] night   night   night   morning morning morning morning
 [8] morning morning day     day     day     day     day
[15] day     day     day     day     day     night   night
[22] night   night   night
Levels: day morning night
```

We compute the RDA on the PCoA with the function `pcaiv()` that runs a PCA with respect to instrumental variables, here with respect to `hour` and `period`, respectively, so that we end up with two db-RDA:

```
rda.hour <- pcaiv(pcoa, df=hour, scannf=FALSE)
rda.period <- pcaiv(pcoa, df=period, scannf=FALSE)
```

The last action consists in plotting with the `adegraphics` function `s.class()` the projection of the observations grouped according to the factor levels in the space defined by the two first axes of the db-RDA. The projections of the observations on the axes of the db-RDA are stored in the item `$ls`. In the case of the `hour` level, each level contains a single observation. The graphic undeniably shows that hours are not randomly positioned but that they draw a circular pattern suggesting that the acoustic communities of the tropical forest change according to the 24 h cycle (Fig. 16.7):

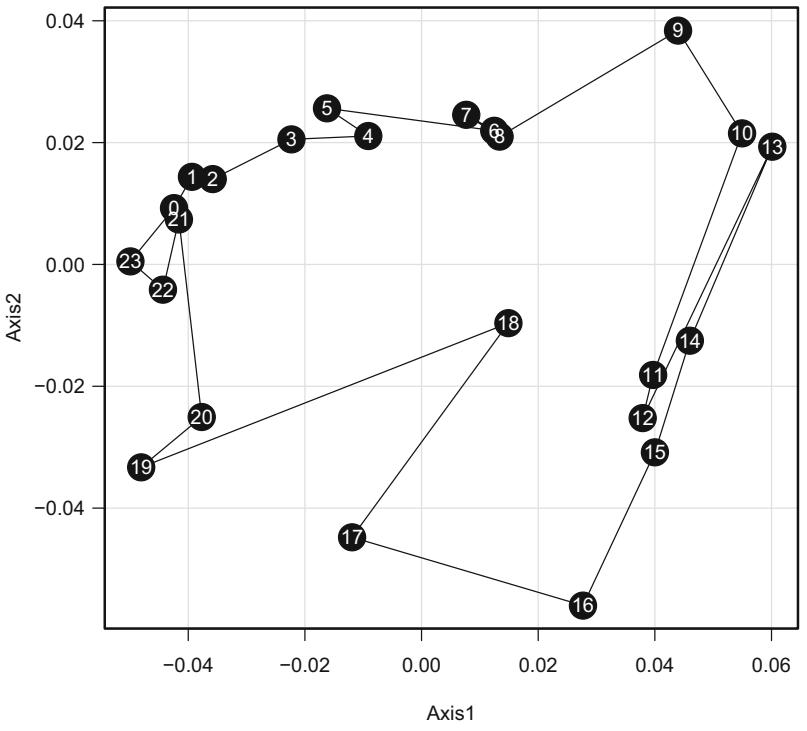
```
s.class(rda.hour$ls, fac=hour)
```

The function `s.class()` uses the package `lattice` that, as `ggplot2` does, implements a new way to create graphics. The `lattice` and the base graphic systems cannot be mixed so that a new grammar should be learned to tune the `ade4` graphics. Another solution is to retrieve the data describing the positions of the factor levels and to manually build a new graphic as illustrated in the DIY box 16.1.

DIY 16.1 — How to tune the visualization of a db-RDA projection

The projection of the observations treated with a dB-RDA can be manually constructed using `R` base graphical functions. In the following the raw data are stored in an object `res` and then plotted with classical functions including `polypath()` that draws a path between points which coordinates are kept in a two-column matrix:

```
res <- rda.hour$ls                          # raw data
plot(res, type="n", las=1)                  # empty plot
grid(lty=1)                                 # x-y grid
polypath(res)                               # path
points(res, pch=19, cex=3, col="black") # black points
text(res, label=0:23, col="white")      # white labels
box(lwd=2)                                  # frame width
```



The `adegraphics` scatterplot returned by `s.sclass()` for the `period` factor is much more interesting. The points are grouped according to the levels of the factor used to compute the db-RDA. For each level, an ellipse is drawn with its center and axes. The center of the ellipse is the centroid of each group, and an arrow connects the centroid to each point drawing a kind of star. The ellipses are not

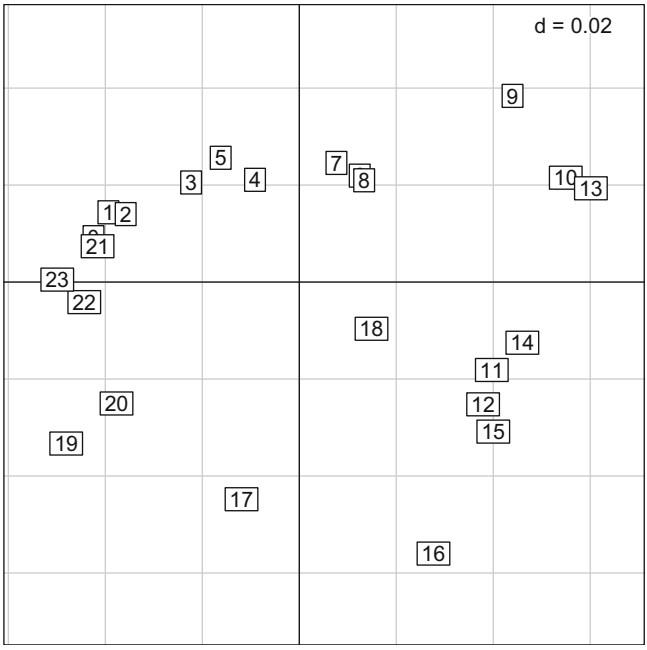


Fig. 16.7 Visualization of a β index matrix with a db-RDA projection according to hour. The β index matrix was treated with a distance-based redundancy analysis, and the observations were projected with `s.class()` in the space defined by the two first axes of the ordination process. Each observation is one factor level, i.e., there is a single observation per factor level

true confidence areas, but they outline an area where a defined percentage p of the observations are expected to be found. This percentage p obeys to the formula:

$$p = 100 \times \left(1 - e^{-\frac{S^2}{2}}\right)$$

where S is the size of the ellipse in the space of the scatterplot. This can be translated in `R` with the function:

```
p <- function(S) {100*(1-exp(-S^2/2))}
```

Setting $S = 1.5$ or $S = 2.5$ returns a percentage of ≈67.5% and ≈95.6 respectively:

```
p(1.5)
[1] 67.53475
p(2.5)
[1] 95.60631
```

If we inspect the documentation of `s.class()`, we see that the argument `ellipseSize` is the size of the ellipse S, with the default value `ellipseSize=1.5`. The following command uses the function `s.class()` with all default values that is with 67.5% ellipses (Fig. 16.8)

```
s.class(rda.period$ls, fac=period)
```

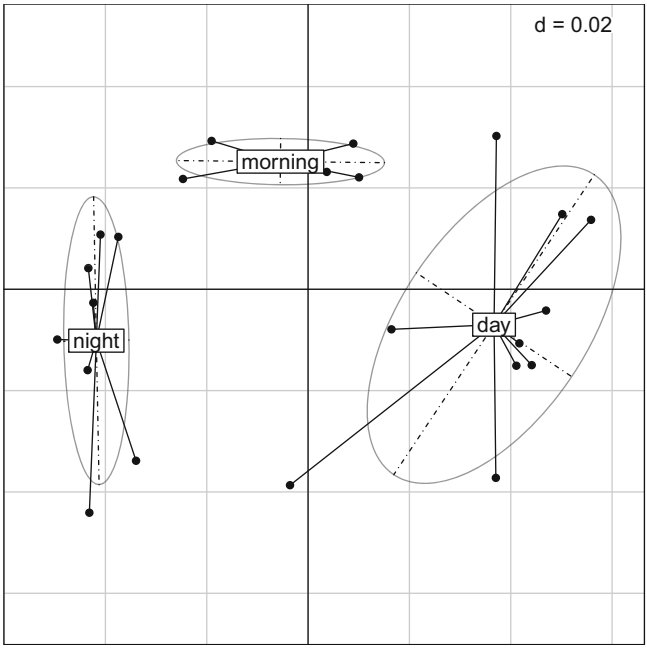


Fig. 16.8 Visualization of a β index matrix with a db-RDA projection according to time periods. The β index matrix was treated with a distance-based redundancy analysis, and the observations were projected with `s.class()` in the space defined by the two first axes of the ordination process. Each observation is grouped according to a factor with the three levels: morning, day, and night. Ellipses would include 67.5% of the observations

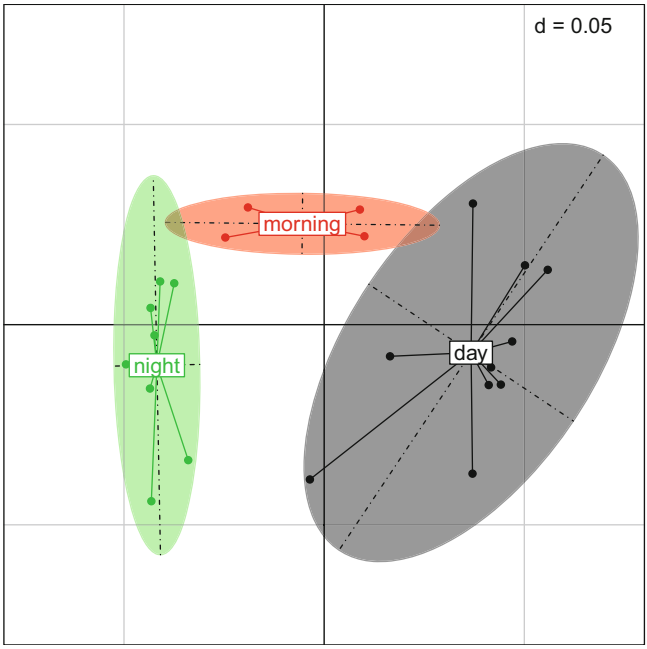


Fig. 16.9 Tuned visualization of the β index matrix with a db-RDA projection according to time periods. This is another version of the graphic displayed in Fig. 16.8 but tuned by modifying some arguments of `s.class()`. In particular the ellipses would here cover 95% of the observations

The display can be tuned by playing with some graphical parameters and by increasing the size of the ellipses, here to reach 95% ellipses (Fig. 16.9):

```
s.class(rda.period$ls, fac=period, col=1:3,
        ellipseSize=2.5, xlim=c(-0.08,0.08), ylim=c(-0.08,0.08))
```

The great advantage of the db-RDA is that a statistical test can be run to objectively interpret the visualization. The null hypothesis H_0 of the test is the absence of a linear relationship between the explanatory variables $\mathbf{X}$ and the projection of the observations $\mathbf{Y}$ along the ordination axes. The test is based on a permutation of the lines of the table that describes the position of the observations in the ordination space. The statistic tested is the coefficient R^2 of the multiple linear regression.

The permutation test can be run with the `ade4` generic function `randtest()`. The number of permutations is specified with the argument `nrepet`. Testing the factor `hour` would be a nonsense as there is a single observation per factor level. In the following, we test the effect of the factor `period`; it is highly significant with $p \approx 0.0009$.

```
test.period <- randtest(rda.period, nrepet=1000)
test.period
Monte-Carlo test
Call: randtest.pcaiv(xtest = rda.period, nrepet = 1000)

Observation: 0.6049304

Based on 1000 replicates
Simulated p-value: 0.000999001
Alternative hypothesis: greater

    Std.Obs Expectation    Variance
9.004050978 0.091154854 0.003255899
```

and we plot the result of the permutation using `plot()` (Fig. 16.10):

```
plot(test.period)
```

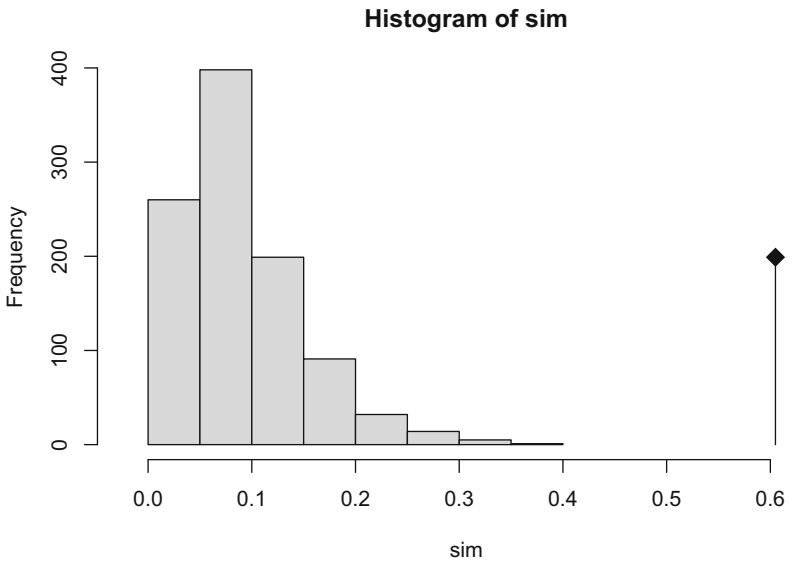


Fig. 16.10 Visualization of the db-RDA permutation test. The histogram shows the distribution of the statistic obtained by permutation when H_0 is true. The statistic observed for the data tested is depicted as a diamond placed on the top of a segment. The p-value of the test is the probability to obtain a statistic greater than the statistic observed, that is, the surface of the histogram on the right of the diamond, here $p \approx 0.0009$

Chapter 17
Comparison and Automatic Detection

The comparison of two sounds is an exercise often practiced when estimating differences among recording devices, broadcasting systems, instruments, singers, music genres, and in the special case of ecoacoustics, animal populations, animal communities, and soundscapes. The comparison of two objects may appear as a simple task as we commonly do this sort of operation in our everyday life. However, to spot the difference between two sounds is not that an easy game for a machine. The first solution can be descriptive: the dissimilarity of two sounds is estimated through a statistical comparison of their time, frequency, and amplitude features (see Chaps. 7, 8, and 10). The second solution can consist in using the β dissimilarity indices developed to assess global differences between pairs of sounds (see Sect. 16.2). A third solution involves algorithms that compare time series.

In this chapter, we will introduce the cross-correlation, the frequency coherence, and the dynamic time warping. We will also see how to proceed a supervised binary automatic detection.

17.1 Cross-Correlation

The cross-correlation of two discrete time series, $x[n]$ and $y[n]$, consists in computing a measure of similarity between the $x[n]$ and a delayed version of $y[n]$ that is between $x[n]$ and $y[n+m]$, where $m \neq 0$ is a positive or negative time lag, usually set to one audio sample (Fig. 17.1).[1] The cross-correlation can be employed to automatically identify a sound of reference (template) in a test file as detailed in Sect. 17.4.

[1] Cross-correlation is related to convolution (see Sect. 14.6.1) and autocorrelation (see Sect. 13.1.2.1).

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_17

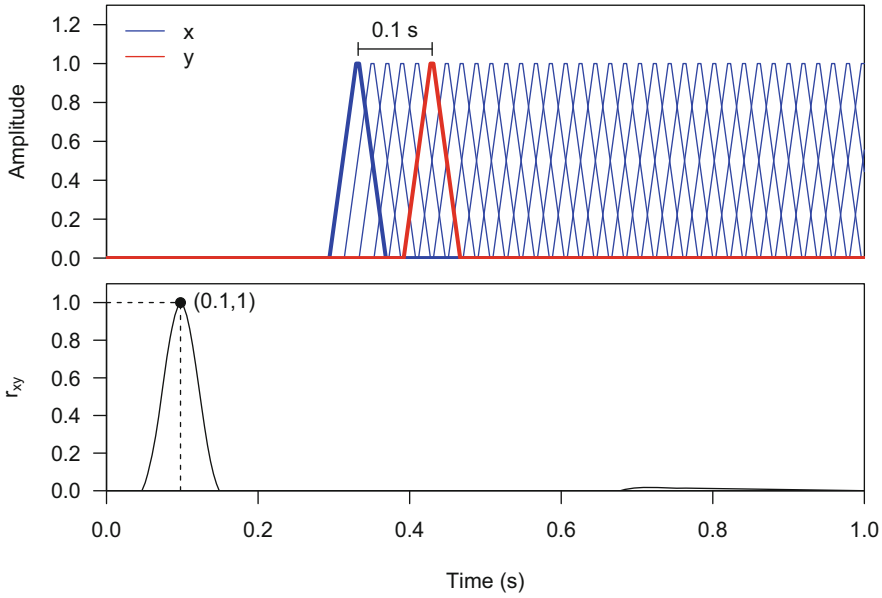


Fig. 17.1 Cross-correlation principle. Cross-correlation mainly consists in moving forward and backward a series along another series and in computing a correlation coefficient at each m lag step. In this graphic, the blue x series is moved forward ($m > 0$) along the red series y (top) generating a time series of the correlation coefficient r_{xy} (bottom). The correlation time series shows a peak for a lag of 0.1 s indicating that the two series are shifted by 0.1 s. The backward movement ($m < 0$) is not shown for a sake of clarity

Any similarity function can be used as a correlation measure. The most common similarity function, named normalized correlation coefficient $r_{xy}[m]$, is obtained according to:

$$r_{xy}[m] = \frac{\sum_{n=1}^{N-m} x[n]y[n+m]}{\sqrt{\sum_{n=1}^{N-m} x^2[n] \sum_{n=1}^{N-m} y^2[n]}}$$

with $r_{xy} \in [0, 1]$.

The result of the cross-correlation is a time series made of the successive r_{xy} values. If m corresponds to a single audio sample, then the time series of the cross-correlation r_{xy} has the same sample frequency than $x[n]$ and $y[n]$. Note that the cross-correlation function is asymmetric such that $r_{xy} \neq r_{yx}$.

Cross-correlation method is not limited to time series. It can be applied to frequency spectra where the spectra are slid against each other according to frequency. It can also be adapted to compare STDFT matrices according to time.

The main advantages of the cross-correlation is to obtain an estimation of the potential shift, or offset, that may occur between the objects to be compared and to be potentially supported by a statistical test. For instance, the two times series

compared in Fig. 17.1 have a similar shape but are delayed by a lag of 0.1 s. The r_{xy} function shows indeed a peak for $m = 0.1$ s, clearly identifying the delay between the two time series.

To illustrate the use of cross-correlation, we will compare the second and the third notes of `tico` extracted with `cutw()` and stored in two `Wave` objects, `note2` and `note3`:

```
note2 <- cutw(tico, from=0.5, to=0.9, output="Wave")
note3 <- cutw(tico, from=0.9, to=1.3, output="Wave")
```

The two objects to be compared should have exactly the same length. This condition can be tested with:

```
length(note2@left)==length(note3@left)
[1] TRUE
```

Cross-correlation for time series is available in the base function `ccf()`. We first need to coerce the objects as `ts` objects:

```
note2.ts <- ts(note2@left, start=0, end = duration(note2),
               frequency=note2@samp.rate)
note3.ts <- ts(note3@left, start=0, end = duration(note3),
               frequency=note3@samp.rate)
```

We then use directly the function `ccf()` setting the maximum lag m as half the number of samples:

```
res <- ccf(note2.ts, note3.ts, lag.max=length(note2@left)/2,
           plot=FALSE)
```

r_{xy} is stored in the list item `$acf` and the lag m in `$lag`. The best correlation occurs at the maximum of `$acf` so that the estimated delay or offset between the two notes is obtained with:

```
res$lag[which.max(res$acf)]
[1] 0.01718821
```

We can now look at the original waveform and the r_{xy} time series with (Fig. 17.2):

```
layout(matrix(1:3, nc=1))
par(mfrow=c(3,1), mar=c(5,4.5,4,2))
oscillo(note2, bty="o", title="note 2")
oscillo(note3, bty="o", title="note 3")
plot(res$lag, res$acf, type="l", bty="o",
     xaxs="i", yaxs="i", cex.lab=1.5,
     xlab=expression(italic(m)~(s)), ylab=expression(r[xy]),
     main="cross-correlation")
```

`seewave` offers three cross-correlation functions: (1) `corenv()` for the cross-correlation between two amplitude envelopes, (2) `corspec()` for the cross-correlation between two frequency spectra, and (3) `covspectro()` for the cross-correlation (or strictly spoken cross-covariance) between two STDFT matrices. The particularity of these functions is to use a statistical correlation coefficient as a similarity measure (either Spearman, Pearson, or Kendall coefficient) for the amplitude envelope and frequency spectra and the mean of the covariance matrix diagonal between the STDFT matrices.

The function `corenv()` cross-correlates the amplitude envelopes of the two sounds to be compared. By default, the function computes the Hilbert amplitude envelopes, applies the cross-correlation, and plots the correlation function. The use of `corenv()` is rather simple but the time of process can be quite long. A solution to save time consists in smoothing the amplitude envelope by parsing the arguments `msmooth`, `ksmooth`, or `ssmooth` of the function `env()` (see Sect. 5.2.3). In the following we use `msmooth=c(50,75)` that is a moving average with a window of 50 samples and with an overlap of 75% (Fig. 17.3):

```
res <- corenv(note2, note3, msmooth=c(50,75))
```

The maximum reached by the correlation function and the time delay in s are stored in the list items `$rmax` and `$t`:

```
res$rmax
[1] 0.9334181
res$t
[1] 0.01367676
```

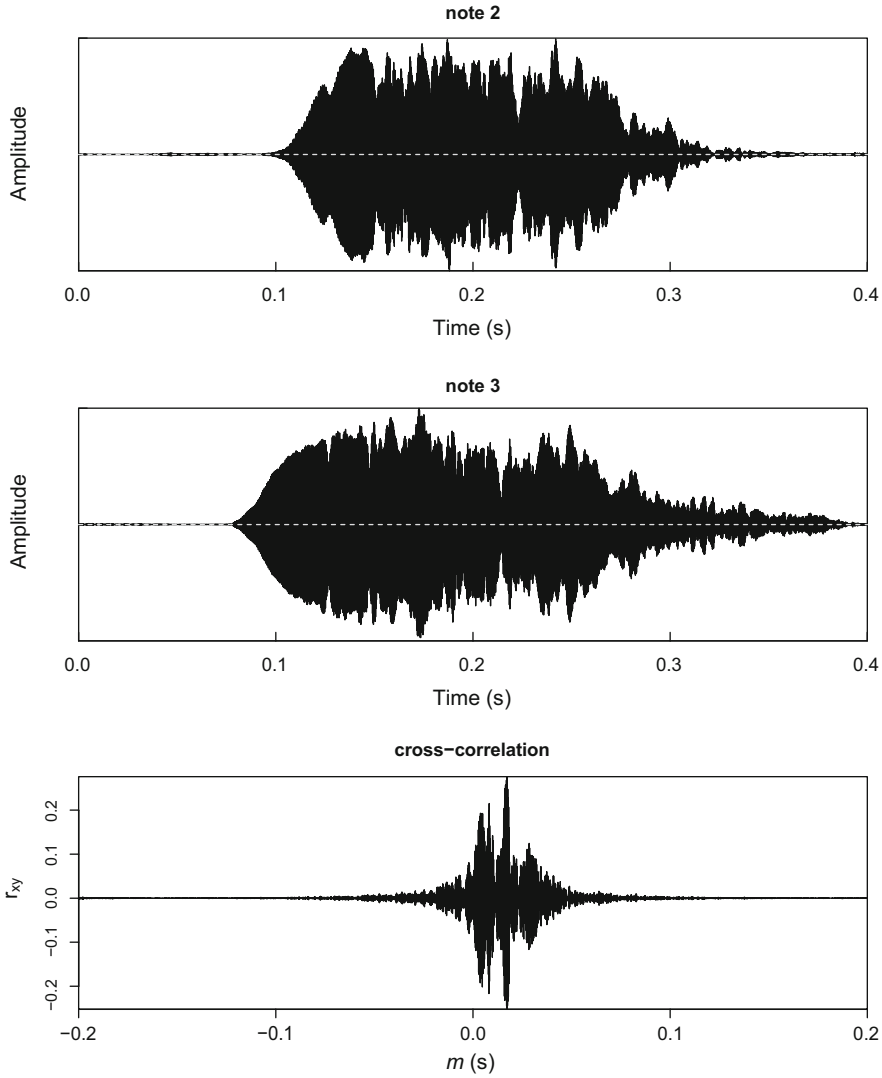


Fig. 17.2 Waveform cross-correlation. The waveform of the second and third notes of `tico` was cross-correlated with the base function `ccf()`. The figure shows the oscillogram of the two notes and the time series of the correlation coefficient $r_{xy}(m)$, where m is the lag in s

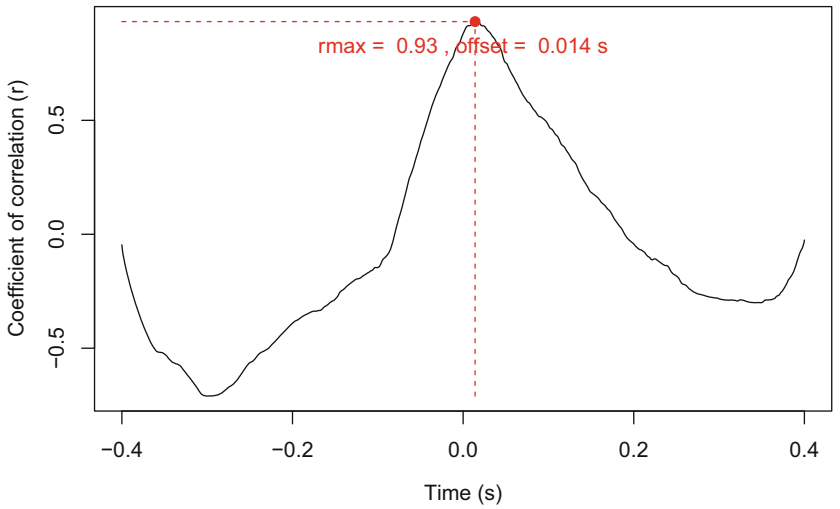


Fig. 17.3 Hilbert amplitude envelope cross-correlation. The Hilbert amplitude envelopes of the second and third note of `tico` were cross-correlated with the function `corenv()`. The cross-correlation indicates a frequency shift, or offset, of 0.014 s

The function `corspec()` works as `corenv()` but takes frequency spectra as input arguments. In the following code, the mean frequency spectra of `note2` and `note3` are computed and displayed, and their cross-correlation is run and shown below (Fig. 17.4):

```
par(mfrow=c(2,1))
note2.mspec <- meanspec(note2, col="blue")
note3.mspec <- meanspec(note3, plot=FALSE)
lines(note3.mspec, col="red")
legend("topright", c("Note 2", "Note 3"), lty=1,
       col=c("blue", "red"), bty="n")
res <- corspec(note2.mspec, note3.mspec)
```

The maximum reached by the correlation function and the frequency shift in kHz are in the list items `$rmax` and `$f`:

```
res$rmax
[1] 0.8362717
res$f
         x
0.2583984
```

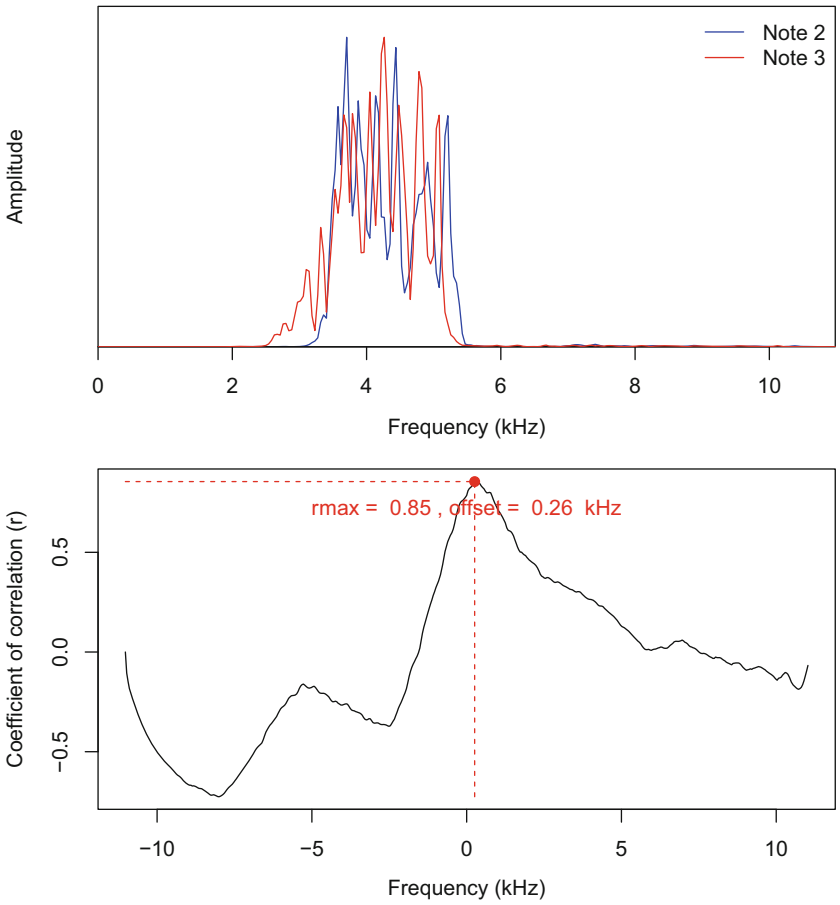


Fig. 17.4 Frequency spectrum cross-correlation. The mean frequency spectra of the second and third note of `tico` were cross-correlated with the function `corspec()`. The cross-correlation indicates a frequency shift, or offset, of 0.26 kHz

The last function, `covspectro()`, computes the STDFT matrices of the two sounds and the covariance between the STDFT matrices at each lag step. The procedure can be quite long. A solution to speed up the computation is to limit the number of covariances computed with the argument `n`, which must be odd. For instance, the following command runs the cross-correlation based on 39 covariances:

```
res <- covspectro(note2, note3, n=39)
```

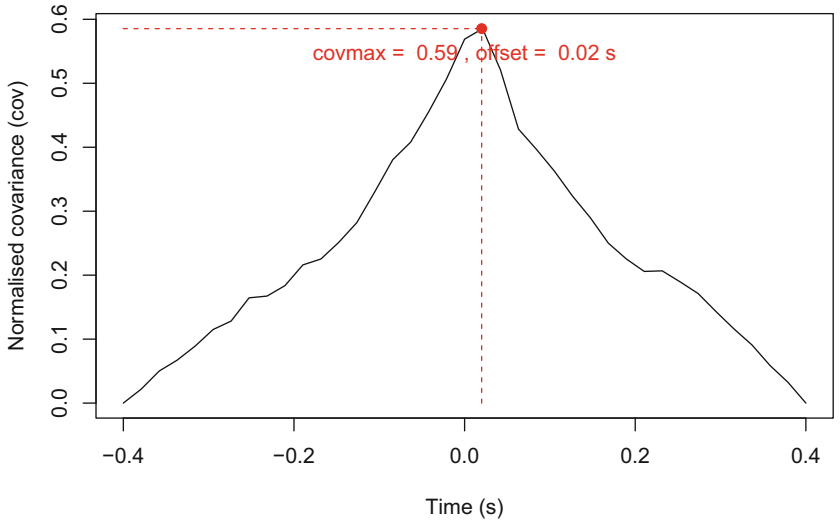


Fig. 17.5 STDFT cross-correlation of STDFT matrices. The STDFT matrices of the second and third note of `tico` were cross-correlated with the function `covspectro()`. The cross-correlation indicates a time shift, or offset, of 0.02 s

The maximum reached by the covariance function and the time delay in s are in the list items `$covmax` and `$t` (Fig. 17.5):

```
res$covmax
[1] 0.5870936
res$t
[1] 0.02000227
```

17.2 Frequency Coherence

Coherence is a frequency function that reveals the degree of a relationship between two signals. The value of the coherence function ranges from 0 to 1. A value of 0 indicates there is no causal relationship between the signals, and a value of 1 points out the existence of a linear frequency response between the two signals. Frequency coherence can be used, for instance, to compare the input and output signals of a system.

The magnitude-squared frequency coherence between two time series $x[n]$ and $y[n]$ is computed according to:

$$C_{xy}[f] = \frac{|G_{xy}[f]|^2}{G_{xx}[f]G_{yy}[f]}$$

where $G_{xy}[f]$ is the crossed power spectral density of $x[n]$ and $y[n]$ which is the Fourier transform of the cross-correlation r_{xy} between $x[n]$ and $y[n]$, and G_{xx} and G_{yy} are the power spectral densities, or auto-spectra, of $x[n]$ and $y[n]$, respectively.

The `seewave` function `coh()` returns the magnitude-squared frequency coherence of two sounds. The function parses the function `spec.pgram()` of the base package `stats`. An example of the coherence could be the comparison of the left and right channel of a single recording. To illustrate this, we select one file among the 24 files recorded in the tropical forest of French Guiana, for instance, the recording obtained at tea time, that is, at 3:00 pm:

```
teatime <- readWave("sample/guiana/M-XV_20101125_150000.wav")
teatime@stereo
[1] TRUE
```

The frequency coherence is here computed for the first 2048 samples (Fig. 17.6):

```
coh(teatime@left[1:2048], teatime@right[1:2048],
    f=teatime@samp.rate)
```

`seewave` includes as well a short-time version of the coherence, named `ccoh()`, which evaluates the variation of the frequency coherence according to time. This produces a spectrogram-like image that shows the variation of coherence according to time. The use of `ccoh()` is direct, and the graphical options are similar to those of the spectrographic function `spectro()` (see Sect. 11.7) (Fig. 17.7):

```
ccoh(teatime@left, teatime@right, f=teatime@samp.rate)
```

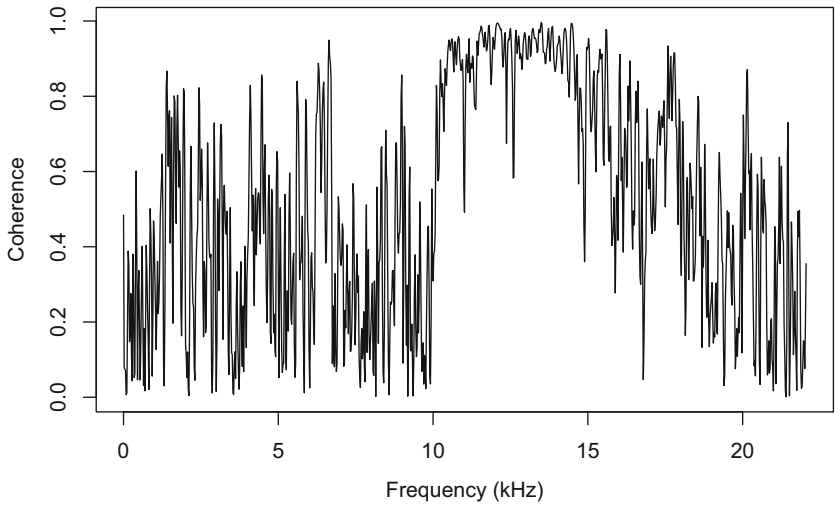


Fig. 17.6 Frequency coherence. Frequency coherence between the left and right channel of a recording achieved at tea time in French Guiana. A value of 1 indicates a pure coherence. Here the coherence is maximum between 10 and 15 kHz

17.3 Dynamic Time Warping

Dynamic time warping (DTW) is a technique introduced by Sakoe and Chiba (1978) to compare two time series that are possibly not in phase or have different duration. DTW algorithms mainly consist in stretching or compressing in a nonlinear way (or warping) the two time series to find their best alignment so that that their similarity is maximized. This can be translated in finding the optimal alignment Φ between two time series $x[n]$ and $y[m]$ with $n = \{1, \ldots, N\}$ and $m = \{1, \ldots, M\}$ in reference to a pair-wise distance d computed between $x[n]$ and $y[m]$ (Giorgino 2009).

The DTW distance D obeys then to the condition:

$$D_{\text{DTW}} = \min_{\Phi} d_{\Phi}(x[n], y[m])$$

The distance d between $x[n]$ and $y[m]$ is usually the Euclidian distance, but other distances as the squared Euclidian, the Manhattan, the Kullback distances can be used (see the `proxy` package for other distances).

The DTW technique has a double interest: (1) DTW returns a similarity distance that could be used for pattern recognition and also as a dissimilarity index (see Sect. 16.2), and (2) DTW shows the correspondence between the points of two series after alignment. The first use of DTW involved the recognition of Japanese spoken words based on time $\times$ frequency series (Sakoe and Chiba 1978). DTW was operated in a vast range of domains including, among others, medical sciences, genetics, image processing, and music. More specifically, in the context of this book,

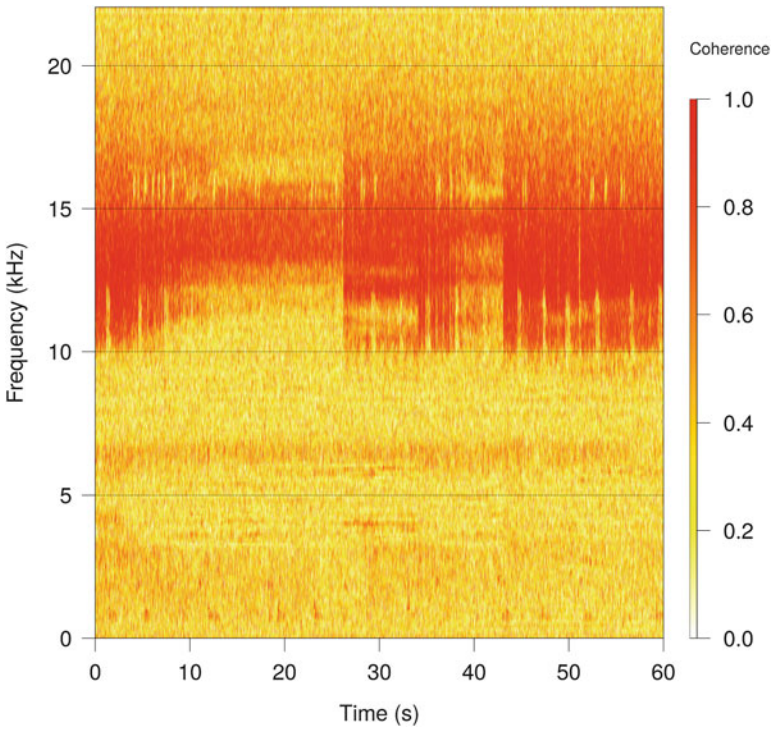


Fig. 17.7 Continuous frequency coherence. The frequency coherence is computed along time using `ccoh()`, a short-term version of `coh()`. Here the function is applied between the left and right channel of the a recording achieved at tea time in French Guiana

DTW can be helpful to compare amplitude envelopes, frequency tracking profiles or contours, and frequency spectra.

The package `dtw` offers all the necessary algorithms to run a nice DTW analysis (Giorgino 2009):

```
library(dtw)
```

Here, we test only the front-end function `dtw()`. The function is facile to use: it basically only requires two vectors to be compared as input and returns an object of class `dtw` that has dedicated plot methods. Other arguments include the choice of the point-wise distance method (argument `dist.method`), the saving of the original series in the returned object (argument `keep`), and the possibility to compute the distance only to save computing time (argument `distance.only`). In the following example, we extract the smoothed Hilbert amplitude envelopes of `note2` and `note3` that we give to `dtw()`. The result is displayed with `plot()`

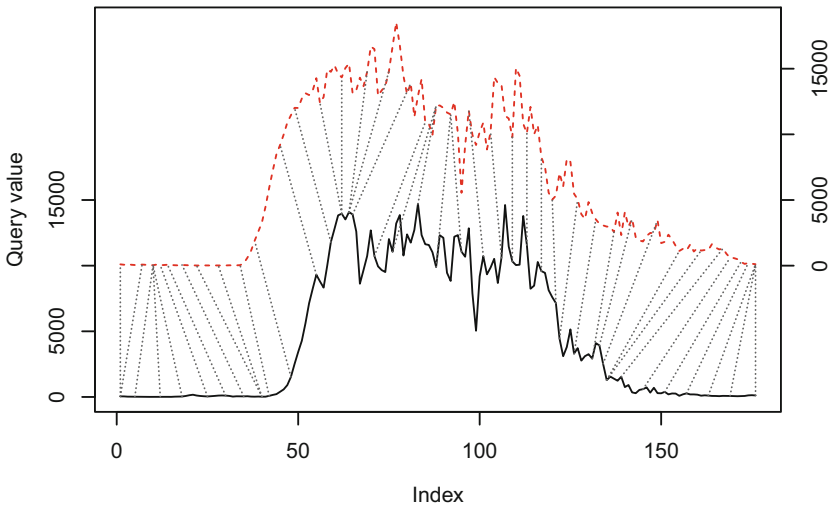


Fig. 17.8 Dynamic time warping on Hilbert amplitude envelope. The smoothed Hilbert amplitude envelopes of `note2` and `note3` of `tico` are compared using dynamic time warping alignment. Note that the envelopes here have the same length (176 samples) but that their length could differ. The dotted gray lines connect the samples that match following the best alignment found by the algorithm

specifying the type of plot (argument `type`), the vertical offset between the two series (argument `offset`), and the number of matching lines between the two series (argument `match.indices`) (Fig. 17.8):

```
note2.env <- env(note2, msmooth=c(50,0), plot=FALSE)
note3.env <- env(note3, msmooth=c(50,0), plot=FALSE)
res <- dtw(note2.env, note3.env, keep=TRUE)
plot(res, type="two", offset=10000,
     match.indices=length(note2.env)/4)
```

The DTW distance D is stored in the list item `$distance`:

```
res$distance
[1] 134790.6
```

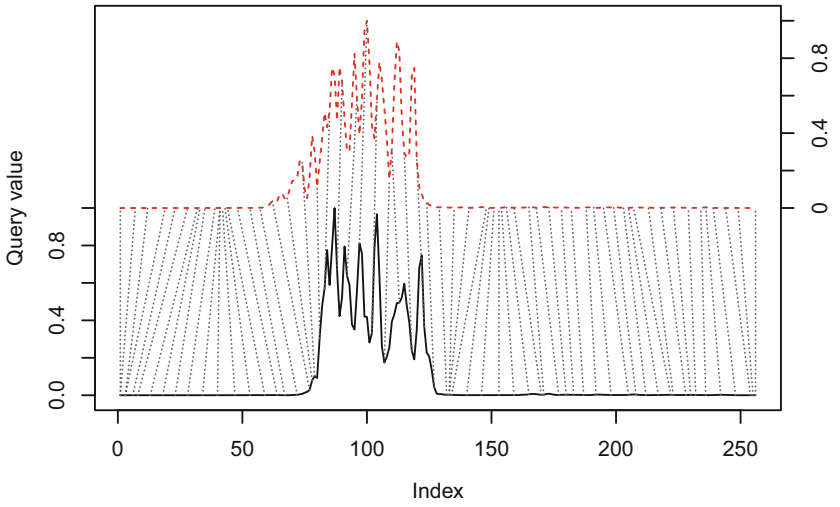


Fig. 17.9 Dynamic time warping on frequency spectra. The mean frequency spectra of `note2` and `note3` of `tico` are compared using dynamic time warping alignment. Note that the frequency spectra have here the same length (256 bins) but that their length could differ. The dotted gray lines connect the frequency bins that match following the best alignment found by the algorithm

A fast way to obtain this distance is to turn the argument `distance.only` to `TRUE`:

```
dtw(note2.env, note3.env, distance.only=TRUE)$distance
[1] 134790.6
```

Similarly, frequency spectra, here mean frequency spectra, can be compared with (Fig. 17.9):

```
note2.mspec <- meanspec(note2, plot=FALSE)[,2]
note3.mspec <- meanspec(note3, plot=FALSE)[,2]
res <- dtw(note2.mspec, note3.mspec, keep=TRUE)
plot(res, type="two", off=1, match.indices=length(note2.mspec)/4)
```

The same procedure can be applied to the dominant frequency tracking (or contour) obtained with `dfreq()` for the two notes (see Sect. 13.1.1). `NA` values that may occur in the result of `dfreq()` should be removed with `na.omit()`. Note that in this case, the two series differ in length (11 and 14, respectively) and that

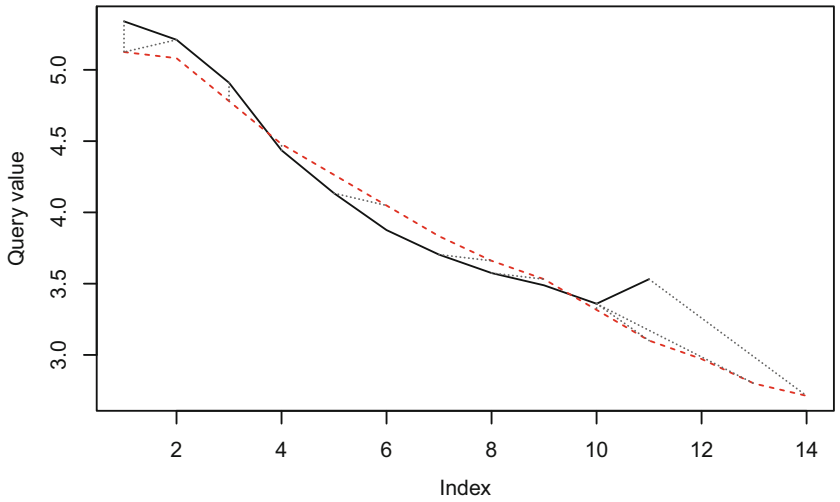


Fig. 17.10 Dynamic time warping on dominant frequency tracking. The dominant frequency of `note2` and `note3` of `tico` was obtained with `dfreq()` and then compared using dynamic time warping alignment. Note that the frequency tracks have not the same length (11 and 14 measurements, respectively). The dotted gray lines connect the dominant frequency measurements that match following the best alignment found by the algorithm

using no graphical offset seems to be useful to compare the frequency modulations (Fig. 17.10):

```
note2.df <- na.omit(dfreq(note2, threshold=5, plot=FALSE)[,2])
note3.df <- na.omit(dfreq(note3, threshold=5, plot=FALSE)[,2])
res <- dtw(note2.df, note3.df, keep=TRUE)
plot(res, type="two", match.indices=length(note2.df))
```

The package `warbleR` offers two end-user functions, `ffDTW()` and `dfDTW()`, to directly apply DTW on fundamental frequency and dominant frequency contours.

17.4 Automatic Identification

17.4.1 Principle

Automatic identification pertains to the field of machine learning, a very active discipline that combines data mining techniques, high-level statistics, and computer sciences. Automatic identification is a computer task that consists in taking a

decision about the identity of an object, usually a digital object, without human expertise. This task can be conducted in a supervised or unsupervised way. In the first case, the identification is achieved with access to labeled data, that is, with a collection of identified objects. In the second case, the identification is performed without any labeled reference.

Here, we will deal with a single automatic identification task that consists in localizing the occurrence of a known sound of interest (SOI), as a bird note, in an unlabeled recording (recording test), as a recording achieved by a naive bird watcher. The task can be seen as a binary classification task: the SOI does occur at time t in the recording test, in that case the outcome of the system is 1; the SOI does not occur at time t in the recording test, in that case the outcome of the system is 0. The task can be of course applied to a group of different recording tests through a batch process.

The achievement of the binary task is based on the development of a **system** that compares one or several examples of the SOI, also known as template, with successive time sections of the recording to be analyzed. The goal is not to find exactly the same sound as the template but to assess the level of similarity between the template and the recording and then to take a decision considering this degree of similarity. Such a system is usually based on the following workflow (Fig. 17.11):

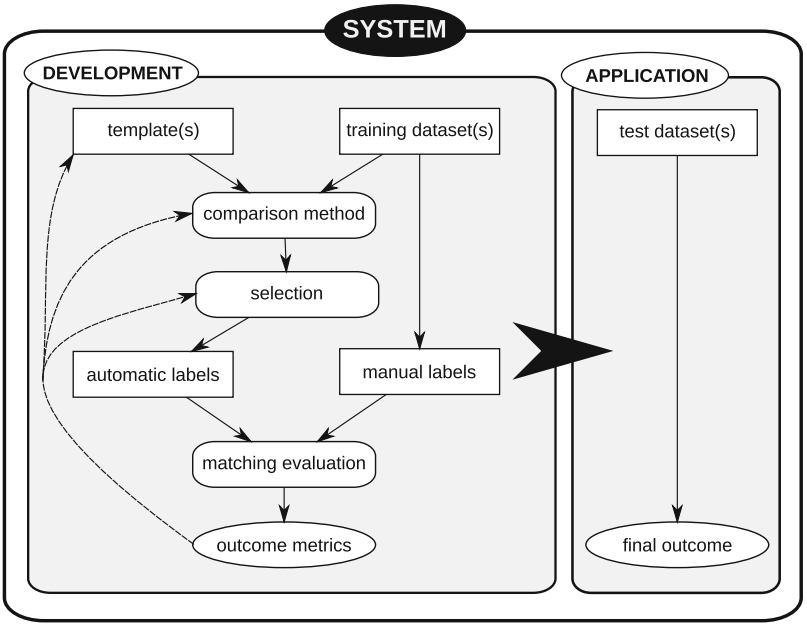


Fig. 17.11 Automatic identification system workflow. An automatic identification system can be divided into two major components: a first phase of development where the system is built and trained based on one or several templates, one or several training datasets, and a second phase of application on one or several test datasets. The plain arrows indicate the basic way of the workflow, and the dashed arrows indicate feedback to optimize the system. See text for further details

1. selection of one or, more commonly, several **templates**. This collection of templates should give a good overview of the time $\times$ frequency variation of the SOI and of the background that can occur in the recordings.
2. selection of one or several labeled datasets, named **training dataset**. This database consists in a collection of recordings where the occurrence of the SOI is manually labeled by a human expert. This reference is also known as the **ground truth**.
3. selection of a **comparison method** that estimates the similarity between the templates and the training dataset. This method can combine signal analysis and machine learning.
4. application of the comparison method on the training dataset. The method returns 0|1 (or presence | absence or positive | negative) predictions in reference to an **output threshold** θ that select the most similar events.
5. **evaluation** of the classification rates by matching the manual labels edited by a human expert and the automatic labels obtained with the comparison method. This leads to 2×2 confusion matrix with the following categories: (1) true positives (TP) that are correct positive predictions, (2) false positives (FP) that are negative examples incorrectly labeled as positive, (3) true negatives (TN) that are correct negative predictions, and (4) false negatives (FN) that are positive examples incorrectly labeled as negative (Table 17.1).
6. evaluation of the performance of the comparison method by computing **outcome metrics** precision such as:

$$\text{true positive rate} = \text{TPR} = \text{sensitivity} = \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{true negative rate} = \text{TNR} = \text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Table 17.1 Confusion matrix in automatic identification process

	Ground truth	
	True	False
System		
True	TP	FP
False	FN	TN

Confusion matrix between ground truth and algorithm prediction indicating the predictions that this is the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN)

$$\text{false positive rate} = \text{FPR} = 1 - \text{specificity} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{positive predictive value} = \text{PPV} = \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

7. **optimization** of the system performance by tuning the parameters of the system in particular the output threshold θ. The system performance is assessed by constructing the receiver operating characteristic curve (ROC curve) (Fig. 17.12) and/or the precision-recall curve (PR curve) that are both θ functions (Davis and Mermelstein 1980; Fawcett 2006). The ROC curve consists in plotting FPR(θ) on the x-axis against TPR(θ) on the y-axis. The PR curves display the same TPR(θ) named "recall" on the x-axis and the "precision" according to θ on the y-axis. The area under the curve (AUC, respectively the AUC-ROC and AUC-PR) is a metric for the performance of the system. The AUC is obtained by computing the integral of TPR(θ) (resp. "precision") with respect to FPR(θ) (resp. "recall") using trapezoid rule integration.
8. application of the tuned comparison method on an unlabeled dataset known as **testing dataset**. The final outcome of the system is a prediction of the occurrence of the SOI.

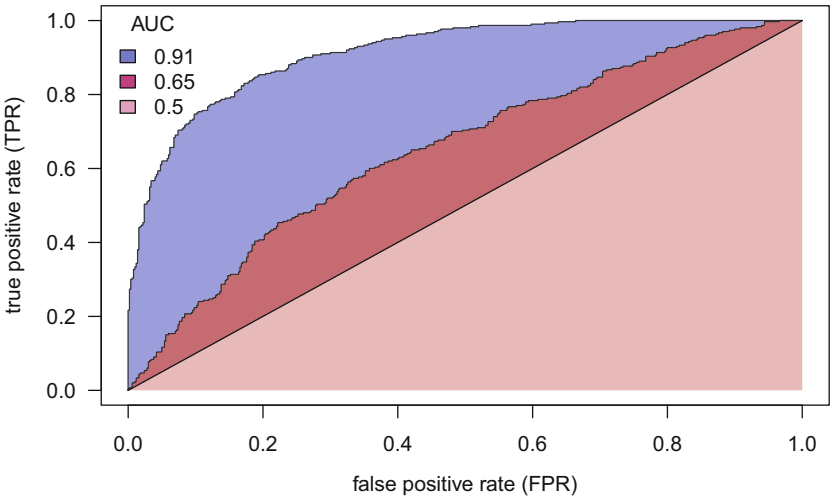


Fig. 17.12 Receiver operating characteristic (ROC). The false positive rate (FPR) and the true positive rate (TPR) define the ROC space. The plain curves indicate the ROC curves for an efficient system (blue), a non-efficient system (red), and a system returning random predictions (pink). Areas under the curve (AUC) are colored accordingly and specified in the legend

17.4.2 In Practice with the Package `monitoR`

The package `monitoR` covers two main aims: (1) manage a database resulting of an acoustic monitoring or survey and (2) run an automatic binary classification task. The package functionalities are described in depth in Katz et al. (2016a,b) and in the "Quick Start" vignette of the package than can be reached with:

```
library(monitoR)
vignette("monitoR_QuickStart", package="monitoR")
```

`monitoR` offers two methods of comparison, namely, cross-correlation as detailed in Sect. 17.1 and bin template matching as defined in Towsey et al. (2012). Bin template matching does not apply a direct comparison between the template and the training dataset. The STDFT matrix of the template is binarized according to an amplitude threshold resulting in a binary matrix with cells labeled as *on* (signal) and *off* (background noise) cells. For instance, if the user sets a threshold of 6 dB, all the cells of the STDFT matrix between 0 and −6 dB will be considered as *on* cells, the remaining cells being defined as *off* cells. A score is then computed for the template according to the equation:

$$\text{score} = \frac{\sum a_{on}}{n_{on}} - \frac{\sum a_{off}}{n_{off}}$$

where a are the amplitudes of the cells and n are the number of cells. The same score is computed on the training set, and the score of the training set is compared to the score of the template according to a selection threshold θ.

Whichever the comparison method selected, the main steps of the workflow followed by `monitoR` can be listed as:

1. manual annotation of the training dataset using `viewSpec()` with the argument `annotate=TRUE`.
2. selection of one or more templates with `makeCorTemplate()` for cross-correlation (resp. `makeBinTemplate()` for bin template matching) and combination of these templates in a single object with `combineCorTemplates()` (resp. `combineBinTemplates()`).
3. setting a negative value to the output threshold θ with `templateCutoff()`.
4. application of the comparison method with `corMatch()` (resp. `binMatch()`).[2] This generates the raw results of the comparison method, that

[2]See the functions, `batchCorMatch()` [resp. `batchBinMatch()`], for operating the template-training dataset comparison on set of files speeding up the complete process.

is, the time series of the cross-correlation $r_{\text{template,training}}$ (resp. bin template matching score).
5. selection of the peaks of $r_{\text{template,training}}$ time series with `findPeaks()`.
6. generation of the outcome of the system as `TRUE` / `FALSE` labels with `getPeaks()`.
7. find a consensus between the detections of the different templates with `timeAlign()`.
8. comparison between manual labels and automatic labels with `eventEval()` with different values of θ so that the ROC curve can be built.
9. selection of θ on the ROC curve for optimizing the system performance.
10. selection of other parameters, as time tolerance τ in manual and automatic comparison, for optimizing the system performance.
11. application of the tuned system on a test dataset.

In the following paragraph, we work a simple detection example. We use the following data and parameters:

SOI vocalizations of the South-American dart poison frog *Allobates femoralis* (see Chap. 14 and Fig. 14.1),
templates four templates stored in the file `Allobates_femoralis.wav` stored in the directory `sample`,
training dataset 28 vocalizations occurring in the 30 s file `Allobates_femoralis_2015-11-10_161500_GFT.wav` stored in the same directory `sample`. The name of the file is, according to `monitoR` instructions, constructed following: `"name_YYYY-MM-DD_HHMMSS_TIMEZONE.wav"`:

```
femo.training <- readWave(
    "sample/Allobates_femoralis_2015-11-10_161500_GFT.wav")
femo.training

Wave Object
Number of Samples:      1323000
Duration (seconds):     30
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

ground truth manual labels edited on the training dataset and written in a `.csv` file,
comparison method cross-correlation,
output threshold set a priori to $\theta = -0.1$ and then tuned,
system performance ROC curve and AUC,
test dataset not included.

We obtained the manual annotations on the training file using the interactive spectrographic function `viewSpec()` with the argument `annotate=TRUE` to allow interactive annotations with the mouse (see Sect. 11.8.2.2). To facilitate the annotation, we focused in frequency between 1 and 7 kHz with the argument `frq.lim`, and we navigated through the sound using overlapping spectrographic views lasting 5 s with the arguments `page.length` and `page.ovlp`. This action produces a `data.frame` written in a `.csv` file saved in a directory specified with the argument `output.dir`:

```
viewSpec(femo.training, frq.lim=c(1, 7),
         page.length=5, page.ovlp=0.2,
         annotate=TRUE, output.dir="data")
```

We just followed the instructions printed in the console by `viewSpec()` to manually delimit and label the SOI. Here, the annotations were saved in a file named `femo-training-annotations.csv`. The resulting `data.frame` is organized in five columns: the start time in s (`start.time`), the end time in s (`end.time`), the minimum frequency in kHz (`min.frq`), the maximum frequency in kHz (`max.frq`), and the text label (`name`) of each annotation. The annotation file can be loaded afterward with a classical table reading function:

```
manual <- read.csv("data/femo-training-annotations.csv")
```

We can check we have 28 frog vocalizations or SOI:

```
dim(manual)
[1] 28  5
head(manual)
  start.time end.time min.frq max.frq name
1      0.288    0.558  1.4952  5.8852    1
2      2.873    3.173  1.4711  5.9093    2
3      6.140    6.352  1.5194  6.1023    3
4      7.867    8.137  1.4952  6.0540    4
5      9.195    9.416  1.4591  5.9937    5
6     10.352   10.603  1.4470  5.7887    6
```

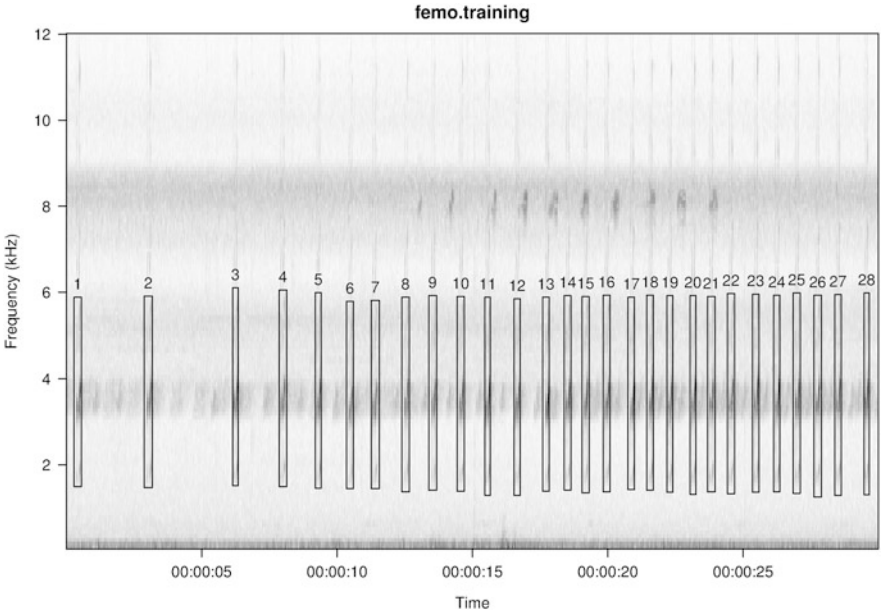


Fig. 17.13 Visualization of manual annotations with `viewSpec()`. The 28 SOI of the `Allobates_femoralis.wav` recording were delimited and overlaid on a spectrographic display with `viewSpec()`

These annotations can also be visually checked afterward using another time `viewSpec()` by importing the `.csv` file containing the annotations (Fig. 17.13):

```
viewSpec(femo.training, annotate=TRUE,
         anno="data/femo-training-annotations.csv")
```

Now that we have listed all the required data and that we have the annotations for the ground truth, we can select and build the templates. The function `makeCorTemplate()` prepares a template for the cross-correlation method. The external file is used as a direct input. Here we select the four vocalizations included in the file `Allobates_femoralis.wav` as separate templates. The selection in time and frequency is achieved using the arguments `t.lim` in `s` and `freq.lim` in kHz. We add a name (`t1`, `t2`, `t3`, `t4`) with the argument `name` to facilitate individual calling:

```
template1 <- makeCorTemplate("sample/Allobates_femoralis.wav",
                             t.lim=c(0.09, 0.24), frq.lim=c(3,6),
                             name="t1")
template2 <- makeCorTemplate("sample/Allobates_femoralis.wav",
                             t.lim=c(0.36, 0.51), frq.lim=c(3,6),
                             name="t2")
template3 <- makeCorTemplate("sample/Allobates_femoralis.wav",
                             t.lim=c(0.9, 1.04), frq.lim=c(3,6),
                             name="t3")
template4 <- makeCorTemplate("sample/Allobates_femoralis.wav",
                             t.lim=c(1.15, 1.32), frq.lim=c(3,6),
                             name="t4")
```

We then associate the four templates in a single object with `combineCorTemplates()`:

```
templates <- combineCorTemplates(template1, template2,
                                 template3, template4)
```

The templates can be quickly visualized with `plot()`:

```
plot(templates)
```

A default threshold or cutoff score $\theta = 0.4$ is associated to each template. The value of θ can be changed with the function `templateCutoff()` and can also be modified later at several steps of the system. Here we will first set it up to $\theta = -0.1$ to consider all possible events as positive, or true:

```
templateCutoff(templates) <- rep(-0.1,4)
templateCutoff(templates)
  t1   t2   t3   t4
-0.1 -0.1 -0.1 -0.1
```

Now comes the time to apply the comparison method, that is, the cross-correlation between each template and the training recording, so that we end up

with four $r_{\text{template,training}}$ series. The action is rather simple: we only give the path to the training file and the template object to the function `corMatch()`:

```
scores <- corMatch(
    "sample/Allobates_femoralis_2015-11-10_161500_GFT.wav",
    templates)
```

`corMatch()` has a print method that summarizes the correlation values obtained for each template:

```
scores

A "templateScores" object

Based on the survey file:  sample/Allobates_femoralis_
    2015-11-10_161500_GFT.wav

And  4  templates
Score information
   min.score max.score n.scores
t1     -0.15      0.93     2571
t2     -0.12      0.86     2571
t3     -0.15      0.92     2571
t4     -0.08      0.85     2569
```

The structure of `scores` is a bit complex but the raw data can be obtained by selecting the right items and subitems. The following instruction plots $r_{\text{template,training}}$ for the first template `"t1"` (Fig. 17.14):

```
plot(scores@scores$t1$time, scores@scores$t1$score,
     type="l", col="blue", xlab="Time (s)", ylab="r")
```

We then identify the peaks occurring in the four $r_{\text{template,training}}$ series with the function `fpeaks()`:

```
peaks <- findPeaks(scores)
```

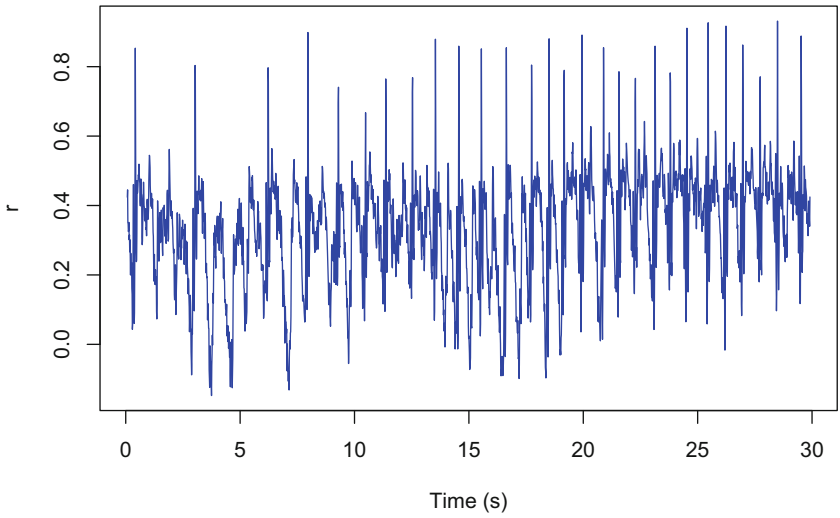


Fig. 17.14 Cross-correlation with the package `monitoR`. The time series of the correlation coefficient as stored in the result of the function `corMatch()`. The function was applied between four templates and a training file `Allobates_femoralis.wav`. Only the score for the template `t1` is here displayed

`findPeaks()` returns an `S4` object containing a long list of items. However, `findPeaks()` has also a print method to summarize the information:

```
peaks

A "detectionList" object

Based on survey file:  sample/Allobates_femoralis_2015-11-10_
    161500_GFT.wav

and  4  templates

Detection information
   n.peaks n.detections min.peak.score max.peak.score
t1     200          200    -0.03492637      0.9313097
t2     206          206     0.18363149      0.8623351
t3     189          189    -0.03607333      0.9169110
t4     142          142     0.35517511      0.8529895
   min.detection.score max.detection.score
t1         -0.03492637           0.9313097
t2          0.18363149           0.8623351
t3         -0.03607333           0.9169110
t4          0.35517511           0.8529895
```

The time position of the peaks selected is stored in the `S4` slot labeled `@peaks`. This slot contains a list with four items, each item corresponding to one template. Each template item is a `data.frame` with four columns giving the absolute time including date of the Gregorian calendar, the time in s, the correlation coefficient $r_{\text{template,training}}$, and a logical label for detection. Here we print the structure of the `data.frame` of the template `t1`:

```
str(peaks@peaks$t1)
'data.frame': 200 obs. of  4 variables:
 $ date.time: POSIXct, format: "2015-11-10 16:14:30" ...
 $ time     : num  0.0813 0.3367 0.418 0.5805 0.6618 ...
 $ score    : num  0.445 0.46 0.853 0.519 0.476 ...
 $ detection: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
```

The peaks can be nicely visualized with a two-panel plot that combines an overlay of the peaks on the spectrogram and the time series. Plotting the peaks of the four templates may generate a blurry graphics so that we can use the option `which.one` to select a single template. We can also remove the legend with the argument `legend` and highlight the peaks with a circle with the argument `hit.marker` (Fig. 17.15):

```
plot(peaks, which.one="t1", legend=FALSE, hit.marker="points")
```

We eventually extract in a `data.frame` the information of each selected peak with the function `getPeaks()`

```
peaks.selected <- getPeaks(peaks)
head(peaks.selected)
  template           date.time       time     score
1       t1 2015-11-10 16:14:30 0.08126984 0.4450226
2       t1 2015-11-10 16:14:30 0.33668934 0.4604150
3       t1 2015-11-10 16:14:30 0.41795918 0.8531751
4       t1 2015-11-10 16:14:30 0.58049887 0.5190559
5       t1 2015-11-10 16:14:30 0.66176871 0.4764891
6       t1 2015-11-10 16:14:30 0.77786848 0.4671649
  detection
1      TRUE
2      TRUE
3      TRUE
4      TRUE
5      TRUE
6      TRUE
```

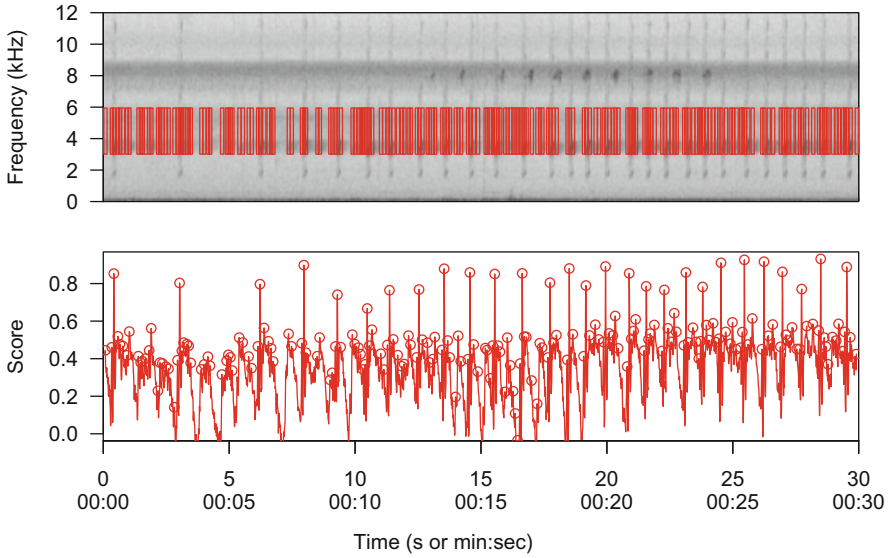


Fig. 17.15 Automatic detection with the package `monitoR`. The two-panel figure obtained with `plot()` on an object obtained with `findPeaks()` on the template 1. The top panel is a spectrogram with detections indicated with red rectangles. The bottom panel shows the time series of the correlation coefficient, here named Score. In this case, no selection (threshold $\theta = -0.1$) was applied so that all peaks were considered as positive or true detections

As we used several templates, we end up with information that may be redundant, that is, a single vocalization detected by several templates. This redundancy can be reduced by combining the detections and keeping only the best ones, that is, the detections that have the best scores. Such a consensus of the results can be obtained with the function `timeAlign()` with the argument `what="peaks"`. The time matching between the peaks might not be perfect such that some time tolerance can be specified with the argument `tol` expressed in s. We can define a tolerance value in reference to the median duration of the SOI to be detected. Here, the median of the frog vocalization duration can be obtained from the manual annotations with:

```
duration <- median(manual$end.time - manual$start.time)
duration
[1] 0.251
```

This duration is here divided by 2 to be used as the time tolerance so that $\tau \approx 0.126$:

```
automatic <- timeAlign(peaks.selected, what="peaks",
                       tol=duration/2)
```

We can now compare the results of the automatic detection operated by the system with the ground truth obtained by an expert in reference to the threshold or score cutoff θ such that we can estimate the number of true positives, true negatives, false positives, and false negatives. This is achieved with the function `eventEval()` which has two main arguments to apply the selection. The argument `score.cutoff` is the threshold θ applied on the correlation time series of the cross-correlation $r_{\text{template,training}}$, and the argument `tol` is the time tolerance τ as in `timeAlign()`. The function compares the position of the detection stored in the column `time` of the object `peaks.selected` and the position of the items labeled by the user with `viewSpec()`, here stored in the object `manual`. The positions obtained automatically in `peaks.selected` correspond to the midpoint time value, that is, the time center of the detection. A similar midpoint position is derived from the manual annotations using the `start.time` and `end.time` columns of `manual`. As for `timeAlign()` when comparing peaks of the correlation $r_{\text{template,training}}$ series, the time matching cannot be totally exact so that some tolerance should be set to make a correspondence between the automatic and manual time positions. We apply similarly $\tau \approx 0.126$. Considering a usual value of $\theta = 0.4$, we end up with the next command:

```
res <- eventEval(detections=automatic, standard=manual,
                 what="peaks", score.cutoff=0.4, tol=duration/2)
head(res)
  template           date.time       time     score
1       t2 2015-11-10 16:14:30 0.08126984 0.5147601
2       t1 2015-11-10 16:14:30 0.33668934 0.4604150
3       t3 2015-11-10 16:14:30 0.41795918 0.8532125
4       t4 2015-11-10 16:14:30 0.49922902 0.5814246
5       t2 2015-11-10 16:14:30 0.58049887 0.5887462
6       t2 2015-11-10 16:14:30 0.66176871 0.5477226
  detection outcome
1      TRUE FALSE +
2      TRUE FALSE +
3      TRUE  TRUE +
4      TRUE FALSE +
5      TRUE FALSE +
6      TRUE FALSE +
```

We can summarize the final results calling the base function `table()` on the results so that we have directly access to the number of outcomes (true positives, false positives, true negatives, and false negatives):

```
table(res$outcome)

FALSE - FALSE +  TRUE -  TRUE +
      1     160      29      31
```

We observe that the number of true positives exceeds the number of SOI in the training file ($31 > 28$). This overestimation might be due to a mischoice in the values of θ and τ. We need first to optimize the performance of the system in reference to θ. The idea is to build the ROC curve by repeating the above process for several values of θ. We can, for instance, set $\theta \in [0, 1]$ with $\theta_{n+1} = \theta_n + 0.01$, something which is written in `R` as:

```
theta <- seq(0, 1, by=0.01)
```

We write a `for` loop so that we obtain a confusion matrix and the number of outcomes for each value of θ. We first prepare a data frame with `NA` values to store the results:

```
categories <- matrix(NA, nrow=length(theta), ncol=5)
categories <- as.data.frame(categories)
colnames(categories) <- c("theta", "tp", "tn", "fp", "fn")
categories[,1] <- theta
```

and we write a `for` loop around θ:

```
for(i in 1:length(theta)){
    tmp <- table(eventEval(detections=automatic, standard=manual,
                           what="peaks", score.cutoff=theta[i],
                           tol=duration/2)$outcome)
    categories[i,2] <- tmp["TRUE +"]
    categories[i,3] <- tmp["TRUE -"]
    categories[i,4] <- tmp["FALSE +"]
    categories[i,5] <- tmp["FALSE -"]
}
```

The output of the loop may contain NA values that we replace by 0 values:

```
categories[is.na(categories)] <- 0
```

The head and tail of categories look then as:

```
head(categories)
  theta tp tn  fp fn
1  0.00 32  1 188  0
2  0.01 32  1 188  0
3  0.02 32  1 188  0
4  0.03 32  2 187  0
5  0.04 32  2 187  0
6  0.05 32  2 187  0
tail(categories)
    theta tp  tn fp fn
96   0.95  0 189  0 32
97   0.96  0 189  0 32
98   0.97  0 189  0 32
99   0.98  0 189  0 32
100  0.99  0 189  0 32
101  1.00  0 189  0 32
```

We can use the table categories to compute the FPR and TPR metrics required to build the ROC curve. We use the base function with() to facilitate the manipulation of the columns:

```
tpr <- with(categories, tp/(tp+fn))
fpr <- with(categories, fp/(fp+tn))
```

The AUC metric can be obtained with the function trapz() of the package caTools which implements the trapezoid rule integration for any curve. The values just need to be reversed:

```
library(caTools)
auc <- trapz(rev(fpr), rev(tpr))
auc
[1] 0.9260086
```

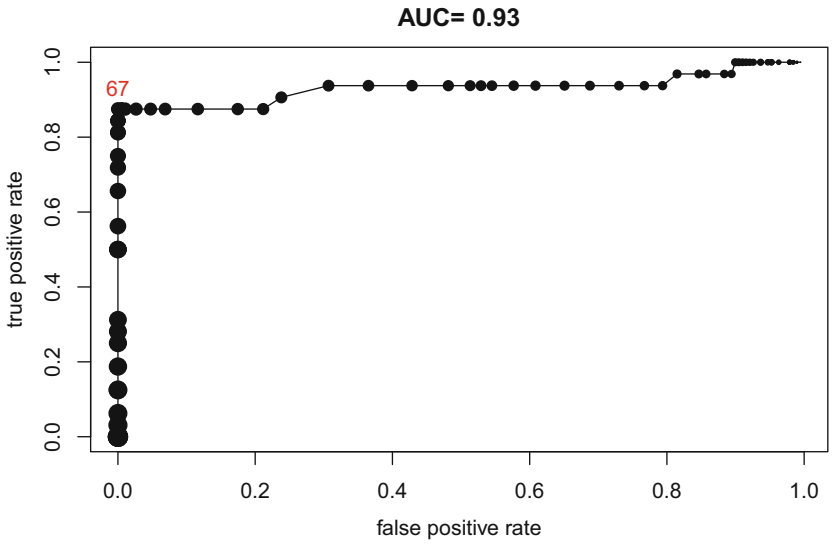


Fig. 17.16 ROC curve for *Allobates femoralis* vocalization identification. The curve was built by varying the output threshold θ from 0 to 1 by step of 0.01. The size of the points is relative to θ. The point 67 was chosen as the best output threshold θ with a good TPR and a null FPR

We can finally display the ROC curve and the AUC value with (Fig. 17.16):

```
plot(fpr, tpr, type="o", pch=19, cex=theta*2,
     xlab="false positive rate", ylab="true positive rate",
     main=paste("AUC=", round(auc,2)))
```

The ROC curve shows a inflection point with a high TPR and a null FPR. This inflection point can be identified using `identify()` as in:

```
identify(fpr, tpr)
```

This allows to flag the point 67 so that we can obtain the corresponding FPR, TPR, and θ values:

```
fpr[67]
[1] 0
tpr[67]
```

(continued)

```
[1] 0.875
theta[67]
[1] 0.66
```

As we wish to minimize the FPR and maximize the TPR, we eventually choose $\theta = 0.66$ as the best output threshold.

However, this choice is not the final point of the story. We now need to optimize the choice of the time tolerance τ set with the argument `tol` of the function `eventEval()`. The idea is to produce a ROC curve for a series of tolerance values and compute the AUC of each ROC curve. The highest AUC should determine the best tolerance value. Hence we will make vary both θ and τ. θ will vary according to $\theta_{n+1} = \theta_n + 0.01$ with $\theta \in [0, 1]$ and τ according to $\tau_{n+1} = \tau_n + 0.01$ with $\tau \in [0, 0.2]$:

```
theta <- seq(0, 1, by=0.01)
tau <- seq(0, 0.2, by=0.01)
```

We first prepare a vector where the AUC values to be computed will be stored:

```
auc <- rep(NA, times=length(tau))
```

We develop a double loop `for` based on the previous code used to obtain the ROC curve:

```
## loop around tau
for(j in 1:length(tau)){
    ## empty data frame for data storage
    categories <- matrix(NA, nrow=length(theta), ncol=4)
    categories <- as.data.frame(categories)
    colnames(categories) <- c("tp", "tn", "fp", "fn")
    ## loop around theta
    for(i in 1:length(theta)){
        tmp <- table(eventEval(detections=automatic,
                               standard=manual,
                               what="peaks",
                               score.cutoff=theta[i],
                               tol=tau[j])$outcome
                     )
        categories[i,1] <- tmp["TRUE +"]
```

(continued)

```
        categories[i,2] <- tmp["TRUE -"]
        categories[i,3] <- tmp["FALSE +"]
        categories[i,4] <- tmp["FALSE -"]
    }
    ## replacement of NA values by 0 values
    categories[is.na(categories)] <- 0
    ## computation of metrics, TPR and FPR
    tpr <- with(categories, tp/(tp+fn))
    fpr <- with(categories, fp/(fp+tn))
    ## computation of AUC with trapezoid rule integration
    auc[j] <- trapz(rev(fpr), rev(tpr))
}
```

We can now look for the coordinates of the highest AUC value:

```
tau.max <- tau[which.max(auc)]
tau.max
[1] 0.09
auc.max <- auc[which.max(auc)]
auc.max
[1] 0.9853628
```

and also plot the result as a simple profile of AUC according to τ (Fig. 17.17):

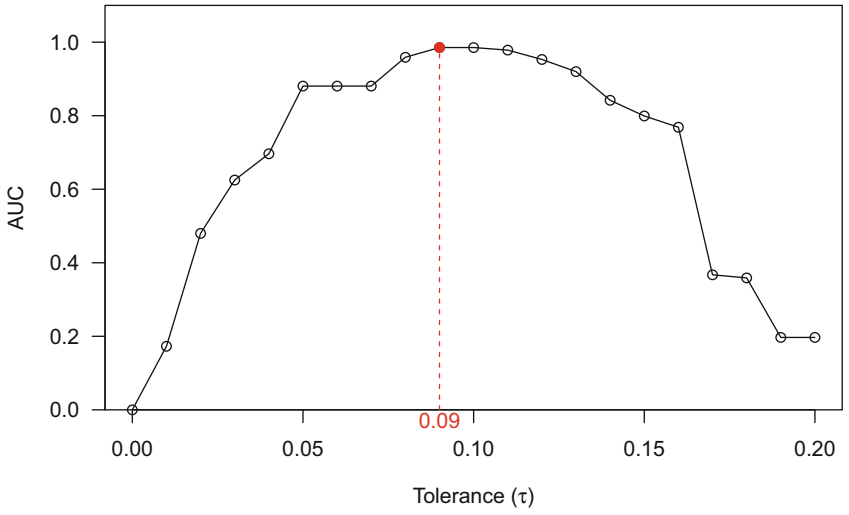


Fig. 17.17 Variation of the area under the curve (AUC) according to time tolerance (τ). The AUC was computed for a series of time tolerances between 0 and 0.2. The area reaches a maximum when $\tau = 0.09$

```
col <- "red"
par(xpd=TRUE, las=1)
plot(tau, auc, type="o", yaxs="i", ylim=c(0,1.1),
     xlab=expression(paste("Tolerance (", tau, ")")),
     ylab="AUC")
points(tau.max, auc.max, pch=19, col=col)
segments(x0=tau.max, y0=0, x1=tau.max, y1=auc.max,
         lty=2, col=col)
text(tau.max, -0.03, label=tau.max, col=col)
```

We can check the system with $\theta = 0.67$ and $\tau = 0.09$ on the training set by running the complete chain of functions:

```
templateCutoff(templates) <- rep(0.67,4)
scores <- corMatch(
    "sample/Allobates_femoralis_2015-11-10_161500_GFT.wav",
    templates)
peaks <- findPeaks(scores)
peaks.selected <- getPeaks(peaks)
automatic <- timeAlign(peaks.selected, what="peaks",
                       tol=duration/2)
res <- eventEval(detections=automatic, standard=manual,
                 what="peaks", score.cutoff=0.67, tol=0.09)
```

The final result is:

```
table(res$outcome)

FALSE -  TRUE -  TRUE +
      2     192      27
```

We see that only one SOI, the sixth vocalization, among the 28 manually annotated is missed by the automatic system (Fig. 17.18):

```
plot(peaks, legend=TRUE, hit.marker="points")
```

This result is of quite good quality and suggests that the system could be applied on an unlabeled test dataset with $\theta = 0.67$ and $\tau = 0.09$.

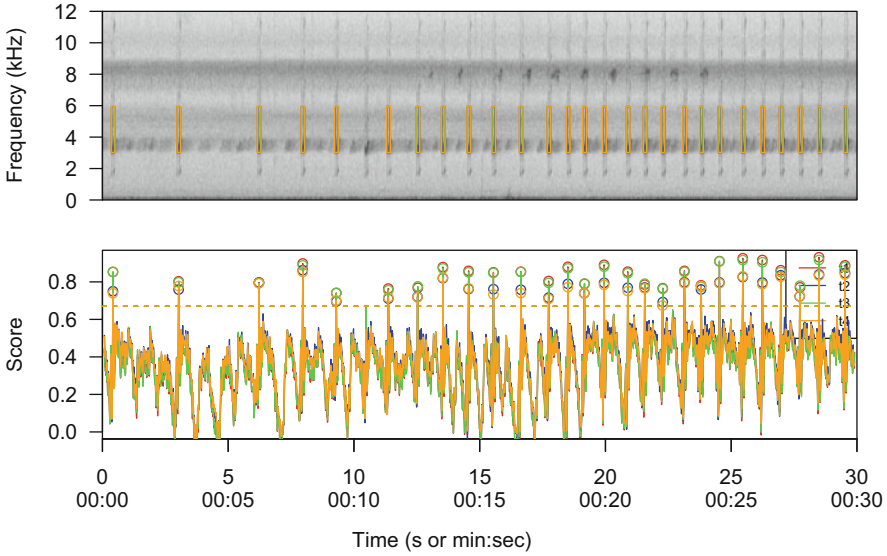


Fig. 17.18 Automatic detection with the package `monitoR`: final check. The final results of the automatic detection system applied on the training dataset, here a single file containing 28 vocalizations of *Allobates femoralis*. The plot shows the detections of all four templates. Only the sixth vocalization is missed

Chapter 18
Synthesis

All the previous chapters dealt with the exploration of sound through parametrization. In this last chapter, we will see how to generate, or synthesize, sound. Sound synthesis can be used in different contexts, including computer music or experimental procedures where the effects of a sound, a stimulus, are tested through a playback procedure onto a specific subject. This latter can be a living animal, including a human being, an environment as a theater or a forest, or a still object as a specific material. Sound synthesis can also be useful to check the quality of a data analysis or to test the validity of a model which fully or partly relies on acoustics.

In the following sections we will see how to generate sound ex nihilo or, occasionally, use some parameters of a pre-existing sound. The chapter will first cover the generation of no sound, that is, of silence, and then of noise, non-sinusoidal sound, sinusoidal sound, tonal sound, and, lastly, the particular case of the human voice.[1] Sound synthesis may require to edit, filter, and modify existing sounds so that we will use occasionally functions described in Chaps. 6, 14, and 15. The sounds synthesized in this chapter can be accessed in the directory `sample` as `.wav` files as mentioned in footnotes.

18.1 Silence

We have seen in Sect. 6.4 that it was possible to edit silence bouts. In particular, the `seewave` function `addsilw()` can be invoked to add a silence bout into a sound object. The `tuneR` function `silence()` can also be used to generate a silence section that can be then bound to any sound. We create with the following action a 16 bit PCM silence `wave` object lasting 1 s and sampled at $f_s = 22{,}050$ Hz. We

[1] We will not consider sonification that is the process of converting data into sound (Hermann et al. 2011). See the packages `playitibyr` and `audiolyzR`.

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7_18

specify with the argument `xunit="time"` that we wish to refer to seconds for the argument `duration` (setting `xunit="samples"` and `duration=1` will produce an object with a single sample)[2]:

```
s <- silence(duration=1, samp.rate=22050, xunit="time",
             bit=16, pcm=TRUE)
```

The object contains a series of 22,050 zeroes:

```
s

Wave Object
Number of Samples:      22050
Duration (seconds):     1
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

```
head(s@left)
[1] 0 0 0 0 0 0
```

Such silence bout could be added to a pre-existing sound, as `tico`[3]:

```
tico.silence <- bind(tico, s)
tico.silence

Wave Object
Number of Samples:      61628
Duration (seconds):     2.79
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

[2] `synth-silence.wav`.

[3] `synth-tico-silence.wav`.

18.2 Noise

Noise can be generated with the `seewave` function `noisew()` and the `tuneR` function `noise()`. The latter is more versatile as it can generate white and colored noise, as pink, red, and any other type knowing that the frequency spectrum of the noise generated should follow the formula (see Sect. 14.6.4):

$$F(f) \propto \frac{1}{f^{\alpha}}$$

with $\alpha = 1$ for pink noise and $\alpha = 1.5$ for red noise.

White noise is a noise with a flat frequency spectrum, that is with all frequencies having equal weight—as light is a light made of all colors. Colored noises have a frequency spectrum with a specific shape. Here is the synthesis of four different noises differing in their frequency spectra (Fig. 18.1). The color of the noise is set with the first argument `kind`. The parameter α of the above expression can be directly set with the argument `alpha` taking care of to specify `kind="power"`:[4]

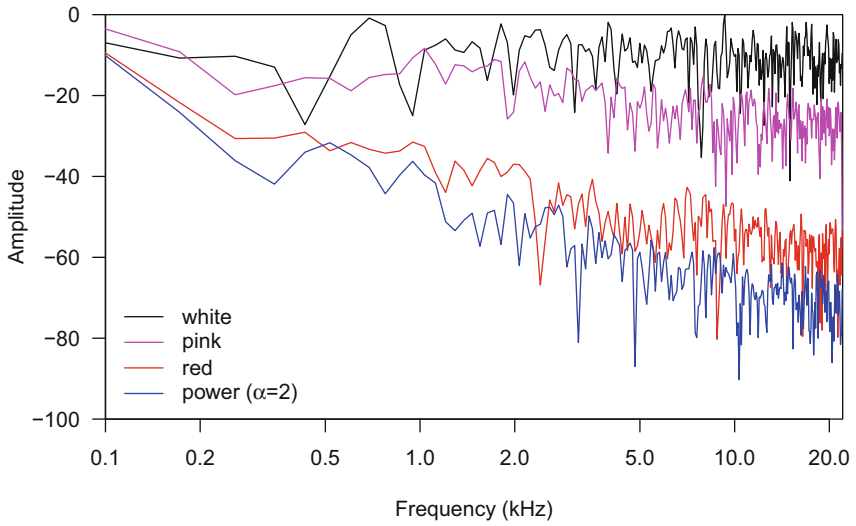


Fig. 18.1 Frequency spectrum of white and colored noises. The noises were obtained with the function `noise()` of `tuneR`. The frequency spectra were built calling `spec()` with a log frequency *x*-axis and a dB *y*-axis

[4] `synth-noise-white.wav`, `synth-noise-pink.wav`, `synth-noise-red.wav`, `synth-noise-power.wav`.

```
d <- 1 ; f <- 44100 ; xunit <- "time"
white <- noise("white", duration=d, samp.rate=f, xunit=xunit)
pink <- noise("pink", duration=d, samp.rate=f, xunit=xunit)
red <- noise("red", duration=d, samp.rate=f, xunit=xunit)
power <- noise("power", duration=d, samp.rate=f, xunit=xunit,
               alpha=2)
```

18.3 Non-sinusoidal Sound

There are four main types of non-sinusoidal but periodic waves: pulse, square, triangle, and sawtooth waves (see Sect. 2.2.6). The first three types are encoded in `tuneR` functions; the fourth type is not available in `R` but can be easily generated by designing a short function as demonstrated in the DIY box 18.1.

18.3.1 Pulse Wave

A pulse is a wave containing 0 values except for a short period where it contains a series of 1 values. A pulse follows then the condition:

$$s(t) = \begin{cases} 1 & \text{if } t \in [0, N] \\ 0 & \text{elsewhere} \end{cases}$$

The `seewave` function `pulsew()` can construct such a wave. The duration of the pulse (i.e., the duration where the wave reaches 1) is set by the argument `dpulse` when the duration of the wave before and after the pulse (i.e., the duration where the wave is 0) is set by the `dbefore` and `dafter` arguments, respectively. The following example produces a pulse lasting 0.001 s (= 1 ms) framed by silence periods lasting 0.05 s (= 50 ms) each. The class of the object returned is chosen with the argument `output` which can accept `"Wave"`, `"matrix"`, `"audioSample"`, or `"ts"` (Fig. 18.2)[5]:

```
pse1 <- pulsew(dbefore=0.05, dpulse=0.001, dafter=0.05,
               f=44100, output="Wave")
```

[5] `synth-pulse-1.wav`.

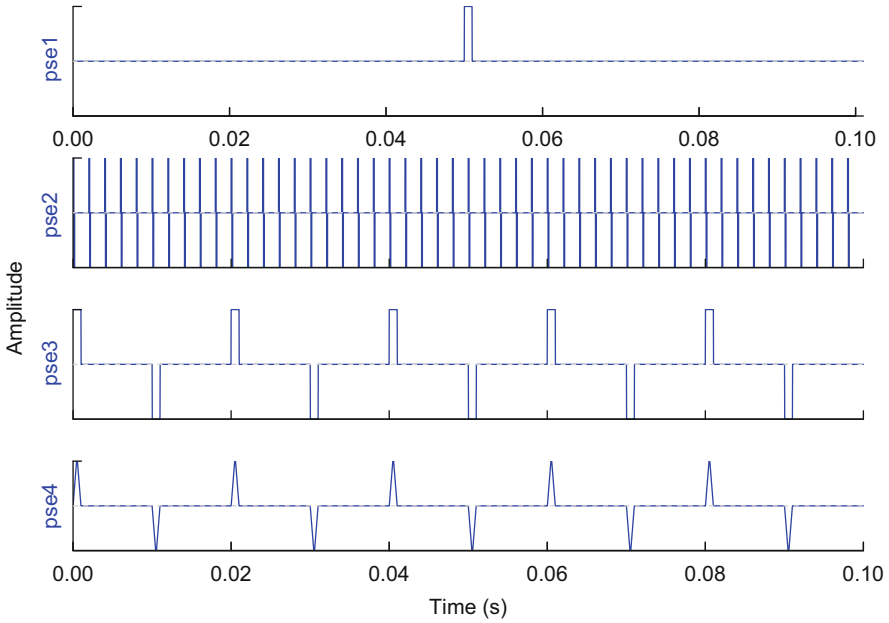


Fig. 18.2 Synthesis of pulse waves. Four series of pulses were generated with `pulsew()` of `seewave` and `pulse()` of `tuneR`. The waveforms were plot with `oscillo()`

The function `pulse()` of `tuneR` generate pulses of `Wave` class that take successively positive (+1) and negative values (−1). In addition, `pulse()` lets the user control several temporal parameters to shape the pulse. The main arguments of `pulse()` are:

`freq` the number of pulses produced per s, that is, the pulse repetition rate or carrier frequency in Hz. Note that this is different from a spectral frequency as a single pulse generates a broadband frequency spectrum,

`duration` the total duration in s of the `Wave` object returned. This is not the duration of the pulse,

`width` the relative pulse width, that is, the proportion of time the amplitude is ±1,

`plateau` the relative plateau width, that is, the proportion of the pulse width where amplitude is ±1,

`interval` the relative interval between the positive part and the negative part of the pulse. An interval of 0 produces pulses with positive and negative parts succeeding immediately when an interval of 1 places the negative and positive parts at regular time intervals.

The following examples illustrates the flexibility of the function `pulse()` (Fig. 18.2):[6]

```
f <- 44100 ; xunit <- ``time"
pse2 <- pulse(freq=50, duration=1, samp.rate=f, xunit=xunit,
           width=0.1, plateau=1, interval=0)
pse3 <- pulse(freq=50, duration=0.1, samp.rate=f, xunit=xunit,
           width=0.1, plateau=1, interval=1)
pse4 <- pulse(freq=50, duration=0.1, samp.rate=f, xunit=xunit,
           width=0.1, plateau=0.1, interval=1)
```

18.3.2 Square Wave

A square wave is a pulse with a symmetry around the amplitude axis following the time equation:

$$s(t) = A \times \text{sgn}(2\pi f_c t)$$

where A is the amplitude, f_c is pulse repetition rate or carrier frequency in Hz, and t is time in s. Such a wave can be synthesized with the function `square()` of `tuneR` which is defined by the following main arguments:

`freq` the total number of squares whichever the total duration of the sound. This is therefore not a repetition rate or frequency expressed in Hz as in `pulse()`, neither a spectral frequency,

`duration` the total duration in s of the `Wave` object returned. This is not the duration of the square,

`up` a number in]0, 1[giving the percentage of the waveform reaching the maximum value. For instance, setting `up=0.3` produces a square wave that has a maximum value during 30% of the period.

Here are some uses of `square()` (Fig. 18.3), all returning `Wave` objects:[7]

[6] `synth-pulse-2.wav`, `synth-pulse-3.wav`, `synth-pulse-4.wav`.

[7] `synth-square-1.wav`, `synth-square-2.wav`, `synth-square-3.wav`, `synth-square-4.wav`.

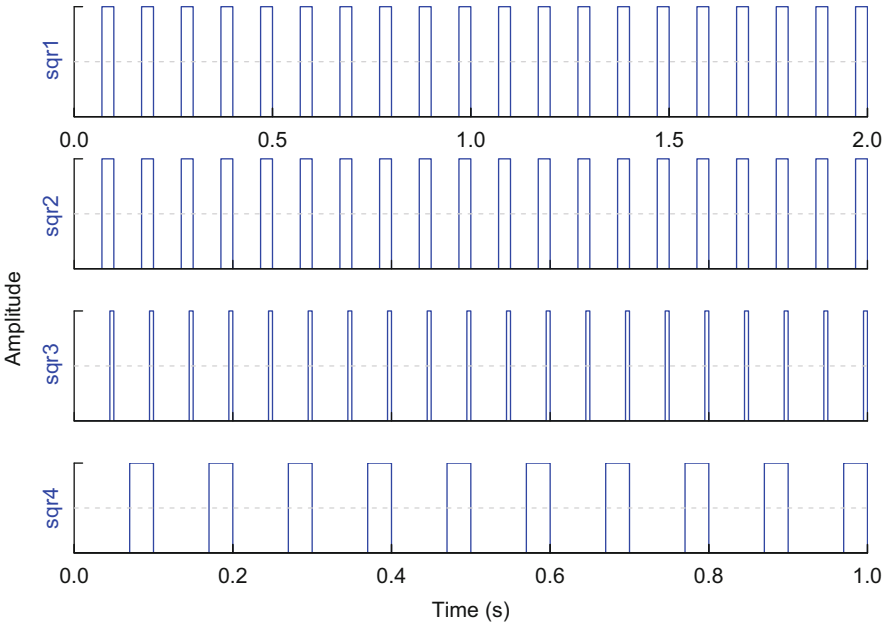


Fig. 18.3 Synthesis of square waves. Four series of squares were generated with `square()` of `tuneR`. The waveforms were plot with `oscillo()`

```
d <- 1 ; f <- 44100 ; xunit <- "time"
sqr1 <- square(freq=20, duration=d+1, up=0.3,
               samp.rate=f, xunit=xunit)
sqr2 <- square(freq=20, duration=d, up=0.3,
               samp.rate=f, xunit=xunit)
sqr3 <- square(freq=20, duration=d, up=0.1,
               samp.rate=f, xunit=xunit)
sqr4 <- square(freq=10, duration=d, up=0.3,
               samp.rate=f, xunit=xunit)
```

18.3.3 Triangle and Sawtooth Waves

A triangle wave is a symmetric wave, that is, a wave with a similar increase and decrease in amplitude (see DIY box 18.1[8]). A sawtooth wave is an asymmetric triangle wave with a slow increase in amplitude followed by fast decrease (or the

[8] `synth-triangle.wav`.

reverse). A sawtooth wave can be defined according to:

$$s(t) = A \times 2\,(f_c t - \lfloor f t + 0.5 \rfloor)$$

where A is the amplitude, t is time in s, and f_c is the repetition rate or carrier frequency in Hz. The function `sawtooth()` of `tuneR` can produce a sawtooth wave. The function, which returns `Wave` objects, has three main arguments:

`freq` the total number of saws whichever the total duration of the sound. This is therefore not a repetition rate or frequency expressed in Hz as in `pulse()`, neither a spectral frequency,

`duration` the total duration in s of the `Wave` object returned. This is not the duration of the saw,

`reverse` a logical, if `TRUE`, then the waveform is mirrored vertically, that is, a fast increase in amplitude precedes a slow decrease in amplitude.

Follow three examples of use of `sawtooth()` (Fig. 18.4)[9]:

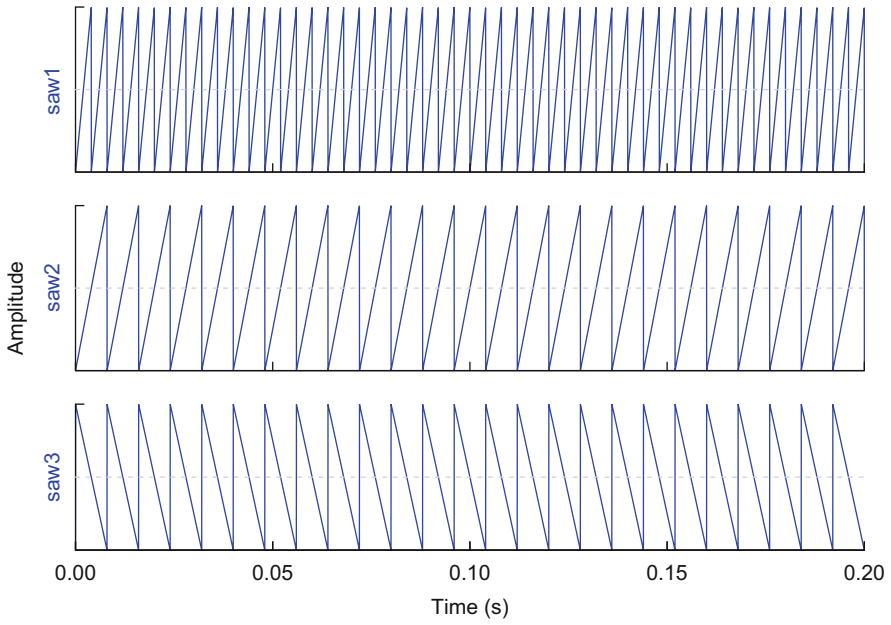


Fig. 18.4 Synthesis of sawtooth waves. Four series of sawtooth were generated with `sawtooth()` of `tuneR`. The waveforms were plot with `oscillo()`

[9] `synth-saw-1.wav`, `synth-saw-2.wav`, `synth-saw-3.wav`.

```
d <- 0.2 ; f <- 44100 ; xunit <- "time"
saw1 <- sawtooth(freq=50, duration=d, samp.rate=f, xunit=xunit,
          reverse=FALSE)
saw2 <- sawtooth(freq=25, duration=d, samp.rate=f, xunit=xunit,
          reverse=FALSE)
saw3 <- sawtooth(freq=25, duration=d, samp.rate=f, xunit=xunit,
          reverse=TRUE)
```

DIY 18.1 — How to a generate a symmetric triangle wave

There is no function to generate a symmetric triangle wave, but we can refer to the time expression of such a wave to design a new function:

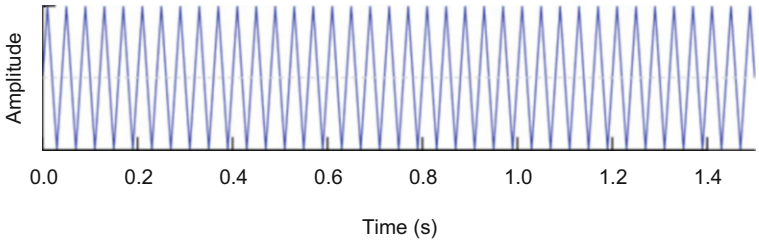
$$s(t) = A\frac{2}{\pi}\sin^{-1}(\sin(2\pi f_c t))$$

where A is the amplitude, t is time in s, and f_c is the carrier frequency in Hz. We coin a new function named `triangle()` with four arguments: (1) `d` for the duration of the wave in s, (2) `f` for the sampling frequency in Hz, (3) `fc` for the carrier frequency, and (4) `A` for the amplitude:

```
triangle <- function(d, f, fc, A){
t <- seq(0, d, length=d*f)
s <- A*(2/pi)*asin(sin(2*pi*fc*t))
return(s)
}
```

We now test and plot the function for a $d = 1.5$ s wave sampled at $f_s = 44,100$ Hz, with a carrier frequency of $f_c = 25$ Hz and an amplitude $A = 2$:

```
s <- triangle(d=1.5, f=44100, fc=25, A=2)
range(s)
[1] -1.999909  1.999970
oscillo(s, f=44100, colwave="blue")
```



18.4 Sinusoidal Sound: Additive Synthesis

18.4.1 Principle

If we refer to the compact expression of the Fourier series, a wave $s(t)$ can be decomposed into a sum of sines (or cosines) following (Sect. 9.2.3):

$$\begin{aligned} s(t) &= C_0 + \sum_{n=1}^{\infty} C_n \sin(\omega_n t + \varphi_n) \\ &= C_0 + \sum_{n=1}^{\infty} C_n \sin(2\pi f_n t + \varphi_n) \end{aligned}$$

where C_0 is the DC offset, C_n is the amplitude of the sine, ω_n is the angular frequency, f_n the regular frequency, and φ_n the phase.

Additive synthesis is a direct application of these equations: it simply consists in adding sine functions with predefined amplitude C_n, frequency f_n, and phase φ_n, the DC offset C_0 being usually set to 0. However, some caution should be taken as unwanted effects may arise when combining sines with inadequate relative properties.

We have seen in Sect. 10.1.4.2 that amplitude modulations may generate frequency sideband series. The reverse is also true: two pure tones, i.e., two sinusoid sounds, with closed carrier frequencies, may generate an amplitude modulation when added. This phenomenon, known as frequency beating, is ruled out by the formula:

$$f_{\text{am}} = |f_1 - f_2|$$

where f_{am} is the beat frequency or frequency of the AM, f_1 the frequency of the first pure tone, and f_2 the frequency of the second pure tone. As illustrated in Fig. 18.5, the addition of two pure tones with $f_1 = 50\,\text{Hz}$ and $f_2 = 55\,\text{Hz}$ generates a sound with two frequency bands at 50 and 55 Hz modulated in amplitude at a frequency of $55 - 50 = 5\,\text{Hz}$.

Similarly, attention should be paid to the phase of the sounds to be added as it can result in constructive or destructive interference (see Sect. 2.2.4). If we have two pure tones, $s_1(t)$ and $s_2(t)$, defined by the same angular frequency ω but potentially different amplitudes, A and B, and different phases, α and β:

$$s_1(t) = A \sin(\omega t + \alpha)$$

$$s_2(t) = B \sin(\omega t + \beta)$$

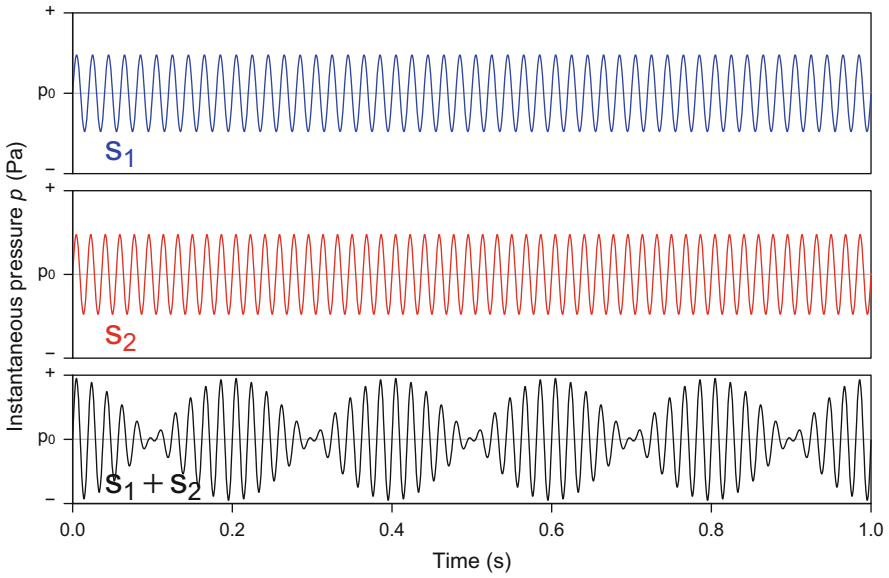


Fig. 18.5 Frequency beating. Beating can arise when adding pure tones closely related in frequency. The addition of two pure tones with carrier frequencies of 50 and 55 Hz generates a sound with an amplitude modulation of 5 Hz

If $\alpha = \beta$, then the sum of $s_1(t)$ and $s_2(t)$ is:

$$s_{1+2}(t) = \sqrt{A^2 + B^2 + 2AB} \times \sin(\omega t + \alpha)$$

The amplitude of the result is then reinforced through constructive interference. However, when a phase shift occurs between $s_1(t)$ and $s_2(t)$, then the result obeys to:

$$s_{1+2}(t) = \sqrt{a^2 + b^2} \times \sin\left(\omega t + \tan^{-1}\frac{b}{a}\right)$$

with

$$a = A\cos(\alpha) + B\cos(\beta)$$

and

$$b = A\sin(\alpha) + B\sin(\beta).$$

For instance, with $A = 1$, $B = 2$, $\alpha = 0$, and $\beta = \pi \div 2$, the result has an amplitude of 2.236 and a phase of 1.107 rad. In the particular case of the addition of two pure tones out of phase, that is, with a phase shift of $|\alpha - \beta| = \pi$ rad, then the addition returns a sound with an amplitude of 0, that is, silence. This destructive interference is actually an issue in audio recording if a pair of microphones receives

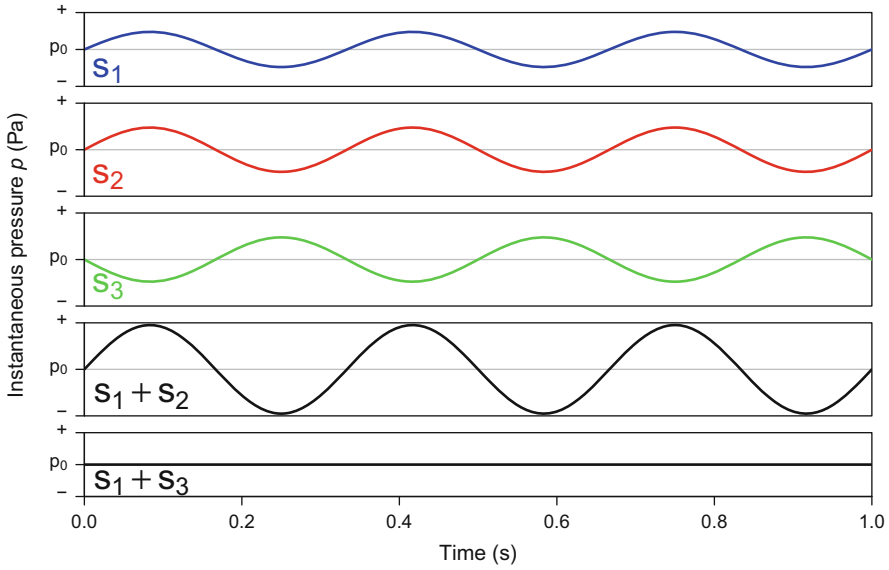


Fig. 18.6 Constructive and destructive interference. The pure tones s_1 and s_2 have a similar frequency of 3 Hz and are in phase, whereas s_1 and s_3 have also a frequency of 3 Hz but are out of phase that is an absolute phase shift of π rad. The sum of s_1 and s_2 returns a reinforced sound due to constructive interference. The sum of s_1 and s_3 leads to a null sound due to destructive interference

two sounds out of phase but is an advantage in active noise control as an opposition of phase may reduce noise (Fig. 18.6).

18.4.2 In Practice with `tuneR`

The function `sine()` of `tuneR` produces a sinusoidal wave, or pure tone, at a specific frequency expressed in Hz provided to the argument `freq`. The object returned is of class `Wave`. The simplest way to call `sine()` to produce a 440 Hz sound is:[10]

```
s <- sine(440)
```

This produces an IEEE 32 bit mono `Wave` object lasting 1 s with a sampling frequency $f_s = 44{,}100$ Hz:

[10] `synth-sine-440-1.wav`.

```
s

Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   FALSE
Bit (8/16/24/32/64):    32
```

We can apply some changes affecting the duration, sampling frequency, digitization depth, number of channels, and format[11]:

```
s <- sine(440, duration=2, samp.rate=22050, xunit="time",
          bit=16, pcm=TRUE, stereo=TRUE)
s

Wave Object
Number of Samples:      44100
Duration (seconds):     2
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Stereo
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

We can use `stereo()` to generate two different channels as seen in Sect. 6.2[12]:

```
left <- sine(440)
right <- sine(880)
s <- stereo(left, right)
s

Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Stereo
PCM (integer format):   FALSE
Bit (8/16/24/32/64):    32
```

[11] `synth-sine-440-2.wav`.

[12] `synth-sine-440-880-stereo.wav`.

The wonderful thing with `Wave` objects is that they are accompanied with arithmetic methods so that they can be combined through arithmetic operations. Here is a direct way to generate a simple harmonic series:[13]

```
cf <- 440
s <- sine(cf) + 0.75*sine(2*cf) + 0.5*sine(3*cf) + 0.25*sine(4*cf)
```

Here is another sum of harmonics that generates a waveform tending toward a square shape (Fig. 18.7):[14]

```
cf <- 440
s <- sine(cf) + 1/3*sine(3*cf) + 1/5*sine(5*cf) + 1/7*sine(7*cf)
```

So far all the sounds synthesized were characterized by a rectangular amplitude envelope. However, this amplitude envelope can be modified with `setenv()` or `drawenv()` (see Sect. 15.1) or `fadew()` (see Sect. 6.5.3). A raw way to change the envelop is to use arithmetics by multiplying the sine sound with a time function describing the envelope. In the following we modify a 440 Hz sound by applying a linear, exponential, or sinusoid increase in amplitude:[15]

```
s <- sine(440)
m <- max(s@left)
n <- length(s@left)
# linear increase
e.lin <- seq(0, m, length.out=n)
s.lin <- s*e.lin
# exponential increase
e.exp <- 1 - exp(seq(0, 4*m, length.out=n))
s.exp <- s*e.exp
# sinusoidal increase
e.sin <- sin(seq(0, pi/2, length.out=n))
s.sin <- s*e.sin
```

[13] `synth-sine-harmonics-1.wav`.

[14] `synth-sine-harmonics-2.wav`.

[15] `synth-sine-linear-amplitude.wav`, `synth-sine-exponential-amplitude.wav`, `synth-sine-sinusoidal-amplitude.wav`.

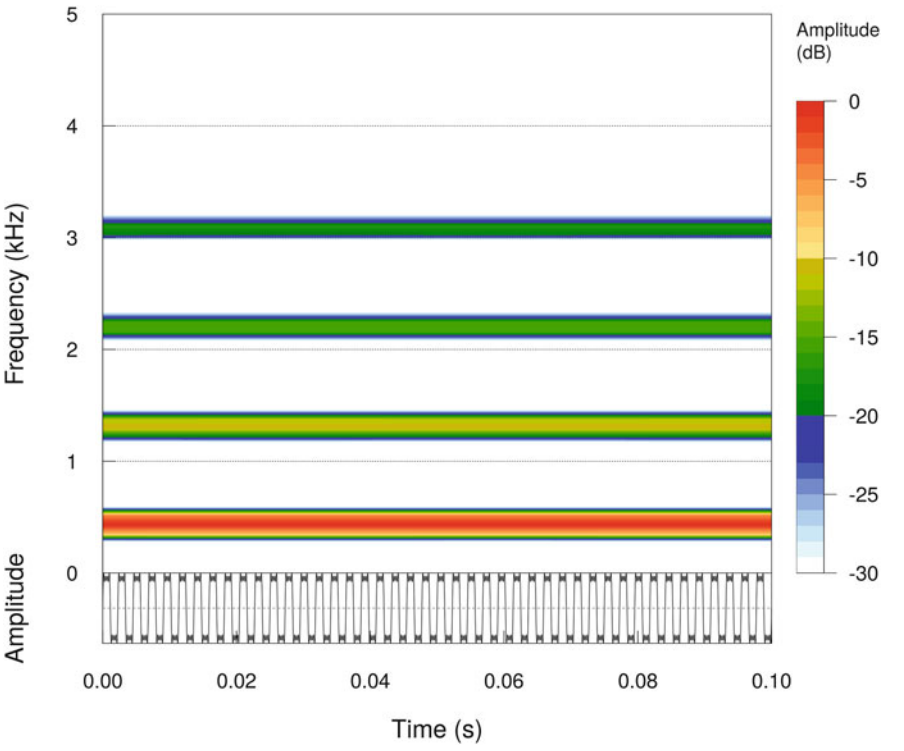


Fig. 18.7 Synthesis of an harmonic series. This series leads to a waveform with a square-like shape. The figure was produced calling `spectro()` using the arguments `tlim` and `flim` to zoom in time and frequency. Fourier window size = 512 samples, overlap = 0%, Hanning window

We can plot the results with (Fig. 18.8):

```
par(mfrow=c(3,1))
oscillo(s.lin)
oscillo(s.exp)
oscillo(s.sin)
```

18.4.3 In Practice with `seewave`

`seewave` has a function to generate sine sound (pure tone) as well. The function, named `synth()`, has a few options to generate pure or harmonic tones. The

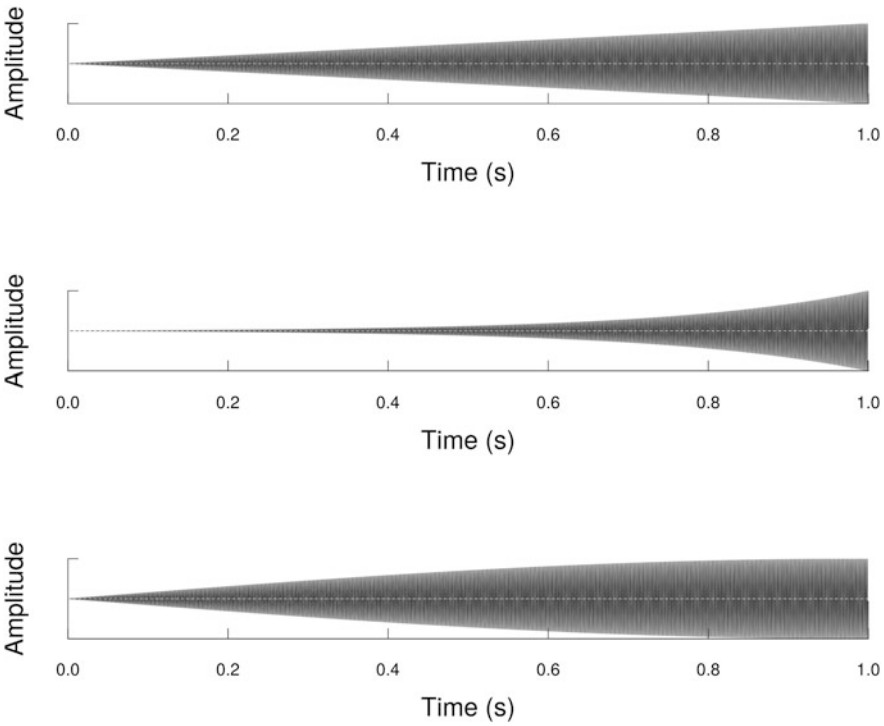


Fig. 18.8 Synthesis of a sine wave with amplitude envelop changes. A 440 Hz sine sound was synthesized using `sine()` and multiplied with an amplitude envelope following a linear (top), exponential (middle) and sinusoid (bottom) increase

best way to understand how `synth()` works is to review its main arguments in reference to the formula of a sine wave:

$$s(t) = A \sin(2\pi f_c t + \varphi)$$

`f` the sampling frequency f_s expressed in Hz,
`cf` the carrier frequency f_c expressed in Hz,
`a` the maximum amplitude A,
`signal` a character vector to specify the shape of the waveform, either `"square"`, `"tria"`, or `"saw"` for a square, triangular, or saw waveform respectively,
`shape` a character vector to specify the shape of the amplitude envelope, either `"incr"`, `"decr"`, `"sine"`, and `"tria"` for a linear increase, linear decrease, sinusoid shape, or triangular shape,
`p` the phase φ expressed in rad,
`harmonics` a numeric vector specifying the number and the relative amplitude of possible harmonics,

`listen` a logical to listen directly to the sound synthesized,
`plot` a logical to view the sound synthesized as a spectrogram, the arguments of `spectro()` can also be passed to control the plot parameters,
`output` a character vector to choose the class of the object to be returned, either `"matrix"`, `"Wave"`, `"Sample"`, `"audioSample"`, `"sound"`, or `"ts"`.

The most basic use of `synth()` is the following command that generates a 440 Hz sine wave sampled at 44,100 Hz and lasting 1 s:

```
s <- synth(f=44100, d=1, cf=440, output="Wave")
s

Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

We can change the maximum amplitude to 10, the phase to $\pi \div 2$, and the shape of the waveform to a square shape and apply a linear increase with:[16]

```
s <- synth(f=44100, d=1, cf=440, a=10, p=pi/2,
           signal="square", shape="incr", output="Wave")
```

The argument `harmonics` of `synth()` simplifies the process of additive synthesis by generating automatically an harmonic series. The argument waits a numeric vector which length corresponds to the number of harmonics+1, given that the first element of the vector corresponds to the fundamental frequency. The values provided to `harmonics` are the relative amplitudes of each harmonic including the fundamental. The value for the fundamental must equal to 1. For instance, setting `cf=500` and `harmonics = c(1, 0.5, 0.25)` produces a sound with three frequency bands (fundamental at 500 Hz + 1 harmonic at 1000 Hz and 1 harmonic at 1500 Hz), the second harmonic having an amplitude half the fundamental amplitude and the second harmonic an amplitude a quarter of the fundamental amplitude (Fig. 18.9 top-left):[17]

[16] `synth-sine-440-3.wav`.
[17] `synth-sine-harmonics-3.wav`.

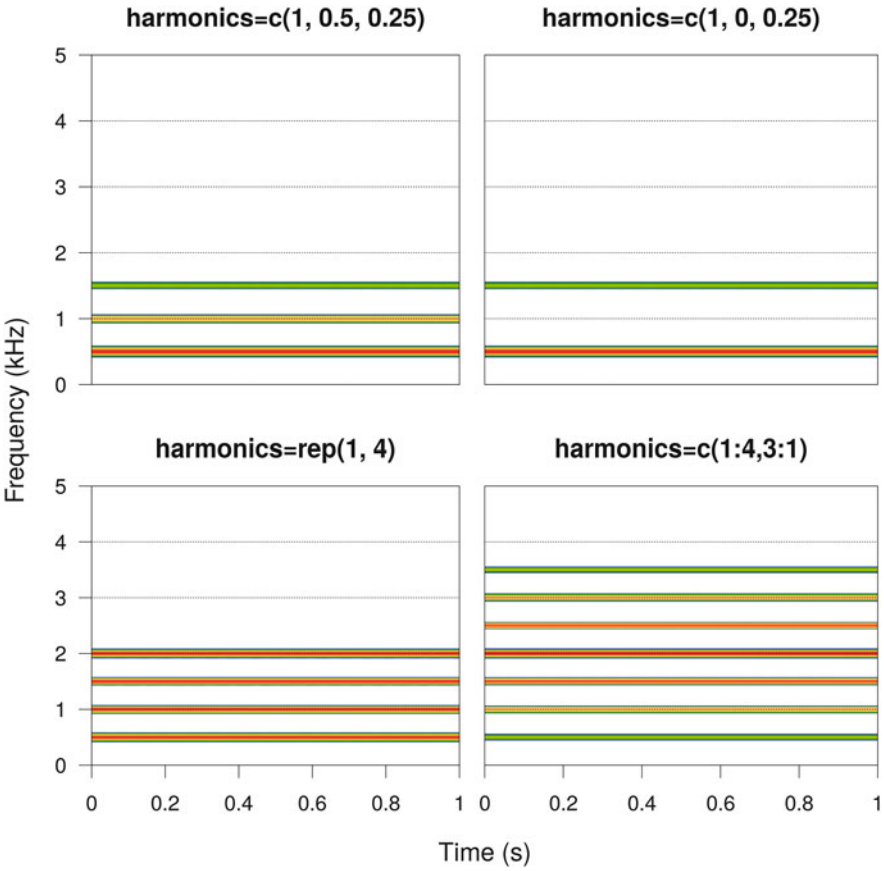


Fig. 18.9 Synthesis of harmonic series. Four examples of use of the argument `harmonics` of `synth()`. See text for details. Fourier window size = 1024 samples, overlap = 0%, Hanning window, frequency zooming between 0 and 5 kHz

```
s <- synth(f=44100, d=1, cf=500,
           harmonics=c(1, 0.5, 0.25), output="Wave")
```

Setting `harmonics = c(1, 0, 0.25)` generates two frequency bands (fundamental at 500 Hz + 1 harmonic at 1500 Hz) as the first harmonic has a null relative amplitude (Fig. 18.9 top-right):[18]

[18] `synth-sine-harmonics-4.wav`.

```
s <- synth(f=44100, d=1, cf=500,
           harmonics=c(1, 0, 0.25), output="Wave")
```

With `harmonics = rep(1,4)`, four harmonics of equal amplitude are produced (fundamental at 500 Hz + 3 harmonics at 1000 Hz, 1500 Hz, and 2000 Hz respectively) (Fig. 18.9 bottom-left):[19]

```
s <- synth(f=44100, d=1, cf=500,
           harmonics=rep(1, 4), output="Wave")
```

Values superior to 1 should be used to have harmonics with an amplitude higher than the amplitude of the fundamental. In the following last example, the third harmonic at 2000 Hz is set to be the dominant frequency (Fig. 18.9 bottom-right):[20]

```
s <- synth(f=44100, d=1, cf=500,
           harmonics=c(1:4,3:1), output="Wave")
```

The function `getRolloff()` of `soundgen` can be extremely useful to generate nice harmonic series. The function returns the relative amplitude in [0, 1] of successive harmonics according to roll-off parameters. In the following example, a steady exponential −12 dB/octave roll-off is specified above 500 Hz with a maximum of 20 harmonics:[21]

```
rolloff <- getRolloff(pitch_per_gc=500, nHarmonics=20,
                      rolloff=-12)
head(rolloff)
                [,1]
1 1.0000000000000000
2 0.3438854545349359
3 0.1833601137104065
4 0.1150234563281094
5 0.0788764935835791
6 0.0572235078486934
s <- synth(f=44100, d=1, cf=500,
           harmonics=rolloff, output="Wave")
```

[19] `synth-sine-harmonics-5.wav`.

[20] `synth-sine-harmonics-6.wav`.

[21] `synth-sine-harmonics-7.wav`.

18.5 Sinusoidal Sound: Modulation Synthesis

18.5.1 Principle

As detailed in Sect. 2.2.7, a sinusoidal sound $s(t)$ with both amplitude and frequency modulations follows the equation:

$$s(t) = A \times a(t) \times \sin(f(t) + 2\pi f_c t + \varphi)$$

where A is the maximum amplitude, $a(t)$ the amplitude modulation in respect with time (AM), $f(t)$ the frequency modulation in respect with time (FM), f_c the carrier frequency in Hz, and φ the phase in rad. Playing with the AM and FM time functions, $a(t)$ and $f(t)$, opens several possibilities in terms of synthesis. We will first review one function of the package `signal` to generate a chirp and then the options offered by `synth()` of `seewave`.

18.5.2 In Practice with `signal`

The package `signal` comes with a `chirp()` function that can generate a numeric vector corresponding to a chirp wave. A chirp wave, or sweep wave, is a pure tone sound modulated in frequency. The frequency modulation can follow a linear, a quadratic, or a time logarithmic function. In the following we produce a 1 s chirp wave sampled at $f_s = 44{,}100$ Hz starting at 5000 Hz and stopping at 15,000 Hz with three different FMs (Fig. 18.10)[22]:

```
library(signal)
f <- 44100
f0 <- 5000 ; f1 <- 15000 ; t1 <- 1
t <- seq(0, 1, length.out=f)
chirp.lin <- chirp(t=t, f0=f0, t1=t1, f1=f1, form="linear")
chirp.qua <- chirp(t=t, f0=f0, t1=t1, f1=f1, form="quadratic")
chirp.log <- chirp(t=t, f0=f0, t1=t1, f1=f1, form="logarithmic")
```

18.5.3 In Practice with `seewave`

There are two additional arguments in `synth()` that can control the amplitude modulation $a(t)$ (AM) and the frequency modulation $f(t)$ (FM) (Fig. 18.11).[23] As

[22] `synth-chirp-linear.wav`, `synth-chirp-quadratic.wav`, `synth-chirp-logarithmic.wav`.

[23] `synth-am-fm-1.wav`.

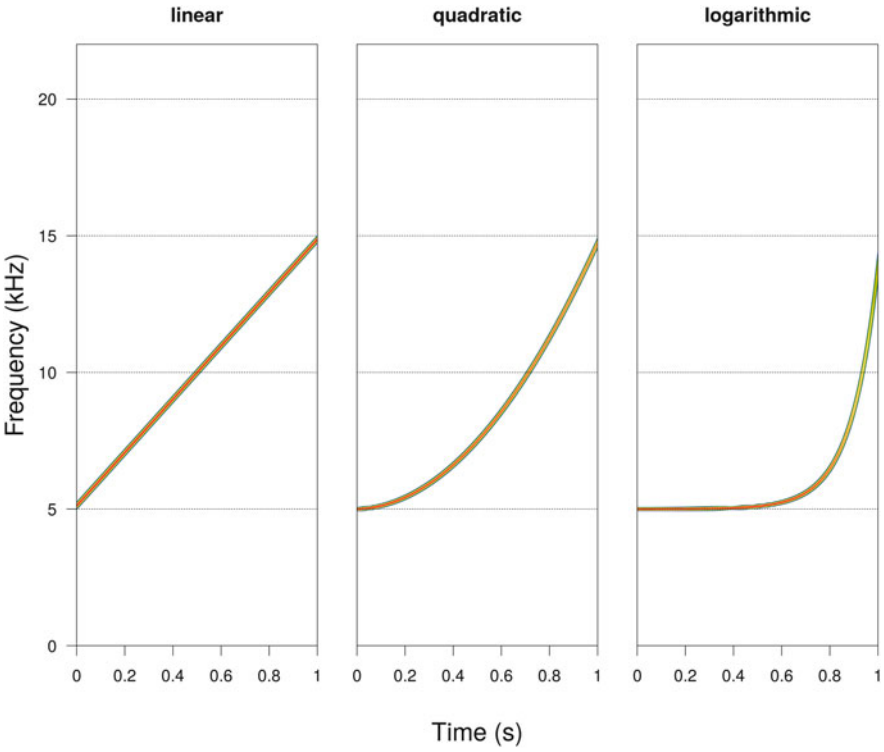


Fig. 18.10 Synthesis of chirps. Linear, quadratic and logarithmic chirps were synthesized with `chirp()` and visualized with `spectro()`. Fourier window size = 1024 samples, overlap = 87.5%, Hanning window

seen in Sect. 2.2.8, an AM can be defined by its depth m, also called modulation rate, that varies between 0 (no modulation) to 100 (maximum modulation) expressed in percentage, its rate or frequency f_{am}, and its phase φ_{am}. These three parameters can be specified in the argument `am` which is a numeric vector of length 3:

`am[1]` the depth m of the AM expressed in %,
`am[2]` the frequency f_{am} of the AM expressed in Hz,
`am[3]` the phase φ_{am} of the AM expressed in rad.

As introduced as well in Sect. 2.2.8, a FM can follow a sinusoidal, linear, or even an exponential function. In the first case, the FM is defined by its own frequency f_{fm}, its phase φ_{fm}, and its modulation index $\beta = \Delta f_c \div f_{\text{fm}}$. These parameters can be controlled with the first, second, and fourth elements of `fm` which is a numeric vector of length 5. A linear FM can be parametrized by its maximum excursion defined in the third element of `fm`. An exponential FM can also be applied using the fifth element of `fm`. Here is a review of the five elements of `fm`:

`fm[1]` the peak frequency deviation Δf_c. This value, expressed in Hz, is the difference between the carrier frequency and either the minimum or maximum

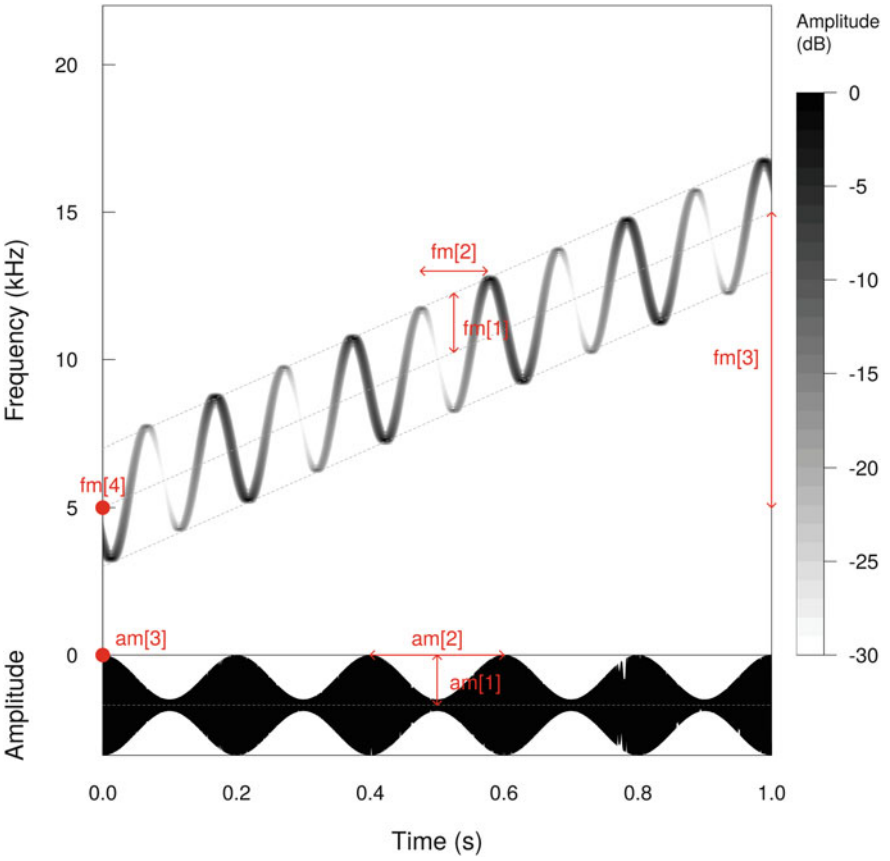


Fig. 18.11 Modulation synthesis: parameters of `synth()`. The arguments `am` and `fm` control the amplitude modulation (AM) and frequency modulation (FM) parameters. Each parameter is labeled according to the element position in the argument. For instance, `fm[2]` indicates the second element of the argument `fm`, that is, the frequency deviation of the sinusoid FM. The sound used as an example combines a sinusoid AM, a positive linear FM, and a sinusoid FM. The sound was synthesized with `synth(f=44100, d=1, cf=5000, fm=c(2000, 10, 10000, pi/2), am=c(80, 5, pi/2))`. Fourier window size = 1024 samples, overlap = 87.5%, Hanning window

frequency reached by the FM, that is, $\Delta f_c = f_{max} - f_{cf} = f_{cf} - f_{min}$. The value, which can be positive only, is therefore half the range of the sinusoid FM. A sinusoid FM covering 1000 Hz will be then defined with $\Delta f_c = 500$ Hz,

`fm[2]` the frequency of a sinusoidal FM f_{fm} expressed in Hz,

`fm[3]` the maximum excursion δf_c of a linear frequency modulation expressed in Hz. This value defines the difference between the start and the end of the FM, or the range of the FM. The value can be either positive (linear increase) or negative (linear decrease). A linear FM starting at 500 Hz and stopping at 2000 will have a maximum excursion of $\delta f_c = 2000 - 500 = 1500$ Hz,

`fm[4]` the phase of a sinusoidal frequency modulation φ_{fm} expressed in rad,
`fm[5]` the maximum excursion Δf_c of an exponential frequency modulation expressed in Hz. This is similar to the element 3 except that the modulation follows an exponential function. This can be used to synthesize a chirp.

Here are some basic examples of `synth()` making use of the arguments `am` or `fm`:[24]

```
# 10 Hz AM with phase changed
s <- synth(f=44100, d=1, cf=2000, am=c(80,10,pi/2))
# + 10 kHz linear FM
s <- synth(f=44100, d=1, cf=5000, fm=c(0,0,10000,0,0))
# - 10 kHz linear FM
s <- synth(f=44100, d=1, cf=15000, fm=c(0,0,-10000,0,0))
# sinusoid FM
s <- synth(f=44100, d=1, cf=5000, fm=c(2000,10,0,0,0))
# sinusoid and linear FM
s <- synth(f=44100, d=3, cf=2000, fm=c(2000,10,15000,0,0))
# sinusoid and exponential FM
s <- synth(f=44100, d=3, cf=2000, fm=c(2000,10,0,0,15000))
```

The arguments can be of course used together as in:[25]

```
s <- synth(f=44100, d=1, cf=2000,
           am=c(80,10,pi/2),
           fm=c(2000,10,10000,0,0))
```

The arguments of `synth()` previously detailed in Sect. 18.4.3 can also be called to tune a bit more the shape of the synthetic sound. Here we produce a sound modulated in amplitude and frequency with harmonics and with a change of the overall amplitude envelope. The spectrographic display is controlled by passing arguments of `spectro()` to `synth()` (Fig. 18.12):[26]

```
s <- synth(f=44100, d=3, a=2, cf=440,
           shape="sine",
           am=c(80,10,pi/2), fm=c(1000,10,0,0,8000),
```

(continued)

[24] `synth-am-fm-2.wav`, `synth-am-fm-3.wav`, `synth-am-fm-4.wav`, `synth-am-fm-5.wav`, `synth-am-fm-6.wav`, `synth-am-fm-7.wav`.
[25] `synth-am-fm-8.wav`.
[26] `synth-am-fm-9.wav`.

```
            harmonics=seq(1,0.2, length=3),
            plot=TRUE, osc=TRUE,
            wl=1024, ovlp=87.5,
            output="Wave")
```

18.5.4 Examples

Here follow a few practical examples making use of `tuneR` and `seewave` functions to synthesize sounds.

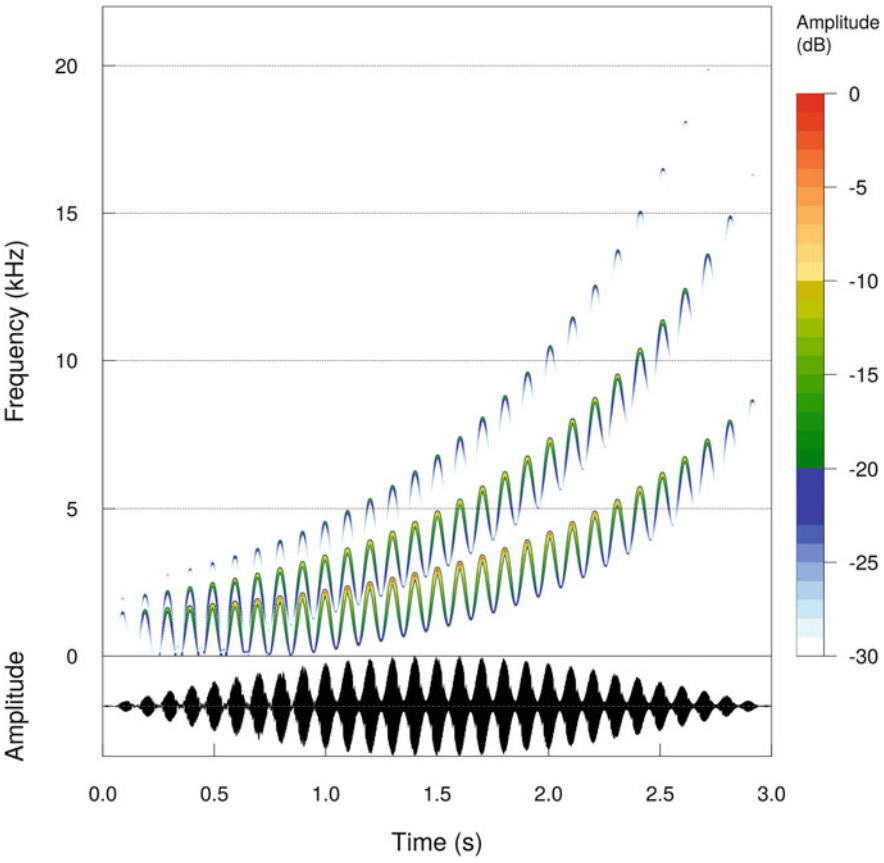


Fig. 18.12 Modulation synthesis full example with `synth()`. The sound was generated using most of the arguments of `synth()`. The display was directly produced with `plot=TRUE`. Fourier window size = 1024 samples, overlap = 87.5%, Hanning window

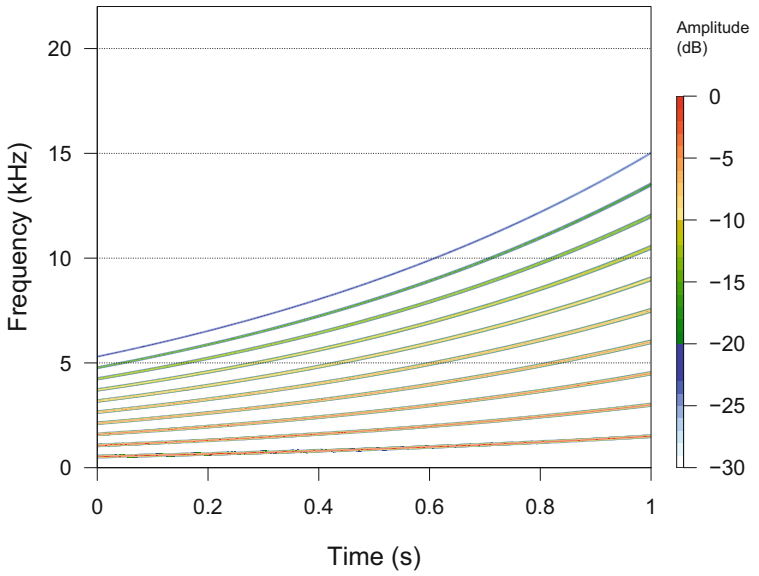


Fig. 18.13 Synthesis of an exponential chirp with harmonics. The sound was generated using the arguments `fm` and `harmonics` of `synth()`. Fourier window size = 1024 samples, overlap = 87.5%, Hanning window

18.5.4.1 Exponential Chirp

We can generate a chirp based on the C Western musical note with harmonics by using the fifth element of the argument `fm` and the argument `harmonics` of `synth()` (Fig. 18.13):[27]

```
s <- synth(f=44100, d=1, cf=notefreq("C", octave=4),
           fm=c(0,0,0,0,1000), har=seq(1,0.1,by=-0.1),
           plot=TRUE, wl=1024, ovlp=87.5, output="Wave")
```

We can also combine chirps using simple arithmetics. Here is the weighted addition of a first chirp sweeping up in frequency from 5000 to 10,000 Hz and a second chirp sweeping down from 10,000 to 5000 Hz (Fig. 18.14):[28]

[27] `synth-chirp-harmonics.wav`.

[28] `synth-chirp-combination.wav`.

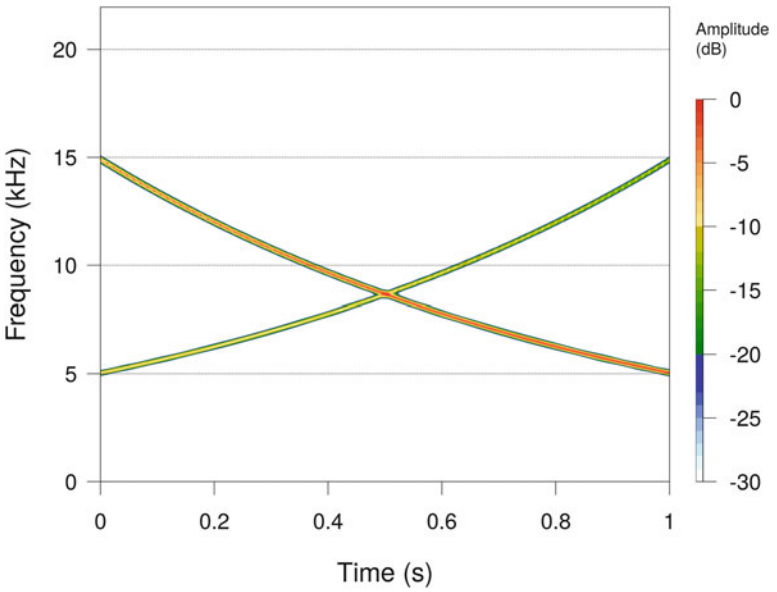


Fig. 18.14 Synthesis of a combination of exponential chirps. The sound was generated using the argument `fm` of `synth()` and the addition of two synthetic sounds. Fourier window size = 512 samples, overlap = 0%, Hanning window

```
s1 <- synth(f=44100, d=1, cf=5000,
            fm=c(0,0,0,0,10000), output="Wave")
s2 <- synth(f=44100, d=1, cf=15000,
            fm=c(0,0,0,0,-10000), output="Wave")
s <- s1+2*s2
spectro(s)
```

18.5.4.2 Synthesis of a Sideband Series with an AM

We have seen in Sect. 10.1.4.2 that a pure tone modulated in amplitude can be made of additional frequency bands or frequency sidebands. We can check this fact by using the argument `am` of `synth()`. We generate four amplitude modulated sounds which differ in their AM depth m expressed in % and in their AM frequency f_{am} expressed in Hz and (Fig. 18.15).[29]

The main shared parameters are:

[29] `synth-AM-sidebands-1.wav`, `synth-AM-sidebands-2.wav`, `synth-AM-sidebands-3.wav`, `synth-AM-sidebands-4.wav`.

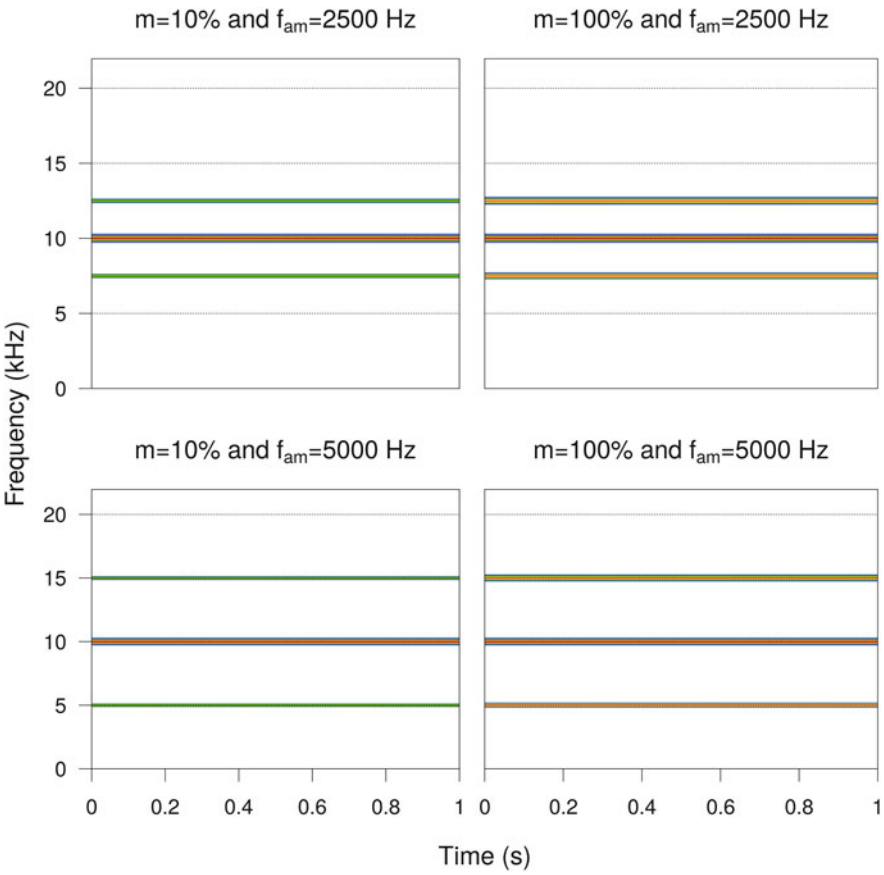


Fig. 18.15 Synthesis of AM waves. Four AM waves differing in the depth (m) and frequency (f_{am}) of the AM. These AM waves are characterized by frequency sidebands. Fourier window size = 512 samples, overlap = 0%, Hanning window, dynamic range = 60 dB

```
d <- 1        # duration
f <- 44100    # sampling frequency
cf <- 5000*2  # carrier frequency
out <- "Wave" # output class
```

AM sound with $m = 10\%$ and $f_{\text{am}} = 2500$ Hz:

```
s1 <- synth(f=f, d=d, cf=cf, am=c(10,2500,0), output=out)
```

AM sound with $m = 100\%$ and $f_{\text{am}} = 2500\,\text{Hz}$:

```
s2 <- synth(f=f, d=d, cf=cf, am=c(100,2500,0), output=out)
```

AM sound with $m = 10\%$ and $f_{\text{am}} = 5000\,\text{Hz}$:

```
s3 <- synth(f=f, d=d, cf=cf, am=c(10,5000,0), output=out)
```

AM sound with $m = 100\%$ and $f_{\text{am}} = 5000\,\text{Hz}$:

```
s4 <- synth(f=f, d=d, cf=cf, am=c(100,5000,0), output=out)
```

18.5.4.3 Synthesis of a Sideband Series with a FM

We have also detailed in Sect. 10.1.4.3 that FM sounds can produce frequency sidebands which frequency and amplitude are ruled out by the modulation index β defined as:

$$\beta = \frac{\Delta f_c}{f_{\text{fm}}}$$

where Δf_c is the peak frequency deviation and f_{fm} is the frequency of the FM. These two parameters are, respectively, the first and the second elements of the argument `fm` of `synth()`. We can therefore generate a sound made of complex sidebands with a few lines of code. The following examples show the results for four values of β (Fig. 18.16)[30]:

[30] `synth-FM-sidebands-1.wav`, `synth-FM-sidebands-2.wav`, `synth-FM-sidebands-3.wav`, `synth-FM-sidebands-4.wav`.

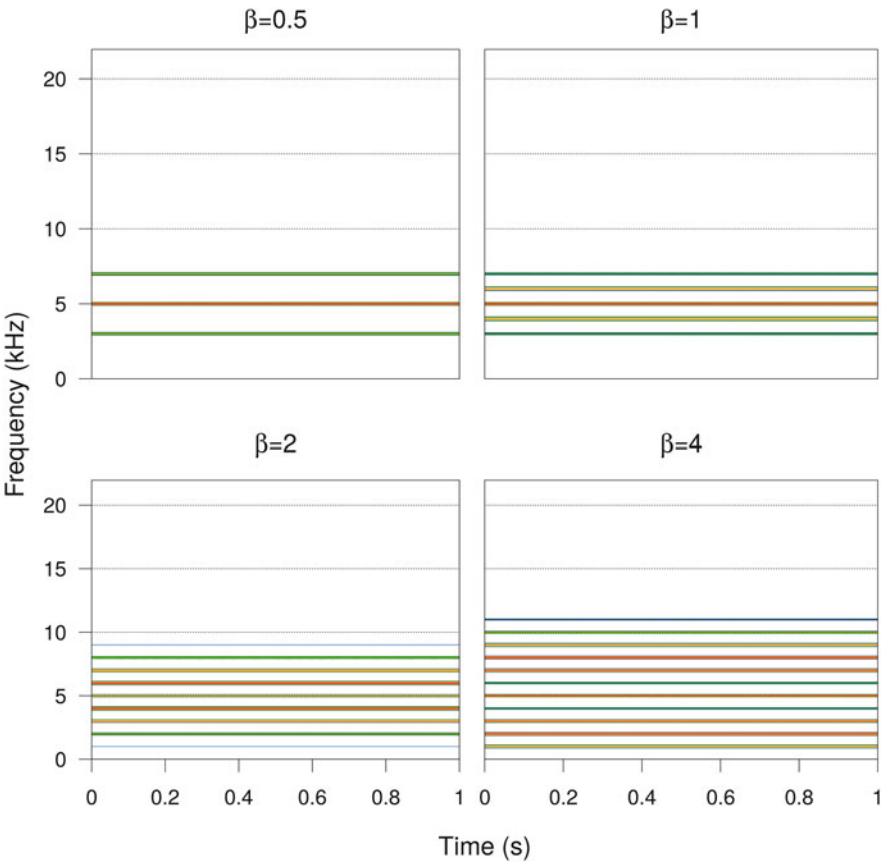


Fig. 18.16 Synthesis of FM waves. Four FM waves differing in their modulation index $\beta = \Delta f_c \div f_{\text{fm}}$ where Δf_c is the carrier frequency and (f_{fm}) is the frequency of the FM. These FM waves are characterized by complex frequency sidebands. Fourier window size = 512 samples, overlap = 0%, Hanning window

The main shared parameters are:

```
# main shared parameters
d <- 1            # duration
f <- 44100        # sampling frequency
cf <- 5000        # carrier frequency
out <- "Wave"     # output class
delta.fc <- 1000  # peak frequency deviation
f.fm <- 1000      # FM frequency
```

FM sound with $\beta = 1000 \div 2000 = 0.5$:

```
# beta = delta.fc/f.fm = 0.5
s1 <- synth(d=d, f=f, cf=cf,
            fm=c(delta.fc,f.fm*2,0,0,0), output=out)
```

FM sound with $\beta = 1000 \div 1000 = 1$:

```
# beta = delta.fc/f.fm = 1
s2 <- synth(d=d, f=f, cf=cf,
            fm=c(delta.fc,f.fm,0,0,0), output=out)
```

FM sound with $\beta = 2000 \div 1000 = 2$:

```
# beta = delta.fc/f.fm = 2
s3 <- synth(d=d, f=f, cf=cf,
            fm=c(delta.fc*2,f.fm,0,0,0), output=out)
```

FM sound with $\beta = 4000 \div 1000 = 4$:

```
# beta = delta.fc/f.fm = 4
s4 <- synth(d=d, f=f, cf=cf,
            fm=c(delta.fc*4,f.fm,0,0,0), output=out)
```

18.5.4.4 The Sound of π and Other Numbers

The first idea of this example is to change a number into a pure tone sound according to Western music scale. This means is that we should be able first to convert any numeric value into a frequency value that corresponds to the Western musical scale (see Sect. 9.4.2). We can get this frequency by applying the following formula:

$$f = f_c \times 2^{\frac{x-1}{12}}$$

where f_c is the carrier frequency of the reference note in Western musical scale and x is the numeric value to be converted into a musical frequency.

This equation can be transferred and tested for a A-440 Hz note and for x values in $\{0, 1, \ldots, 9\}$ with:

```
A <- 440
x <- 0:9
A*2^((x-1)/12)
 [1] 415.3047 440.0000 466.1638 493.8833 523.2511 554.3653
 [7] 587.3295 622.2540 659.2551 698.4565
```

We can now try to write a function, named `numsound()`, that applies this formula and synthesizes a short sound for each value of x. To do this, we need to specify the sampling frequency f_s in Hz of the output sound, the duration in s of each short sound associated to each value of x, and the frequency of the reference note in Hz. To make the function a bit more fancy, we add harmonics and we listen to the sound directly. We therefore end with a function with six arguments:

```
numsound <- function(
                    x,              # input numeric vector
                    f=44100,        # sampling frequency
                    d=0.1,          # duration of each sound
                    cf=440,         # carrier frequency
                    harmonics=1,    # harmonics amplitudes
                    listen=FALSE    # listen output
                    )

{
    # empty initial sound
    s <- Wave(left=numeric(0), samp.rate=f, bit=16)
    # loop around x
    for(i in 1:length(x))
    {
        # temporary sound corresponding to a single value of x
        tmp <- synth(f=f, d=d, cf=cf*2^((x[i]-1)/12),
                    shape="sine",
                    harmonics=harmonics,
                    output="Wave")
        # bind/paste the successive notes
        s <- bind(s,tmp)
    }
    # output with listen option
    if(listen) {listen(s) ; invisible(s)}
    else return(s)
}
```

We can now use the function with a simple vector starting at $440 \times 2^{(1-1)/12} = 440$ Hz and increasing up to $440 \times 2^{(40-1)/12} = 3951$ Hz and decreasing down back to $440 \times 2^{(1-1)/12} = 440$ Hz. Setting the argument `harmonics` with a numeric

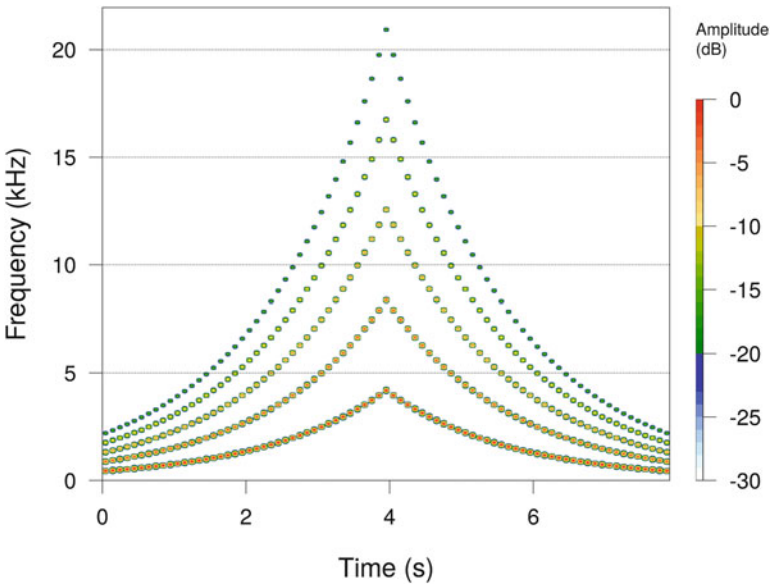


Fig. 18.17 Synthetic sound based on a numeric vector. The sound was generated using the handmade function `numsound()`. Fourier window size = 512 samples, overlap = 0%, Hanning window

vector of length 5 implies to find the highest harmonic at $5 \times 440 \times 2^{(40-1)/12} = 20,930$ Hz. The result can be visualized using `spectro()` (Fig. 18.17)[31]:

```
s <- numsound(c(1:40,39:1),
              harmonics=seq(1,0.2,length.out=5),
              listen=TRUE)
spectro(s)
```

We can now play with the function. For instance, we could wish to listen to the decimal values of gold numbers as π is. To do this, we first need to extract the decimal values of these peculiar numbers. There are no `R` function to achieve this. However, the base functions `format()`, `strsplit()` and `as.numeric()` can be used to manipulate the class of the object and to isolate the different elements of a numeric vector. We take care of increasing from 7 (default) to 15 the number of digits displayed by `R` using the base function `options()`:

[31] `synth-numsound-test.wav`.

```
options(digits=15)
get.digits <- function(x){
    as.numeric(strsplit(format(x),"")[[1]][-(1:2)])
    }
get.digits(1.12345678987654)
 [1] 1 2 3 4 5 6 7 8 9 8 7 6 5 4
```

We can apply this new function to gold numbers, as π, the golden ratio, the Euler number, and a rational number with decimals showing a repeating sequence:

```
pi.digits <- get.digits(pi)               # pi
gold.digits <- get.digits((1+sqrt(5))/2) # golden ratio
euler.digits <- get.digits(exp(1))        # Euler number
rational.digits <- get.digits(22/7)       # rational number
```

The results can be forwarded to `numsound()`:[32]

```
numsound(pi.digits, listen=TRUE)
numsound(gold.digits, listen=TRUE)
numsound(euler.digits, listen=TRUE)
numsound(rational.digits, listen=TRUE)
```

18.5.4.5 C Major Scale with a Clarinet Timbre

We can use the functions `notefreq()` (see Sect. 9.4.2) and `synth()` to generate a series of pure tones which frequencies follow the C major scale of Western music (Fig. 18.18). The sound created here[33] imitates the timbre of a clarinet by specifying an appropriate harmonic series:[34]:

[32] `synth-numsound-pi.wav`, `synth-numsound-golden-ratio.wav`, `synth-numsound-euler.wav`, `synth-numsound-rational.wav`.

[33] `synth-C-major-sale.wav`.

[34] http://www.phy.mtu.edu/ suits/clarinet.html

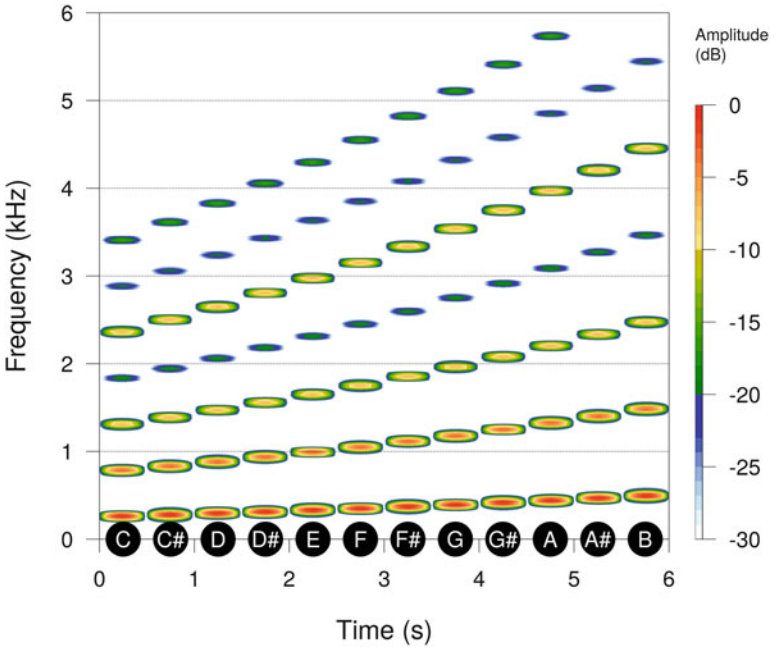


Fig. 18.18 Synthesis of C major scale notes. Synthesis of the 12 notes of the C major scale following Western music. Fourier window size = 4096 samples, overlap = 87.5%, Hanning window

```
# data
notes <- c("C","C#","D","D#","E","F","F#","G","G#","A","A#","B")
freq <- notefreq(notes)          # the frequency of the 12 notes
f <- 44100                       # sampling frequency
d <- 0.5                         # duration of each note
s <- Wave(left=numeric(0),       # empty initial sound
          samp.rate=f, bit=16)
# loop around the number of notes
for (i in 1:length(notes))
       {
         # temporary sound corresponding to each note
         tmp <- synth(d=d, f=f,
                      cf=freq[i],
                      har=c(1,0,0.75,0,0.5,0,0.14,
                            0,0.5,0,0.12,0,0.17),
                      shape="sine",
                      output="Wave")
         # bind/paste the successive notes
         s <- bind(s,tmp)
       }
```

(continued)

```
# visualization
spectro(s, f, flim=c(0,6), wl=1024, ovlp=87.5)
par(xpd=TRUE)
x <- seq(0.25,5.75,length=length(notes))
y <- rep(0, length(notes))
points(x, y, cex=3.5, pch=19)
text(x, y=y, labels=notes, col="white")
```

18.5.4.6 Shepard Scale

The Shepard scale is an ordered succession of tones, or musical notes, made of several frequency bands.[35] The frequency and amplitude of these bands can be fixed according to the following instructions: (1) the frequency of each band is twice the frequency of the band below so that bands are equally spaced on a logarithmic scale, and (2) the amplitude of the bands follow a density function of a normal distribution according to logarithmic frequency (Shepard 1964; Deutsch 2010). Through a phenomenon of circularity, the repeated play of such tone series gives the illusion of endlessly ascending tones. In the following example, we generate a Shepard scale made of six successive tones.

We first set the main time and frequency parameters of the sounds to be generated:

```
f <- 44100            # sampling frequency
mu <- 440             # mean of the normal distribution
lmu <- log(mu)/log(2) # log conversion
sigma <- 0.5          # s.d. of the normal distribution
n <- 5                # bands above the fundamental frequency
phi <- 1/10           # phase along the normal distribution
d.tone <- 0.12        # duration of the tones
d.sil <- 0.84         # duration of the silence between tones
```

We include a new function, $g(x)$, that generates the density function of the normal distribution knowing that the density function of $\mathcal{N}(\mu, \sigma)$ is written as:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

[35] This example was kindly provided by Laurent Lellouch.

```
g <- function (x) {
    (1/sqrt(2*pi*sigma^2))*exp(-(x-lmu)^2/(2*sigma^2))
}
```

We create the fundamental frequency of each tone with:

```
lf0 <- floor(lmu-n) + 0:5/6 + phi
f0 <- 2^lf0
f0
[1]  8.57418770029035  9.62420028865693 10.80279956934552
[4] 12.12573253208319 13.61067457521369 15.27746566256667
```

It is now time to generate the bands above the fundamental frequency. To achieve this, we use the function $g(x)$ created above that we include into a `for` loop so that the bands are generated for each tone:

```
h1 <- h2 <- h3 <- h4 <- h5 <- h6 <- NULL
for (i in 0:(2*n+1)){
    h1 <- c(h1, rep(0,2^i-1), g(lf0[1]+i))
    h2 <- c(h2, rep(0,2^i-1), g(lf0[2]+i))
    h3 <- c(h3, rep(0,2^i-1), g(lf0[3]+i))
    h4 <- c(h4, rep(0,2^i-1), g(lf0[4]+i))
    h5 <- c(h5, rep(0,2^i-1), g(lf0[5]+i))
    h6 <- c(h6, rep(0,2^i-1), g(lf0[6]+i))
}
```

We check that `h1` is actually the frequency spectrum of the first tone and that the frequency bands are equally spaced according to the log of the frequency (Fig. 18.19):

```
plot(x=seq(1, f/2, length.out=length(h1)), y=h1, col="blue",
     xlab= "log frequency (Hz)", ylab="Amplitude",
     type="l", log="x")
```

We write a short function that parses `synth()`, `normalize()`, and `addsilw()` to generate the tone wave and together with the silence bouts framing this wave. The function has only two arguments: (1) `cf` for the carrier or fundamental frequency and (2) `bands` for the frequency harmonics:

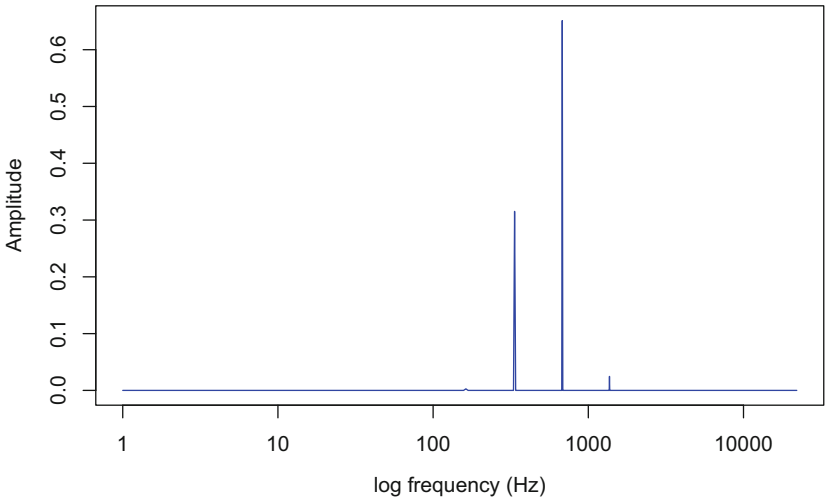


Fig. 18.19 Frequency spectrum of a Shepard scale tone. The bands are equally spaced along a log frequency scale

```
shepard <- function(cf, bands){
  s <- synth(f=f, d=d.tone, cf=cf, harmonics=bands,
             shape="sine", output="Wave")
  s <- normalize(s)
  s <- addsilw(s, f=f, at="start", d=d.sil/2, output="Wave")
  s <- addsilw(s, f=f, at="end", d=d.sil/2, output="Wave")
  return(s)
}
```

We call this function and generate a wave for each tone. The bands are divided by their first value for scaling purposes:

```
s1 <- shepard(cf=f0[1], bands=h1/h1[1])
s2 <- shepard(cf=f0[2], bands=h2/h2[1])
s3 <- shepard(cf=f0[3], bands=h3/h3[1])
s4 <- shepard(cf=f0[4], bands=h4/h4[1])
s5 <- shepard(cf=f0[5], bands=h5/h5[1])
s6 <- shepard(cf=f0[6], bands=h6/h6[1])
```

The six tones are then bounded in a specific order using `bind()`:

```
s <- bind(s4,s5,s6,s1,s2,s3)
```

The result can be visualized as a spectrogram (Fig. 18.20):

```
spectro(s, flim=c(0,5), wl=1024, ovlp=87.5)
```

Listening once and three times the Shepard scale demonstrates the pitch ascending illusion:[36]

```
listen(s)                                  # play once
listen(repw(s, times=3, output="Wave")) # play 3 times
```

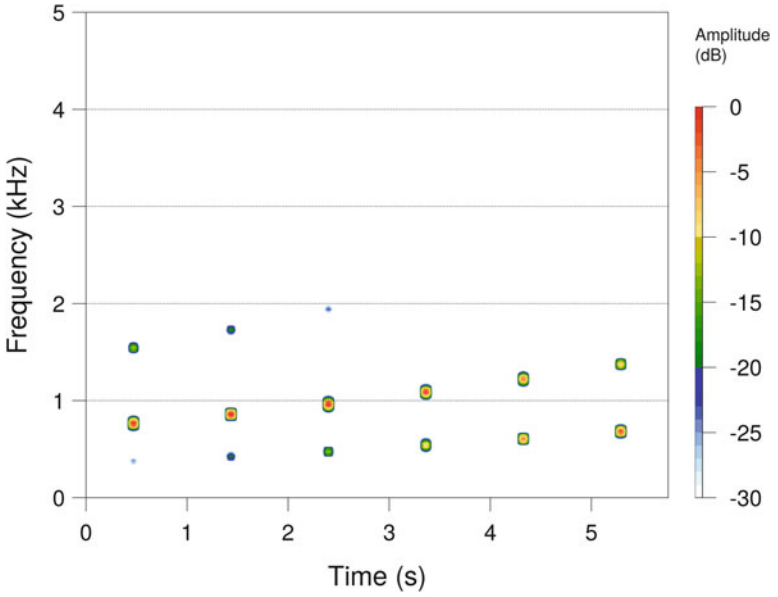


Fig. 18.20 Synthesis of a Shepard scale. Six tones, or notes, composed, ordered to create an illusion of endlessly ascending pitch when repeated. Frequency zoom in between 0 and 5 kHz. Fourier window size = 4096 samples, overlap = 87.5%, Hanning window

[36] `synth-shepard-scale.wav`.

18.5.4.7 Risset Glissando

Similar to Shepard scale illusion (Sect. 18.5.4.6), we can create a Risset glissando which gives the illusion to be endless (Risset 1978) (Fig. 18.21).[37]

The glissando is made by the superposition of identical chirp components shifted in time and whose amplitude and frequency follow the rules defined by the Shepard scale (Sect. 18.5.4.6). We will synthesize a glissando made of $n + p = 17$ components lasting $p \times d = 63$ `second` each, with a shift of $d = 7$ s. As usual, we initiate the process by defining the main parameters of the sound to be produced:

```
f <- 44100   # sampling frequency
cf <- 32     # carrier or fundamental frequency
n <- 8
p <- 9
d <- 7
mu <- 3.5    # mean of the normal distribution
sigma <- p*f # s.d. of the normal distribution
```

We first generate an exponential sweep using the fifth element of the argument `fm` of `synth()`. The sweep duration and the spectral width are set using the parameter `p`. Here the duration is `p*d=63` s, the carrier frequency f_c is `cf=32` Hz, and the frequency modulation increase equals $2^p - 1 * \texttt{cf} = 16,352$ Hz:

```
a <- synth(f=f, d=p*d, cf=cf, fm=c(0,0,0,0,(2^p-1)*cf))
```

We then aim at generating the frequency components following the rules of the Shepard scale. We use again the density function of a normal distribution:

```
l <- length(a)
g <- (1/sqrt(2*pi*sigma^2))*exp(-((1:l)-l/mu)^2/(2*sigma^2))
```

We first initiate a vector made of 0 values and we then use a `for` loop to add the successive components with a `d*f` time shift:

[37] This example was kindly provided by Laurent Lellouch.

```
s <- rep(0,(n+2*p)*d*f)
for (i in 0:(n+p)) {
    s <- s + c(rep(0,i*d*f), a*g, rep(0,(n+p-i)*d*f))
}
```

We trim the sound so that we eliminate time sections that have no interest for the acoustic illusion. This is also the occasion to coerce the numeric vector into a `Wave` object with the argument `output`:

```
s <- cutw(s, f=f, from=p*d, to=(n+p)*d, output="Wave")
```

We end up by adding fade-in and fade-out effects of duration `d=7` s:

```
s <- fadew(s, din=d, dout=d, output="Wave")
```

We look at the result with a simple call to `spectro()` (Fig. 18.21):

```
spectro(s, f, flim=c(0,5), wl=4096, ovlp=87.5,
        collevels=seq(-60,0,1))
```

And the acoustic illusion can be appreciated with:[38]

```
listen(s, f)
```

18.5.4.8 Imitation of Stridulation of the Tree Cricket *Oecanthus pellucens*

We introduced in Sect. 10.1 the stridulation of the Italian tree cricket (Fig. 10.1) which is stored in the dataset `pellucens`. We wish to produce a synthetic version of the second stridulation sequence that can be found between 2.1 and 3.2 s of the `Wave` object. This sequence is the repetition of short motifs also named echemes. Each echeme is characterized by a carrier frequency at 2300 Hz with a linear FM of −315 Hz. A second frequency band is found at 1.9 times the frequency of the

[38] `synth-risset-glissando.wav`.

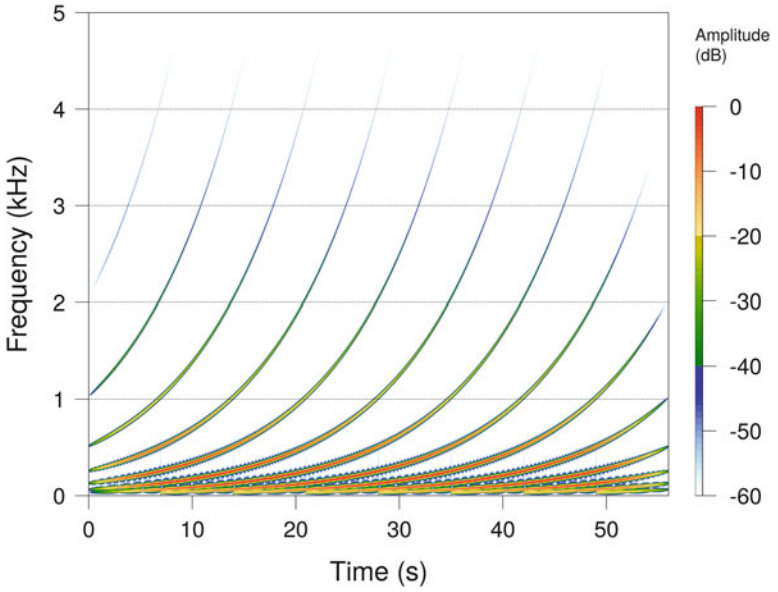


Fig. 18.21 Synthesis of a Risset glissando. Fourier window size = 4096 samples, overlap = 87.5%, Hanning window, dynamic range = 60 dB

carrier frequency with a relative amplitude of 0.12. The amplitude envelope of each echeme shows a sine-like shape. The echemes are repeated 20 times and separated by a period of silence of about 0.015 s. Finally, the complete sequence starts with a slow increase in amplitude that can be mimicked with a fade-in effect.

Here is the complete code to take into account all these features (Fig. 18.22)[39]:

```
d <- 0.03                              # echeme duration
f <- 11025                             # sampling frequency
cf <- 2300                             # carrier frequency
out <- "Wave"                          # output class
s1 <- synth(d=d, cf=cf, f=f,           # first frequency component
            fm = c(0,0,-315,0,0),      # (fundamental)
            shape="sine", output=out)
s2 <- synth(d=d, cf=1.9*cf, f=f,       # second frequency component
            fm=c(0,0,-315,0,0),        # (first harmonic)
            shape="sine", output=out)
s <- s1 + (0.12*s2)                    # weighted addition
```

(continued)

[39] `synth-oecanthus-pellucens.wav`.

```
s <- addsilw(s, d=0.015,              # adding a silence period
             at="end", output=out)
s <- repw(s, times=20, output=out)    # repetition of 20 echemes
s <- fadew(s, f=f, din=0.25,          # fade-in effect
           shape="cos", output=out)
s <- addsilw(s, d=0.1,                # silence before the sequence
             at="start", output=out)
s <- addsilw(s, d=0.1,                # silence after the sequence
             at="end", output=out)
```

18.5.4.9 Imitation of the Call of the Frog *Eleutherodactylus martinicensis*

The call of the Martinique Robber frog *Eleutherodactylus martinicensis* was used in Sect. 8.3.2 to test automatic time measurement. We here imitate the first four two-note vocalizations that occur between 1.6 and 5.4 s:

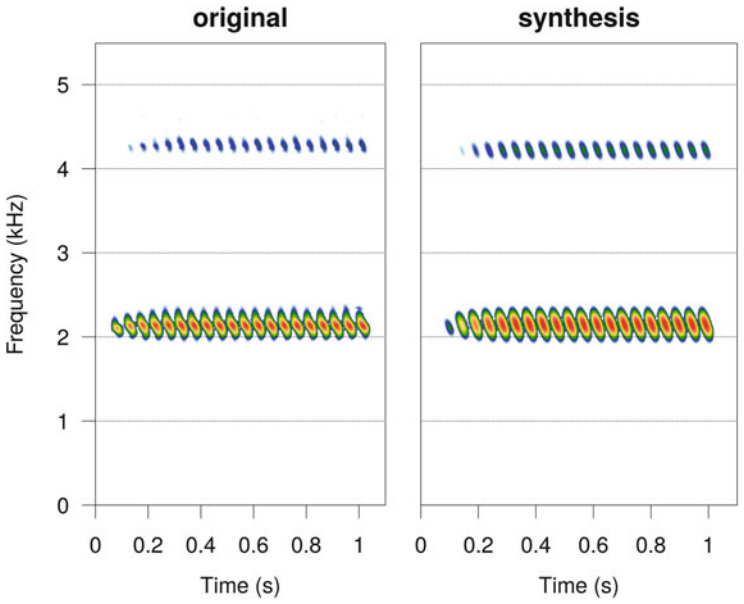


Fig. 18.22 Synthesis of the call of the tree cricket *Oecanthus pellucens*. Original (left) and synthesis (right) of one stridulation of the Italian tree cricket *Oecanthus pellucens*. Fourier window size = 512 samples, overlap = 87.5%, Hanning window

```
frog <- readWave("sample/Eleutherodactylus_martinicensis.wav",
                 from=1.6, to=5.4, unit="seconds")
```

By taking direct measurements on the oscillogram and spectrogram, we can estimate that the duration of the first note is about 0.21 s, the duration of the second note is about 0.17 s, and that two-note vocalizations are separated by a silent bout of approximately 0.3 s. In terms of frequency, the first note can be described as a linear FM sound starting at 1850 Hz and stopping at 2100 Hz, and the second note as a linear FM sound beginning as well at 2800 Hz and ending at 3750 Hz. This leads to the following script (Fig. 18.23):[40]

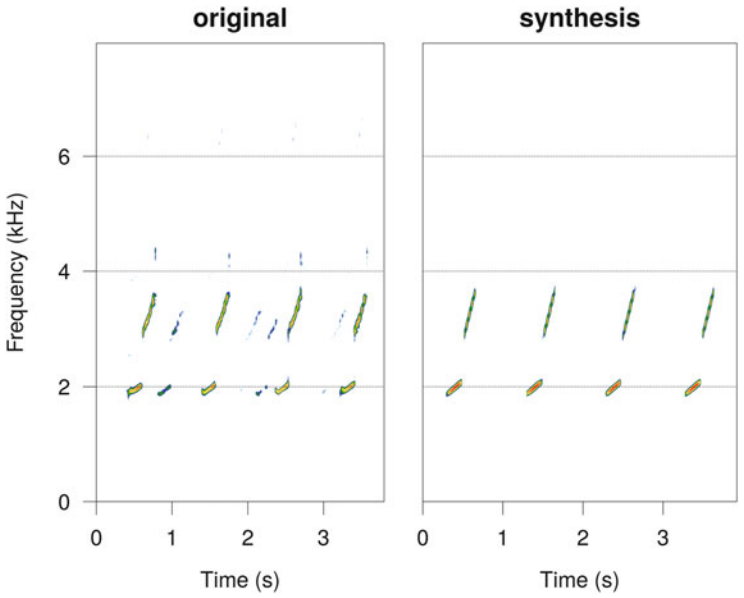


Fig. 18.23 Synthesis of the call of the frog *Eleutherodactylus martinicensis*. Original (left) and synthesis (right) of four two-note vocalizations of the Martinique Robber frog *Eleutherodactylus martinicensis*. Fourier window size = 512 samples, overlap = 0%, Hanning window

[40] `synth-eleutherodactylus_martinicensis.wav`.

```
f <- frog@samp.rate
# synthesis of the first note
s1 <- synth(f=f, d=0.21, cf=1850,
            fm=c(0,0,2100-1850,0,0), out="Wave")
# synthesis of the second note
s2 <- synth(f=f, d=0.17, cf=2800,
            fm=c(0,0,3750-2800,0,0), out="Wave")
# fade in and fade out to smooth the attack and tail
s1 <- fadew(s1, din=0.1, dout=0.05, shape="cos", output="Wave")
s2 <- fadew(s2, din=0.05, dout=0.05, shape="cos", output="Wave")
# bind the two notes
s <- bind(s1,s2)
# add silence at the start and end
s <- addsilw(s, at="start", d=0.3, out="Wave")
s <- addsilw(s, at="end", d=0.3, out="Wave")
# repeat 4 times the two-note vocalization
s <- repw(s, times=4, out="Wave")
```

18.6 Tonal Synthesis

18.6.1 Principle

Tonal synthesis, following Beeman (1998), essentially consists in combining instantaneous frequency (frequency contour) and instantaneous amplitude (amplitude envelope), to generate a tonal sound, that is, a sound made of a single frequency band. The instantaneous frequency and instantaneous amplitude can be generated *de novo* through mathematics functions (sine or others) or can derive from a Hilbert transform applied on a pre-existing sound (see Sects. 5.2.1 and 13.1.4.1). This latter option is actually often used to modify a sound as illustrated with `tico` in Sect. 15.5.

18.6.2 In Practice with `seewave`

Tonal synthesis is available with the `seewave` function `synth2()`. The two main arguments of this function are (1) `env` which waits a numeric vector describing the amplitude envelope either created with usual numeric generation tools or obtained with `env()` (see Sect. 5.2.2) and (2) `ifreq` which describes the instantaneous frequency in Hz with a numeric vector generated de novo or resulting from `ifreq()$f` (see Sect. 13.1.4.1). The numeric vectors given to `env` and `ifreq`

must have exactly the same length. Here follow three examples with a 1 s pure tone continuously beating at 2000 Hz but with different amplitude envelopes:[41]:

```
f <- 44100           # sampling frequency
output <- "Wave"     # output class
ifreq <- rep(2000,f) # 2000 Hz instantaneous frequency
# linear increase of the amplitude envelope
s <- synth2(env=seq(0,1,length=f),
            ifreq=ifreq, f=f, output=output)
# square-root increase of the amplitude envelope
s <- synth2(env=sqrt(seq(0,1,length=f)),
            ifreq=ifreq, f=f, output=output)
# square-root increase and decrease of the amplitude envelope
s <- synth2(
      env=c(sqrt(seq(0,1,length=f/2)), sqrt(seq(1,0,length=f/2))),
      ifreq=ifreq, f=f, output=output)
```

The next code illustrates modifications of the instantaneous frequency only, the default amplitude envelope following a rectangular shape. Each sound lasts similarly 1 s:[42]

```
f <- 44100           # sampling frequency
output <- "Wave"     # output class
# instantaneous frequency from 500 to 4000 Hz by step of 500 Hz
s <- synth2(ifreq=rep(seq(500,4000,by=500), each=f/8),
            f=f, output=output)
# instantaneous frequency following a x^2 function
s <- synth2(ifreq=4000 + seq(-75,75, length=f)^2,
            f=f, output=out)
```

As a last example the density function of a normal distribution is used to shape both the AM and FM (Fig. 18.24):[43]

```
f <- 44100           # sampling frequency
output <- "Wave"     # output class
norm <- dnorm(-(f/2):(f/2-1), sd=7000)
s <- synth2(env=norm, ifreq=5000+(norm/max(norm))*10000,
            f=f, output=output)
```

[41] `synth-tonal-1.wav`, `synth-tonal-2.wav`, `synth-tonal-3.wav`.

[42] `synth-tonal-4.wav`, `synth-tonal-5.wav`.

[43] `synth-tonal-6.wav`.

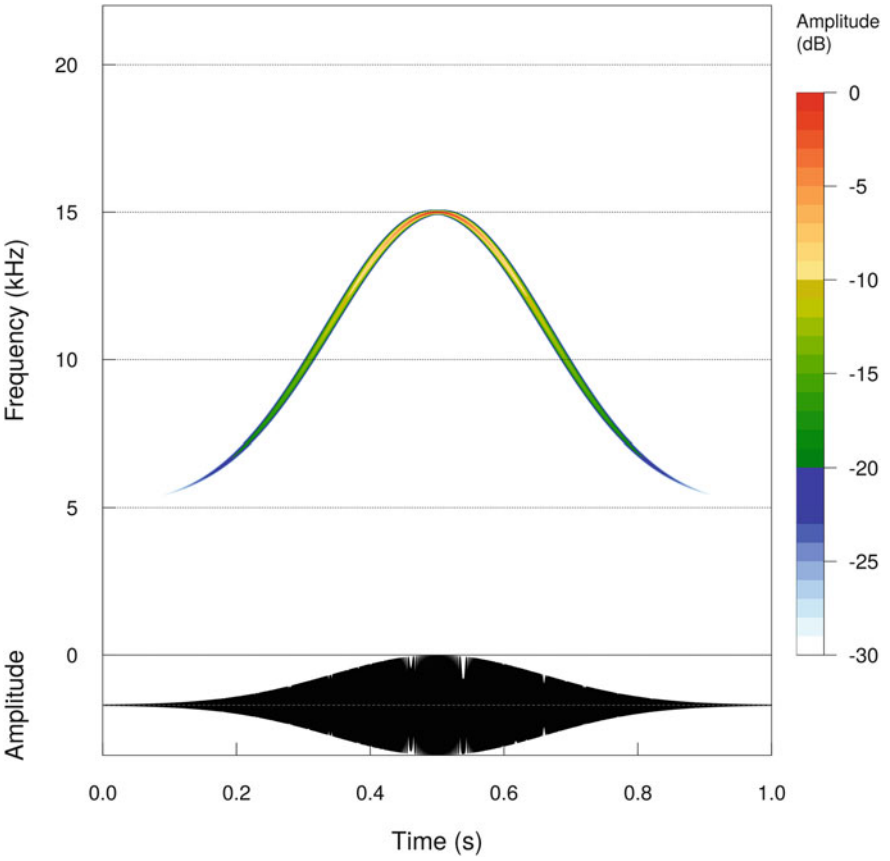


Fig. 18.24 Synthetic sound with AM and FM following a normal density function. The sound was generated using tonal principle with the function `synth2()`. Fourier window size = 1024 samples, overlap = 87.5%, Hanning window

18.6.3 Examples

18.6.3.1 Synthesis of Frequency Bands Based on a Pre-existing Sound

As we went through in Sect. 15.5 with several examples changing `tico`, tonal synthesis can be used to generate a new sound from a pre-existing sound. The following practice consists in generating a sound that has the same properties than the fundamental frequency of `peewit` but containing frequency bands of equal energy. The first step of the process consists in selecting the fundamental frequency of `peewit` by filtering out all other frequencies, that is, by applying a bandpass filter between 1000 and 1500 Hz with `fir()` (see Sect. 14.6). Then the instantaneous frequency and the instantaneous amplitude of the fundamental frequency are retrieved thanks to the functions `ifreq()` and `env()`, respectively:

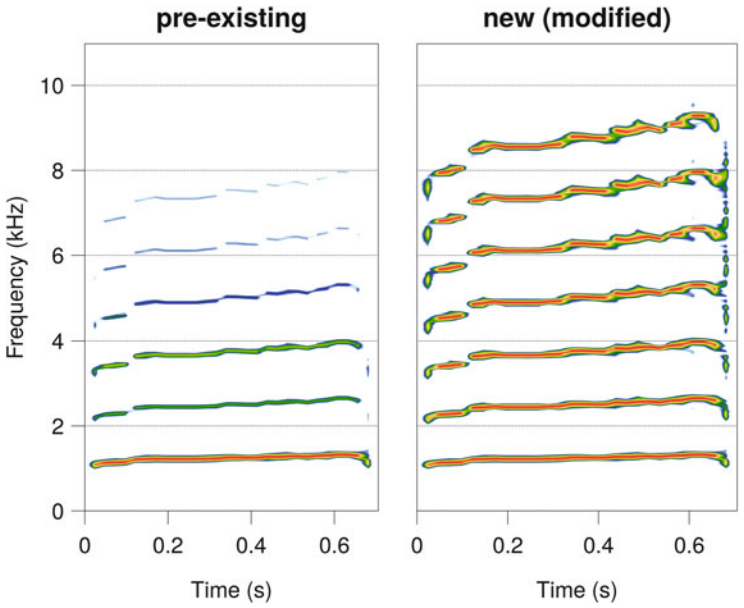


Fig. 18.25 Tonal synthesis based on a pre-existing sound. The pre-existing sound of `peewit` (left) was used to synthesize a new sound (right) with several frequency bands of equal energy. Fourier window size = 512 samples, overlap = 0%, Hanning window

```
data(peewit)
f <- peewit@samp.rate
peewit.filt <- fir(peewit, from=1000, to=1500, output="Wave")
peewit.ifr <- ifreq(peewit.filt, plot=FALSE)$f[,2]*1000
peewit.env <- env(peewit.filt, plot=FALSE)
```

We now apply the function `synth2()` each time that we wish to add a frequency band. Here we decide to add six frequency harmonics which relative amplitudes are all similar. The final sound is then obtained with a loop `for` initiated with a `Wave` object containing zeroes only (Fig. 18.25):[44]

```
s <- Wave(numeric(length(peewit.env)), samp.rate=f, bit=16)
for(i in 1:7) s <- s + synth2(peewit.env,
                              i*peewit.ifr, f=f, output=out)
```

[44] `synth-peewit.wav`.

18.6.3.2 Face

We used in Chap. 11 a synthetic wave which spectrographic display reminds a smiling face. This wave was partly built using `synth2()`. The idea was to generate a sound for each face element and to sum up at the end of the process all the elements. The main constraint is to generate individual sounds which all have the same sampling frequency f_s and exactly the same duration, that is the same number of samples, here 1 s.

The smile sound is based on `synth2()`, the amplitude envelope following the density function of a normal distribution and the instantaneous frequency being defined by a square function:

```
f <- 44100 ; d <- 1
# amplitude envelope
env.smile <- dnorm(-(f/2):(f/2 - 1), sd=5000)
# instantaneous frequency
ifreq.smile <- 1500 + seq(-135,135,length.out=d*f)^2
smile <- synth2(env=env.smile, ifreq=ifreq.smile,
                f=f, output="Wave")
```

The eyes are also based on `synth2()` with a density function for the amplitude envelope recycled for the instantaneous frequency:

```
env.eyes <- rep(dnorm(-(f/4):(f/4-1), sd=1000), 2)
ifreq.eyes <- 15000 + (env.eyes/max(env.eyes))*1000
eyes <- synth2(env=env.eyes,ifreq=ifreq.eyes, f=f, output="Wave")
```

The nose is a pure tone sound without any AM or FM generated with `synth()`. Silence are added before and after the pure tone to reach 1 s sound:

```
nose <- synth(f=44100, d=0.2, cf=10000, output="Wave")
nose <- addsilw(nose, at="start", d=0.4, output="Wave")
nose <- addsilw(nose, at="end", d=0.4, output="Wave")
```

The hair are made of two pure tones affected by a sinusoid FM and with a triangular amplitude envelope. This is obtained by summing the results of two `synth()` calls:

```
hair <- synth(f=f,d=d, cf=20000, fm=c(250,10,0,0,0),
              shape="tria", output="Wave")
hair <- hair + 0.8*synth(f=f,d=d, cf=21000, fm=c(250,10,0,0,0),
              shape="tria", output="Wave")
```

The last operation consists in normalizing in amplitude each element so that they have similar amplitude ranges and in summing up all the elements (Fig. 18.26)[45]:

```
face <- normalize(smile) + normalize(eyes) +
        normalize(nose) + normalize(hair)
```

```
spectro(face, ovlp=75)
```

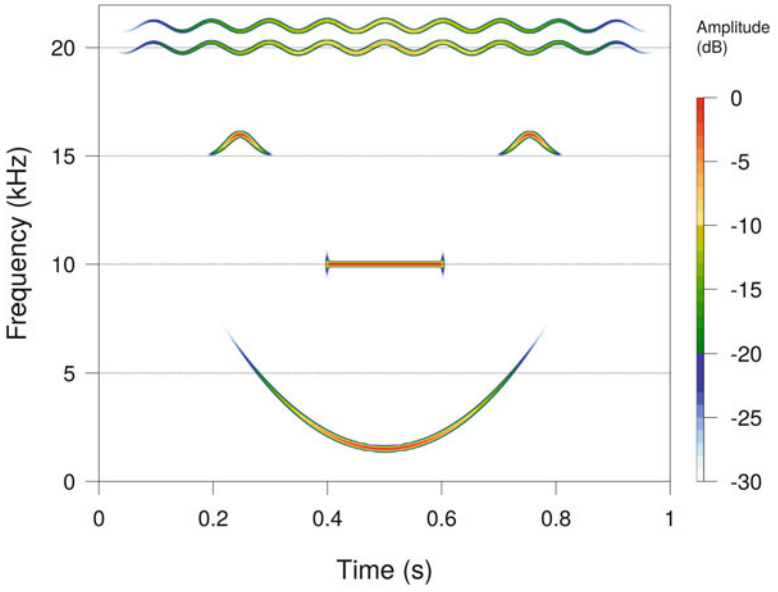


Fig. 18.26 Synthesis of a face-like sound. This smiling face was synthesized using additive synthesis with `synth()` and tonal synthesis `synth2()`. Fourier window size = 512 samples, overlap = 75%, Hanning window

[45] `synth-face.wav`.

18.7 Speech

There are two main ways to synthesize speech ex nihilo[46]: (1) articulatory synthesis that attempts to follow the mechanical model of speech production also known as the source-filter model (articulatory source + resonant tracts) and (2) formant synthesis which works as an additive synthesis method that mainly consists in adding a fundamental frequency and a series of formants. The formant synthesis is available thanks to the function `vowelsynth()` of the package `phonTools` and the articulatory synthesis with the function `soundgen()` of the package `soundgen`.

18.7.1 Solution with the Package `phonTools`

The function `vowelsynth()`, which follows the method defined by Klatt (1980), can synthesize a vowel based on the following main arguments:

`ffs` the center frequencies for each formant in Hz. A list containing two vectors may be provided to indicate the initial and final values so that a linear FM can be applied.
`fbw` the bandwidths for each formant in Hz.
`dur` the duration in ms, by default 300 ms.
`f0` the fundamental frequency in Hz. A vector with initial and final values may also be provided for a linear FM.
`fs` the sampling frequency f_s in Hz.
`power` the amplitude envelope.

Noise can be added to make the output less robotic with the following two arguments:

`noise1` the level of the noise that would affect the speech source, that is before the filtering action of the formants.
`noise2` the level of the noise that would be added to the output, that is, once the sound has been produced by the speaker.

[46]Concatenative synthesis or diphone synthesis, which consists in concatenating diphones, is not strictly speaking a synthesis method as it relies on real speech recordings. A diphone is a unit of speech that makes the transition between two simple speech sounds known as phones.

We first call the library:

```
library(phonTools)
```

The following example, extracted from the documentation of `vowelsynth()`, consists in synthesizing the five vowels of an English speaker. We use `returnsound=FALSE` to retrieve a numeric vector instead of a `sound` object:[47]

```
f0 <- c(125, 105)
i <- vowelsynth(f0=f0, returnsound=FALSE)
a <- vowelsynth(ffs=c(700, 1300, 2300, 3400, 4400),
                f0=f0, returnsound=FALSE)
e <- vowelsynth(ffs=c(400, 2000, 2600, 3400, 4400),
                f0=f0, returnsound=FALSE)
o <- vowelsynth(ffs=c(400, 900, 2300, 3400, 4400),
                f0=f0, returnsound=FALSE)
u <- vowelsynth(ffs=c(300, 750, 2300, 3400, 4400),
                f0=f0, returnsound=FALSE)
silence <- rep(0, 1000)  # 100 ms of silence
s <- c(silence, a, silence, e,
       silence, i, silence, o,
       silence, u, silence)
```

The results can be viewed with (Fig. 18.27):

```
spectro(s, f=10000, ovlp=87.5, collevels=seq(-60,0,1))
text(x=(0.1+0.3)*0:4 + (0.1+0.15), y=2.5,
     labels=c("[i]", "[a]", "[e]", "[o]", "[u]"),
     col="red",
     cex=3)
```

18.7.2 Solution with the Package `soundgen`

The function `soundgen()` of the eponymous package can synthesize a great variety of sounds including human nonlinguistic and animal vocalizations. Based on the

[47] `synth-vowels-phontools.wav`.

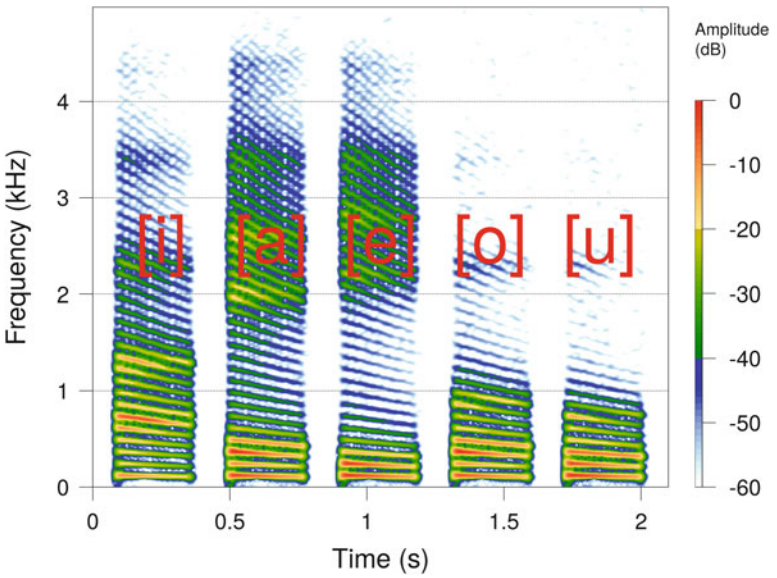


Fig. 18.27 Synthesis of an English speaker vowels with `phonTools`. The five vowels were synthesized with `vowelsynth()` of the package `phonTools`. Fourier window size = 512 samples, overlap = 87.5%, Hanning window, dynamic range = 60 dB

source-filter model, the function generates a fundamental frequency accompanied with a harmonic series for the articulatory source, a coloring noise for breathing sound, and a frequency filter for the resonant tract and the lip radiation. The function can produce several syllables with a control over the contour of the fundamental frequency f_0, the contour of the amplitude that is the amplitude envelope, and the amplitude profiles of the formants. Other parameters control for a random variation between successive syllables (the so-called temperature), the gender of the voice (male *vs* female), nonlinear effects (subharmonics, jitter, and shimmer), and background noise (e.g., breathing, hissing). This versatility in the setting parameters leads to the possibility to synthesize vowel, scream, roar, moan, sigh, gasp, laugh, cry, and Mmm sounds as well as mammal sounds as cat meowing, cow bellowing, dog barking, elephant trumpeting, chimpanzee vocalizations, or bird calls.

The most easiest way to use `soundgen()` is to take advantage of presets that correspond to vowels. Here follows the code used to generate a sequence of the five vowels uttered by a male.[48] The vowels are literally written in the argument `formants`, the corresponding number of syllables is given in `nSyl`, the syllable duration is set with `sylLen`, the silence duration between the syllables is provided in ms with `pauseDur_mean`, and the sampling frequency f_s is set with `samplingRate`:

[48] `synth-vowels-soundgen.wav`.

```
f <- 10000
s <- soundgen(formants='iaeou', nSyl=5, sylLen=300,
                  pauseDur_mean=100, samplingRate=f)
```

The function returns a vector that can be coerced into a `Wave` object

```
s <- Wave(s, samp.rate=f, bit=16)
```

and visualized with `spectro()` (Fig. 18.28):

```
spectro(s, ovlp=87.5, collevels=seq(-60,0,1))
text(x=c(0.2,0.75,1.25,1.75,2.25), y=4.5,
     labels=c("[i]", "[a]", "[e]", "[o]", "[u]"),
     col="red",
     cex=3)
```

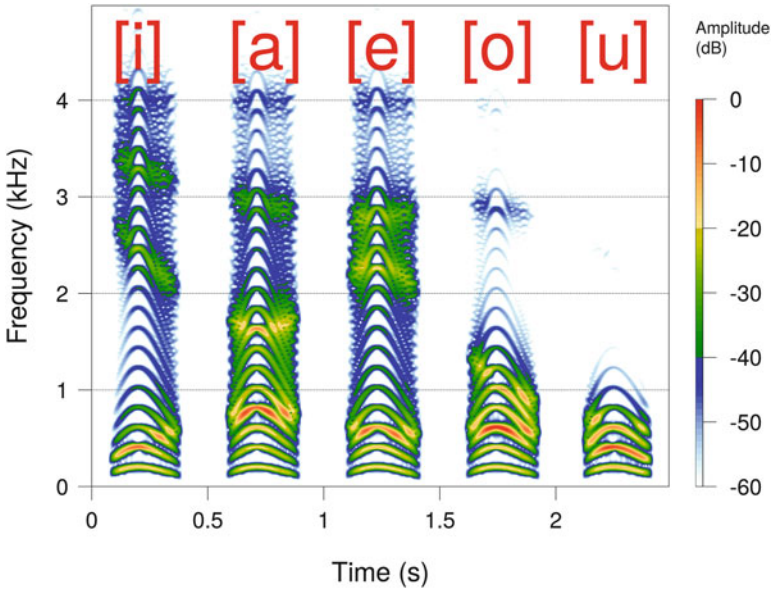


Fig. 18.28 Synthesis of an English speaker vowels with `soundgen`. The five vowels were synthesized with `generateBout()`. Fourier window size = 512 samples, overlap = 87.5%, Hanning window, dynamic range = 60 dB

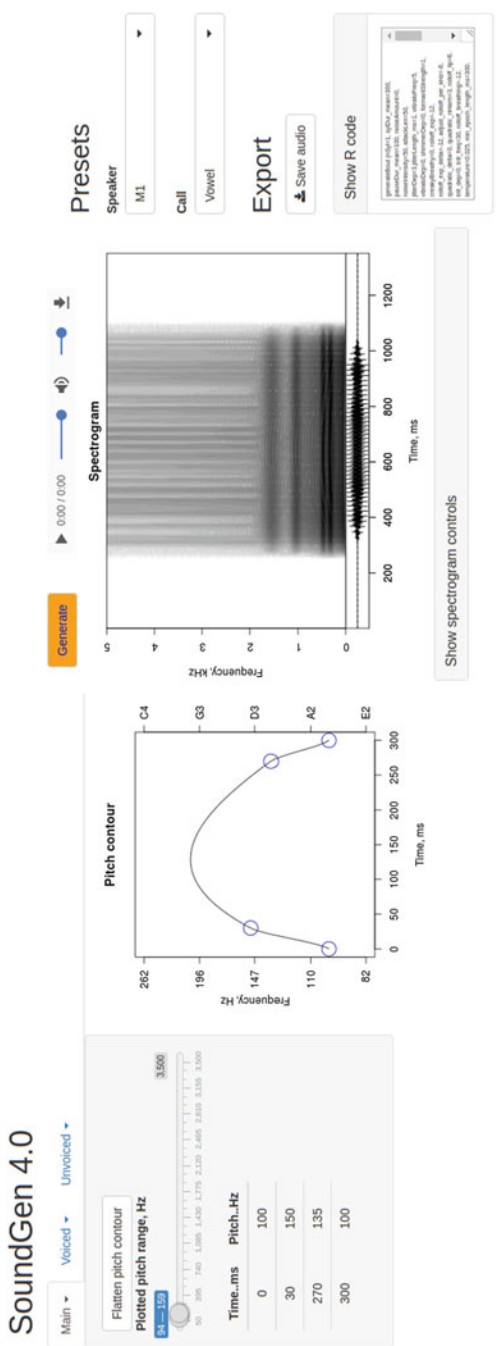


Fig. 18.29 `soundgen` Shiny application. A web Shiny application linked to the package `soundgen`

More complex examples can be tested with the vignette "Sound generation with `soundgen`":

```
vignette("acoustic_generation", package="soundgen")
```

For those who are unfortunately reluctant to the use of the `R` console, a Shiny application (Chang et al. 2016) has also been developed for an interactive use of `soundgen()`. The synthesis parameters and the sound can be saved in `.csv` and `.wav` format, respectively. The shiny application can be launched from the `R` console with (Fig. 18.29):

```
soundgen_app()
```

Appendix A
List of R Functions

List of audio 0.1.5 (Urbanek 2013), monitoR 1.0.5 (Katz et al. 2016b), phonTools 0.2.2.1 (Barreda 2015), seewave 2.1.0 (Sueur et al. 2008a), signal 0.7.6 (Ligges et al. 2015), soundecology 1.3.2 (Villanueva-Rivera and Pijanowski 2016), soundgen 1.1.0 (Anikin 2017), tuneR 1.3.1 (Ligges et al. 2014), and warbleR 1.1.5 (Araya-Salas and Smith-Vidaurre 2016) functions used in this book and grouped by themes. Some functions can appear in several themes. The description comes from the description field of the documentation as provided by the package authors. A complete definition of each function is included in the related package documentation. The number of the main chapter where each function is introduced is given in the last column of the table.

Name	Package	Description	Chapter
Input/output			
audio.drivers()	audio	Lists all currently loaded and available audio drivers	4
checkwavs()	warbleR	Checks .wav files	4
current.audio.driver()	audio	Returns the name of the currently active audio driver	4
getWavPlayer()	tuneR	Gets the default player for .wav files	4
loadsound()	phonTools	Allows .wav files to be loaded into R	4
load.wave()	audio	Loads a .wav file	4
load.audio.driver()	audio	Attempts to load a modular audio driver	4
listen()	seewave	Plays a sound wave	4
mp32wav()	warbleR	Converts .mp3 files to .wav	4
play()	audio	Plays audioSample objects	4

(continued)

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7

Name	Package	Description	Chapter
play()	tuneR	Plays Wave objects	4
playlist()	seewave	Runs a playlist of sound files	4
playsound()	phonTools	Plays sounds in R using VLC player	4
pause()	audio	Pauses (stops) audio recording or playback	4
querxc()	warbleR	Accesses Xeno-Canto recordings and metadata	4
readMP3()	tuneR	Reads an MPEG-2 layer 3 file into a Wave object	4
readWave()	tuneR	Reads and writes .wav files	4
record()	audio	Records audio	4
resume()	audio	Resumes previously paused audio recording or playback	4
rewind()	audio	Rewinds audio recording or playback	4
savewav()	seewave	Saves audio data as .wav file	4
save.wave()	audio	Saves audioSample objects as .wav file	4
set.audio.driver()	audio	Selects an audio driver as the current driver	4
setWavPlayer()	tuneR	Sets the default player for .wav files	4
sox()	seewave	Calls externally SoX	3
wait()	audio	Waits for an event during a recording session	4
wav2flac()	seewave	wav-flac file conversion	4
writesound()	phonTools	Creates a WAV file from a numeric vector or sound object	4
writeWave()	tuneR	Reads and writes Wave files	4
xcmaps()	warbleR	Maps Xeno-Canto recordings by species	4
Objects			
as.audioSample()	audio	Converts an object into an audio sample object	4
audioSample()	audio	audiosample class and constructor	4
duration()	seewave	Duration of a time wave	4
equalWave()	tuneR	Checks for some kind of equality of objects of class Wave	4
makesound()	phonTools	Creates a sound object from a numeric vector.	4
MCnames()	tuneR	Defaults channel ordering for multichannel wave files (WaveMC objects)	6
nchannel()	tuneR	Number of channels	6
updateWave()	tuneR	Updates old Wave objects for use with new versions of tuneR	4
Wave()	tuneR	Constructors and coercion for class Wave objects	4
WaveMC()	tuneR	Constructors and coercion for class WaveMC objects	4

(continued)

Name	Package	Description	Chapter
Edition			
addsilw()	seewave	Adds or inserts a silence section	6
bind()	tuneR	Concatenates Wave objects	6
channel()	tuneR	Channel conversion for Wave objects	6
crossFade()	soundgen	Joins two waveforms by cross-fading	6
cutw()	seewave	Cuts a section of a time wave	6
deletew()	seewave	Deletes a section of a time wave	6
downsample()	tuneR	Downsamples a Wave or WaveMC object	6
equalWave()	tuneR	Checks Wave objects	4
extractWave()	tuneR	Extractor for Wave objects	6
mono()	tuneR	Converts stereo to mono and vice versa	6
mutew()	seewave	Replaces time wave data by 0 values	6
normalize()	tuneR	Rescales the range of values	6
noSilence()	tuneR	Cuts off silence from a Wave object	6
panorama()	tuneR	Narrows the panorama of a stereo sample	6
pastew()	seewave	Pastes a time wave to another one	6
prepComb()	tuneR	Prepares the combination/concatenation of Wave objects	6
repw()	seewave	Repeats a time wave	6
resamp()	seewave	Resamples a time wave	6
rmoffset()	seewave	Removes the offset of a time wave	6
stereo()	tuneR	Converts (extracts, joins) stereo to mono and vice versa	6
zapsilw()	seewave	Zaps silence periods of a time wave	6
Time/amplitude			
acoustat()	seewave	Statistics on time and frequency STFT contours	11
ama()	seewave	Amplitude modulation analysis of a time wave	8
corenv()	seewave	Cross-correlation between two time wave envelopes	17
crest()	seewave	Crest factor	7
discrets()	seewave	Discretization of a numeric (time) series	10
drawenv()	seewave	Draws the amplitude envelope of a time wave	15
dynoscillo()	seewave	Dynamic oscillogram	5
env()	seewave	Amplitude envelope of a time wave	5
oscillo()	seewave	Shows a time wave as an oscillogram	5
oscilloST()	seewave	Shows a stereo time wave as oscillograms	6
phaseplot()	seewave	First, second, and third derivatives of a wave	10
phaseplot2()	seewave	Phase portrait of a wave	10
powertrack()	seewave	Creates a power track for a sound	5
roughness()	seewave	Roughness of a curve (a time wave or a spectrum)	10
rugo()	seewave	Rugosity of a time wave or time series	10
segment()	soundgen	Segments a sound	8
setenv()	seewave	Sets the amplitude envelope of a time wave to another one	15

(continued)

Name	Package	Description	Chapter
`timer()`	`seewave`	Time measurements of a time wave	8
`th()`	`seewave`	Temporal entropy	16
Frequency			
`bark2hz()`	`tuneR`	Frequency scale conversion	9
`bwfilter()`	`seewave`	Butterworth frequency filter	14
`ceps()`	`seewave`	Cepstrum or real cepstrum	10
`coh()`	`seewave`	Coherence between two time waves	17
`comb filter()`	`seewave`	Combfilter	14
`corspec()`	`seewave`	Cross-correlation between two frequency spectra	17
`cutspec()`	`seewave`	Cuts a frequency spectrum	10
`diffcumspec()`	`seewave`	Difference between two cumulative frequency spectra	16
`diffspec()`	`seewave`	Difference between two frequency spectra	16
`drawfilter()`	`seewave`	Draws the frequency response of a filter	14
`fbands()`	`seewave`	Frequency band plot (equalizer plot)	10
`FF()`	`tuneR`	Estimation of fundamental frequencies from a Wspec object	13
`FFpure()`	`tuneR`	Estimation of fundamental frequencies from a Wspec object	13
`ffilter()`	`seewave`	Frequency filter	14
`fma()`	`seewave`	Frequency modulation analysis	13
`fpeaks()`	`seewave`	Frequency peak detection	10
`ftwindow()`	`seewave`	Fourier transform windows	9
`fund()`	`seewave`	Fundamental frequency	10
`H()`	`seewave`	Total entropy	16
`hz2bark()`	`tuneR`	Frequency scale conversion	9
`hz2mel()`	`tuneR`	Frequency scale conversion	9
`HzToSemitones()`	`soundgen`	Convert Hz to semitones	9
`ifreq()`	`seewave`	Instantaneous frequency	13
`itakura.dist()`	`seewave`	Itakura-Saito distance	16
`kl.dist()`	`seewave`	Kullback-Leibler distance	16
`ks.dist()`	`seewave`	Kolmogorov-Smirnov distance	16
`lifter()`	`tuneR`	Liftering of cepstra	12
`localpeaks()`	`seewave`	Local maximum frequency peak detection	10
`logspec.dist()`	`seewave`	Log-spectral distance	16
`meanspec()`	`seewave`	Mean frequency spectrum of a time wave	11
`mel()`	`seewave`	Hertz/Mel conversion	9
`mel2hz()`	`tuneR`	Frequency scale conversion	9
`melfcc()`	`tuneR`	MFCC calculation	12
`melfilterbank()`	`seewave`	Mel-frequency filter bank	12
`notesDict()`	`soundgen`	Conversion table from Hz to semitones above C0 to musical notation	9
`notefreq()`	`seewave`	Frequency of a musical note	9

(continued)

Name	Package	Description	Chapter
noteFromFF()	tuneR	Derives notes from frequencies	10
periodogram()	tuneR	Periodogram (spectral density) estimation on Wave objects	10
preemphasis()	seewave	Preemphasis frequency filter	14
Q()	seewave	Resonance quality factor of a frequency spectrum	10
roughness()	seewave	Roughness of a curve (a time wave or a spectrum)	10
SAX()	seewave	Symbolic aggregate approximation	10
sfm()	seewave	Spectral flatness measure	10
sh()	seewave	Spectral entropy	10
semitonesToHz()	soundgen	Convert semitones to Hz	9
simspec()	seewave	Similarity between two frequency spectra	16
soundscapespec()	seewave	Soundscape frequency spectrum of a time wave	11
spec2cep()	tuneR	Spectra to cepstra conversion	12
spec()	seewave	Frequency spectrum of a time wave	10
specan()	warbleR	Measures acoustic parameters in batches of sound files	10
specprop()	seewave	Spectral properties	10
squarefilter()	seewave	Frequency response of a square filter	14
symba()	seewave	Symbol analysis	10
zcr()	seewave	Zero-crossing rate	13
Time/frequency			
acoustat()	seewave	Statistics on time and frequency STFT contours	11
analyze()	soundgen	Analyzes sound	13
audspec()	tuneR	Frequency band conversion	12
autoc()	seewave	Short-time autocorrelation of a time wave	13
batchBinMatch	monitoR	Batch template detection	17
batchCorMatch	monitoR	Batch template detection	17
binMatch	monitoR	Calculates spectrogram template matching scores	17
ccoh()	seewave	Continuous coherence function between two time waves	17
combineBinTemplates	monitoR	Combines acoustic template lists	17
combineCorTemplates	monitoR	Combines acoustic template lists	17
corMatch	monitoR	Calculates spectrogram template matching scores	17
covspectro()	seewave	Covariance between two spectrograms	17
deltas()	tuneR	Calculates delta MFCC features	12
dfDTW()	warbleR	Dynamic time warping on dominant frequency contours	17
dfreq()	seewave	Dominant frequency of a time wave	13

(continued)

Name	Package	Description	Chapter
dynspec()	seewave	Dynamic sliding spectrum	11
dynspectro()	seewave	Dynamic sliding spectrogram	11
eventEval	monitoR	Evaluates detected events with known event sources and times	17
ffDTW()	warbleR	Dynamic time warping on fundamental frequency contours	17
ffilter()	seewave	Frequency filter	14
findformants()	phonTools	Finds formants given a sound or set of LPC coefficients	12
findPeaks()	monitoR	Finds score peaks and detections in a templateScores object	17
formanttrack()	phonTools	Creates a formant track for a sound	13
ftwindow()	seewave	Fourier transform windows	9
fund()	seewave	Fundamental frequency	10
getPeaks()	monitoR	Extracts detections or peaks from a detectionList object	17
hilbert()	seewave	Hilbert transform and analytic signal	13
ifreq()	seewave	Instantaneous frequency	13
istft()	seewave	Inverse of the short-time Fourier transform	11
lpc()	phonTools	Predicts autoregressive filter coefficients	12
lspec()	warbleR	Creates long spectrograms of whole sound files	11
makeBinTemplate()	monitoR	Makes an acoustic template	17
makeCorTemplate()	monitoR	Makes an acoustic template	17
manualoc()	warbleR	Interactive view of spectrograms	11
manualoc.df()	warbleR	Data frame of manualoc() selections	11
meanspec()	seewave	Mean frequency spectrum of a time wave	11
melodyplot()	tuneR	Plots a melody	13
powspec()	tuneR	Power spectrum	11
periodogram()	tuneR	Periodogram (spectral density) estimation on Wave objects	11
pitchtrack()	phonTools	Creates a pitch (fundamental frequency) track for a sound	13
readBinTemplates()	monitoR	Reads acoustic templates from a local disk	17
readCorTemplates()	monitoR	Reads acoustic templates from a local disk	17
showPeaks()	monitoR	Views or verifies detections or peaks	17
spectro()	seewave	2D spectrogram of a time wave	11
spectrogram()	phonTools	Creates and displays spectrograms	11
spectrogram()	soundgen	Spectrogram	11
spectro3D()	seewave	3D-spectrogram of a time wave	11
stft.ext()	seewave	Short-time Fourier transform using external C libraries	11

(continued)

Name	Package	Description	Chapter
templateCutoff()	monitoR	Queries or sets template cutoffs	17
timeAlign()	monitoR	Condenses detections or peaks from multiple templates	17
viewSpec()	monitoR	Interactively views and annotates spectrograms	11
wf()	seewave	Waterfall display	11
TKEO()	seewave	Teager-Kaiser energy operator	13
zc()	seewave	Instantaneous frequency of a time wave by zero crossing	13
Indices			
ACI()	seewave	Acoustic complexity index	16
acoustic_complexity()	soundecology	Acoustic complexity index	16
acoustic_diversity()	soundecology	Acoustic diversity index	16
acoustic_evenness()	soundecology	Acoustic evenness index	16
AR()	seewave	Acoustic richness index	16
bioacoustic_index()	soundecology	Bioacoustic index	16
diffcumspec()	seewave	Difference between two cumulative frequency spectra	16
diffspec()	seewave	Difference between two frequency spectra	16
fpeaks()	seewave	Frequency peaks detection	10
H()	seewave	Total entropy	16
itakura.dist()	seewave	Itakura-Saito distance	16
kl.dist()	seewave	Kullback-Leibler distance	16
ks.dist()	seewave	Kolmogorov-Smirnov distance	16
logspec.dist()	seewave	Log-spectral distance	16
M()	seewave	Amplitude index	16
multiple_sounds()	soundecology	Multiple sound files	16
ndsi()	soundecology	Normalized difference soundscape index	16
NDSI()	seewave	Normalized difference soundscape index	16
roughness()	seewave	Roughness of a curve (a time wave or a spectrum)	10
SAX()	seewave	Symbolic aggregate approximation	10
sfm()	seewave	Spectral flatness measure	10
sh()	seewave	Spectral entropy	16
simspec()	seewave	Similarity between two frequency spectra	16
th()	seewave	Temporal entropy	16
Filters and modifications			
afilter()	seewave	Amplitude filter	15
bwfilter()	seewave	Butterworth frequency filter	14
echo()	seewave	Echo generator	15

(continued)

Name	Package	Description	Chapter
fadew()	seewave	Fade in and fade out of a time wave	6
fir()	seewave	Finite impulse response filter	14
lfs()	seewave	Linear frequency shift	15
preemphasis()	seewave	Preemphasis frequency filter	14
revw()	seewave	Time reverse of a time wave	6
rmam()	seewave	Removes the amplitude modulations of a time wave	15
rmnoise()	seewave	Removes Gaussian noise	14
rmoffset()	seewave	Removes the offset of a time wave	6
Synthesis			
chirp()	signal	Creates a chirp signal	18
getRolloff()	soundgen	Controls roll-off of harmonics	18
noise()	tuneR	Creates Wave objects of special waveforms	18
noisew()	seewave	Generates noise	18
pulsew()	seewave	Generates rectangle pulse	18
sawtooth()	tuneR	Creates Wave objects of special waveforms	18
silence()	tuneR	Creates Wave objects of special waveforms	18
sine()	tuneR	Creates Wave objects of special waveforms	18
square()	tuneR	Creates Wave objects of special waveforms	18
soundgen()	soundgen	Generates a sound	18
soundgen_app()	soundgen	soundgen shiny app	18
synth()	seewave	Synthesis of time wave (additive model)	18
synth2()	seewave	Synthesis of time wave (tonal model)	18
vowelsynth()	phonTools	Creates synthetic vowels using a cascade formant synthesizer	18
Extra			
attenuation()	seewave	Generates sound intensity attenuation data	7
convSPL()	seewave	Converts sound pressure level in other units	7
dBscale()	seewave	dB color scale for a spectrogram display	11
dBweight()	seewave	dB weightings	7
meandB()	seewave	Mean of dB values	7
micsens()	seewave	Microphone sensitivity	7
moredB()	seewave	Addition of dB values	7
notenames()	tuneR	Generates note names from numbers	9
octaves()	seewave	Octaves series	9
quantMerge()	tuneR	Quantization of notes to produce sheet music	13
quantize()	tuneR	Quantization of notes to produce sheet music	13
quantplot()	tuneR	Plots the quantization of a melody	13
rms()	seewave	Root mean square	7
songmeter()	seewave	Reads and interprets songmeter file name	4

Appendix B
Sound Samples

Here are information of the sounds used as samples, or examples. Several sounds were very kindly provided by colleagues.

Name: `bat`
Data source: file 'Pipistrellus_kuhlii.wav'
Description: one call of the European bat *Pipistrellus kuhlii*
Author: Jean-François Julien
Location: Cournonterral, France
Properties:

```
Wave Object
Number of Samples:      3841
Duration (seconds):     0.02
Samplingrate (Hertz):   192000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `cockroach`
Data source: file 'Elliptorhina_chopardi.wav'
Description: one male courtship call of the hissing cockroach from Madagascar *Elliptorhina chopardi*
Author: Jérôme Sueur
Location: laboratory, Orsay, France
Properties:

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7

```
Wave Object
Number of Samples:      19137
Duration (seconds):     0.43
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `elephant`
Data source: file 'Loxodonta_africana.wav'
Description: eight sounds used in Gilbert et al. (2014, figures 2–3). Sounds 1–4 were produced produced with a 3 m hose pipe. Sounds 1 and 2 are low-amplitude, "non-brassy" sounds produced with a low-intensity source. Sounds 3 and 4 are high-amplitude, "brassy" sounds produced with a high (increasing)-intensity source. Sounds 5–8 are trumpet calls recorded from a 20-year-old female African elephant. Sounds 5 and 6 are low-amplitude, "non-brassy" trumpet calls. Sounds 7 and 8 are high-amplitude trumpet calls.
Author: Joël Gilbert
Location: Beauval Zoo, Saint-Aignan, France for the elephant trumpet calls
Properties:

```
Wave Object
Number of Samples:      366706
Duration (seconds):     8.32
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `face`
Data source: file 'synth-face.wav'
Description: a synthetic sound made of various items modulated in amplitude and frequency. The sound was created with the code detailed in Sect. 18.6.3.2.
Author: Jérôme Sueur
Location: *in silico*
Properties:

```
Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    32
```

Name: `femo`
Data source: file 'Allobates_femoralis.wav'
Description: four vocalizations of the South-American dart poison frog *Allobates femoralis*
Author: Jérôme Sueur
Location: Kaw, Guiana, France
Properties:

```
Wave Object
Number of Samples:      61740
Duration (seconds):     1.4
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `femo`
Data source: file 'Allobates_femoralis_2015-11-10_161500_GFT.wav'
Description: 28 vocalizations of the South-American dart poison frog *Allobates femoralis*
Author: Jérôme Sueur
Location: Location: Kaw, Guiana, France
Properties:

```
Wave Object
Number of Samples:      1323000
Duration (seconds):     30
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `forest`
Data source: file 'forest.wav'
Description: one minute of soundscape recording in the tropical forest during the first part of the night
Author: Jérôme Sueur
Location: Kaw, Guiana, France
Properties:

```
Wave Object
Number of Samples:      2646000
Duration (seconds):     60
```

(continued)

```
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `frog`
Data source: file 'Eleutherodactylus_martinicensis.wav'
Description: 17 two-note vocalizations of the Martinique Robber frog *Eleutherodactyus martinicenis*
Author: Renaud Boistel
Location: Lesser Antilles, France
Properties:

```
Wave Object
Number of Samples:      316602
Duration (seconds):     19.79
Samplingrate (Hertz):   16000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `guiana`
Data source: files
'M-XV_20101125_000000.wav', 'M-XV_20101125_010000.wav',
'M-XV_20101125_020000.wav' 'M-XV_20101125_030000.wav',
'M-XV_20101125_040000.wav' 'M-XV_20101125_050000.wav',
'M-XV_20101125_060000.wav', 'M-XV_20101125_070000.wav',
'M-XV_20101125_080000.wav', 'M-XV_20101125_090000.wav',
'M-XV_20101125_100000.wav', 'M-XV_20101125_110000.wav',
'M-XV_20101125_120000.wav', 'M-XV_20101125_130000.wav',
'M-XV_20101125_140000.wav', 'M-XV_20101125_150000.wav',
'M-XV_20101125_160000.wav', 'M-XV_20101125_170000.wav',
'M-XV_20101125_180000.wav', 'M-XV_20101125_190000.wav',
'M-XV_20101125_200000.wav' 'M-XV_20101125_210000.wav',
'M-XV_20101125_220000.wav', 'M-XV_20101125_230000.wav'
Description: 24 stereo files recorded in the tropical forest in French Guiana every hour from 00:00 to 23:00 on the 25 November 2010
Author: Amandine Gasc and Jérôme Sueur
Location: Nouragues, Guiana, France
Properties: first file

```
Wave Object
Number of Samples:      2646016
Duration (seconds):     60
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Stereo
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `hello`
Data source: file 'hello.wav'
Description: English word "hello" said by a 7-year old French native girl
Author: Jérôme Sueur
Location: Paris, France
Properties:

```
Wave Object
Number of Samples:      38400
Duration (seconds):     0.8
Samplingrate (Hertz):   48000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `noise`
Data source: file 'noise.wav'
Description: a white noise broadcast by a low-quality loudspeaker and recorded with a high-quality microphone in an anechoic chamber.
Author: Diego Llusia
Location: Paris, France
Properties:

```
Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `peewit`
Data source: `seewave` accompanying data
Description: song emitted by a peewit (lapwing) male *Vanellus vanellus*
Author: Thierry Aubin
Location: France
Properties:

```
Wave Object
Number of Samples:      15561
Duration (seconds):     0.71
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `pellucens`
Data source: `seewave` accompanying data
Description: two stridulations of the Italian tree cricket *Oecanthus pellucens*
Author: Jérôme Sueur
Location: Brain-sur-Allonnes, France
Properties:

```
Wave Object
Number of Samples:      36476
Duration (seconds):     3.31
Samplingrate (Hertz):   11025
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `sheep`
Data source: `seewave` accompanying data
Description: a single bleat of the Préalpes-du-Sud *Ovis aries*
Author: Frédéric Sèbe
Location: Brouessy, France
Properties:

```
Wave Object
Number of Samples:      19764
Duration (seconds):     2.47
Samplingrate (Hertz):   8000
```

(continued)

```
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `tico`
Data source: `seewave` accompanying data
Description: four notes of the rufous-collared sparrow *Zonotrichia capensis*
Author: Thierry Aubin
Location: Brazil
Properties:

```
Wave Object
Number of Samples:      39578
Duration (seconds):     1.79
Samplingrate (Hertz):   22050
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `theremin`
Data source: recording freely accessible at
https://www.freesound.org/people/realtheremin/sounds/119007/
Description: a frequency-modulated sound produced by a theremin, an electronic instrument
Author: –
Location: –
Properties:

```
Wave Object
Number of Samples:      626176
Duration (seconds):     14.2
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `toad`
Data source: file 'Alytes_obstetricans.wav'
Description: four single-note vocalizations of the European midwife toad *Alytes*

obstetricans with important background noise due to wind and nocturnal insects (Orthoptera)
Author: Jérôme Sueur
Location: Badefols-sur-Dordogne, France
Properties:

```
Wave Object
Number of Samples:      264000
Duration (seconds):     5.5
Samplingrate (Hertz):   48000
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

Name: `tuningfork`
Data source: file 'tuning-fork.wav'
Description: a synthetic 440 Hz pure tone mimicking a A-tone tuning fork. The sound was created with `synth(f=44100, d=1, cf=440, output="Wave")` Author: Jérôme Sueur
Location: *in silico*
Properties:

```
Wave Object
Number of Samples:      44100
Duration (seconds):     1
Samplingrate (Hertz):   44100
Channels (Mono/Stereo): Mono
PCM (integer format):   TRUE
Bit (8/16/24/32/64):    16
```

References

Adler D, Murdoch D (2016) rgl: 3D visualization device system (OpenGL). http://CRAN.R-project.org/package=rgl, R package

Aldersley A, Champneys A, Homer M, Robert D (2016) Quantitative analysis of harmonic convergence in mosquito auditory interactions. J R Soc Interface 13:20151007

Anikin A (2017) soundgen: Parametric voice synthesis. https://CRAN.R-project.org/package=soundgen, R package 1

Araya-Salas M, Smith-Vidaurre G (2016) warbleR: an R package to streamline analysis of animal acoustic signals. Meth Ecol Evol 8:184–191

Barreda S (2015) phonTools: functions for phonetics in R. https://cran.r-project.org/web/packages/phonTools/index.html, R package

Beeman K (1998) Digital signal analysis, editing, and synthesis. Springer, Berlin/Heidelberg, pp 59–103

Bennet-Clark HC (1999) Which Qs to choose: questions of quality in bioacoustics? Bioacoustics 9:351–359

Boelman NT, Asner GP, Hart PJ, Martin RE (2007) Multi-trophic invasion resistance in Hawaii: bioacoustics, field surveys, and airborne remote sensing. Ecol Appl 17:2137–2144

Boersma P (1993) Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. In: Proceedings of the Institute of Phonetic Sciences, Amsterdam, vol 17, pp 97–110

Bogert BP, Healy MJR, Tukey JW (1963) The quefrency alanysis of time series for echoes: cepstrum, pseudo-autocovariance, cross-cepstrum, and saphe cracking. In: Time series analysis. Wiley, New York, pp 209–243

Borcard D, Gillet F, Legendre P (2011) Numerical ecology with R. Springer, New York

Bradbury JW, Vehrencamp SL (1998) Principles of animal communication. Sinauer Associates, Sunderland

Butterworth S (1930) On the theory of filter amplifiers. Exp Wirel Wirel Eng 7:536–541

Butts CT (2008) network: a package for managing relational data in r. J Stat Softw 24(2):2–36

Carson JR (1922) Notes on the theory of modulation. Proc Institute Radio Eng 10:57–64

Cazelles B (2004) Symbolic dynamics for identifying similarity between rhythms of ecological time series. Ecol Lett 7:755–763

Chang W, Cheng J, Allaire J, Xie Y, McPherson J (2016) shiny: web application framework for R. https://CRAN.R-project.org/package=shiny, R package

Chowning JM (1973) The synthesis of complex audio spectra by means of frequency modulation. J Audio Eng Soc 21:526–531

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7

Cooley JW, Tukey JW (1965) An algorithm for the machine computation of complex Fourier series. Math Comput 19:297–301
Cooper MA, Dultsev FN, Minson T, Ostanin VP, Abell C, Klenerman D (2001) Direct and sensitive detection of a human virus by rupture event scanning. Nat Biotechnol 19:833–837
Cryer JD, Chan KS (2008) Time series analysis with applications in R. Springer, New York
Das A (2012) Signal conditioning. An introduction to continuous wave communication and signal processing. Springer, Berlin
Davis SB, Mermelstein P (1980) Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. IEEE Trans Acoust Speech Signal Process 28:357–366
Deecke VB, Janik VM (2006) Automated categorization of bioacoustic signals: avoiding perceptual pitfalls. J Acoust Soc Am 119:645–653
Depraetere M, Pavoine S, Jiguet F, Gasc A, Duvail S, Sueur J (2012) Monitoring animal diversity using acoustic indices: implementation in a temperate woodland. Ecol Indic 13:46–54
Deutsch D (2010) The paradox of pitch circularity. Acoust Today 6:8–15
Dziak RP, Bohnenstiehl DR, Baker ET, Matsumoto H, Caplan-Auerbach J, Embley RW, Merle SG, Walker SL, Lau TK, Chadwick WW (2015) Long-term explosive degassing and debris flow activity at west mata submarine volcano. Geophys Res Lett 42:1480–1487
Fawcett T (2006) An introduction to ROC analysis. Pattern Recogn Lett 27:861–874
Felisberto P, Jesus SM, Zabel F, Santos R, Silva J, Gobert S, Beer S, Björk M, Mazzuca S, Procaccini G, Runcie JW, Champenois W, Borges AV (2015) Acoustic monitoring of O2 production of a seagrass meadow. J Exp Mar Biol Ecol 464:75–87
Fisher RA (1936) The use of multiple measurements in taxonomic problems. Ann Eugen 7:179–188
Fitch W, Neubauer J, Herzel H (2002) Calls out of chaos: the adaptive significance of nonlinear phenomena in mammalian vocal production. Anim Behav 63:407–418
Fletcher NH (1992) Acoustic systems in biology. Oxford University Press, Oxford
Fristrup KM, Watkins WA (1992) Characterizing acoustic features of marine animal sounds. Tech. rep., Woods Hole Oceanographic Institution Technical Report WHOI-92-04
Fuller S, Axel AC, Tucker D, Gage SH (2015) Connecting soundscape to landscape: which acoustic index best describes landscape configuration? Ecol Indic 58:207–215
Gagliano M, Mancuso S, Robert D (2012) Towards understanding plant bioacoustics. Trends Plant Sci 17:323–325
Gasc A, Sueur J, Jiguet F, Devictor V, Grandcolas P, Burrow C, Depraetere M, Pavoine S (2013a) Assessing biodiversity with sound: do acoustic diversity indices reflect phylogenetic and functional diversities of bird communities? Ecol Indic 25:279–287
Gasc A, Sueur J, Pavoine S, Pellens R, Grandcolas P (2013b) Biodiversity sampling using a global acoustic approach: contrasting sites with microendemics in new caledonia. PLoS ONE 8:e65311
Gasc A, Pavoine S, Lellouch L, Grandcolas P, Sueur J (2015) Acoustic indices for biodiversity assessments: analyses of bias based on simulated bird assemblages and recommendations for field surveys. Biol Conserv 191:306–312
Gençay R, Selçuk F, Whitcher B (2001) An introduction to wavelets and other filtering methods in finance and economics. Academic Press, San Diego
Gilbert J, Dalmont JP, Potier R, Reby D (2014) Is nonlinear propagation responsible for the brassiness of elephant trumpet calls? Acta Acustica United Acustica 100:734–738
Giorgino T (2009) Computing and visualizing dynamic time warping alignments in R: the dtw package. J Stat Softw 31:1–24
Gustafsson MV, Aref T, Kockum AF, Ekström MK, Johansson G, Delsing P (2014) Propagating phonons coupled to an artificial atom. Science 346:207–211
Hartmann WM (1998) Signals, sound, and sensation. Springer, New York
Hermann T, Hunt A, Neuhoff JG (eds) (2011) The sonification handbook. Logos Verlag, Berlin
Hopp SL, Owren MJ, Evans CS (1998) Animal acoustic communication. Springer, Berlin/Heidelberg

Ihaka R (2010) R: lessons learned, directions for the future. In: Joint statistical meetings proceedings
Ihaka R, Gentleman R (1996) R: A language for data analysis and graphics. J Comput Graph Stat 5:299–314
Josse J, Pagès J, Husson F (2008) Testing the significance of the RV coefficient. Comput Stat Data Anal 53:82–91
Kabacoff R (2013) R in action, data analysis and graphics with R. Manning Publications, New York
Kaiser JF (1990) On a simple algorithm to calculate the "energy" of a signal. In: International conference on acoustics, speech, and signal processing, ICASSP-90, pp 381–384
Kantz H, Schreiber T (2003) Non linear time series analysis. Cambridge University Press, Cambridge
Kasten EP, Gage SH, Fox J, Joo W (2012) The remote environmental assessment laboratory's acoustic library: an archive for studying soundscape ecology. Eco Inform 12:50–67
Katz J, Hafner SD, Donovan T (2016a) Assessment of error rates in acoustic monitoring with the R package monitor. Bioacoustics 25:177–196
Katz J, Hafner SD, Donovan T (2016b) Tools for automated acoustic monitoring within the R package monitor. Bioacoustics 25:191–210
Kendrick P, Lopez L, Waddington D, Young R (2016) Assessing the robust of soundscape complexity indices. In: 23rd international congress on sound and vibration, Athens, pp 1–8
Klatt DH (1980) Software for a cascade/parallel formant synthesizer. J Acoust Soc Am 67:971–995
Kvedalen E (2003) Signal processing using the Teager energy operator and other nonlinear operators. Master's thesis, Department of Informatics, University of Oslo
Larsen NJ, Wahlberg M (2017) Sound and sound sources. In: Comparative bioacoustics: an overview. Bentham Science, Sharjah, pp 3–61
Lauterborn W, Parlitz U (1988) Methods of chaos physics and their application to acoustics. J Acoust Soc Am 84:1975–1993
Legendre P, Anderson MJ (1999) Distance-based redundancy analysis: testing multispecies responses in multifactorial ecological experiments. Ecol Monogr 69:1–24
Lellouch L, Pavoine S, Jiguet F, Glotin H, Sueur J (2014) Monitoring temporal change of bird communities with dissimilarity acoustic indices. Methods Ecol Evol 5:495–505
Lemon RE (1971) Vocal communication by the frog *Eleutherodactylus martinicensis*. Copeia 49:211–217
Ligges U, Krey S, Mersmann O, Schnackenberg S (2014) tuneR: Analysis of music. http://r-forge.r-project.org/projects/tuner, R package
Ligges U, Short T, Kienzle P (2015) signal: signal processing. http://CRAN.R-project.org/package=signal, R package
Lin J, Keogh E, Lonardi S, Chiu B (2003) A symbolic representation of time series, with implications for streaming algorithms. In: Proceedings of the 8th ACM SIGMOD workshop on research issues in data mining and knowledge discovery, DMKD '03. ACM, New York, pp 2–11
Magurran AE, McGill BJ (2011) Biological diversity, frontiers in measurement and assessment. Oxford University Press, Oxford
Mallat S (2009) A wavelet tour of signal processing: the sparse way. Elsevier, Amsterdam
Mbu Nyamsi RG, Aubin T, Bremond JC (1994) On the extraction of some time dependent parameters of an acoustic signal by means of the analytic signal concept. Its application to animal sound study. Bioacoustics 5:187–203
McGregor PK (ed) (2005) Animal communication networks. Cambridge University Press, Cambridge
Mellinger DK, Clark CW (2003) Blue whale (*Balaenoptera musculus*) sounds from the North Atlantic. J Acoust Soc Am 114(2):1108–1119
Merchant ND, Fristrup KM, Johnson MP, Tyack PL, Witt MJ, Blondel P, Parks SE (2015) Measuring acoustic habitats. Methods Ecol Evol 6:257–265

Meyer D, Buchta C (2016) proxy: distance and similarity measures. https://CRAN.R-project.org/package=proxy, R package
Mezquida DA, Martinez JL (2009) Platform for beehives monitoring based on sound analysis. a perpetual warehouse for swarm's daily activity. Span J Agric Res 7:824–828
Nason GP (2008) Wavelet methods in statistics with R. Springer, New York
Oppenheim AV, Schafer RW (1975) Digital signal processing. Prentice-Hall, Upper Saddle River
Oppenheim A, Schafer R (2004) From frequency to quefrency: a history of the cepstrum. IEEE Signal Process Mag 21:95–106
Percival DB, Walden AT (2000) Wavelet methods for time series analysis. Cambridge University Press, Cambridge
Pieretti N, Farina A, Morri FD (2011) A new methodology to infer the singing activity of an avian community:the acoustic complexity index (ACI). Ecol Indic 11:868–873
Quatieri TF (2002) Discrete-time speech signal processing: principles and practice. Pearson, Noida
Ramsay JO, Silverman BW (2005) Functional data analysis. Springer, New York
Rice AN, Land BR, Bass AH (2011) Nonlinear acoustic complexity in a fish "two-voice" system. Proc R Soc B: Biol Sci 278:3762–3768
Richter I, Koenders C, Auster HU, Frühauff D, Götz C, Heinisch P, Perschke C, Motschmann U, Stoll B, Altwegg K, Burch J, Carr C, Cupido E, Eriksson A, Henri P, Goldstein R, Lebreton JP, Mokashi P, Nemeth Z, Nilsson H, Rubin M, Szegö K, Tsurutani BT, Vallat C, Volwerk M, Glassmeier KH (2015) Ann Geophys 33:1031–1036
Risset JC (1978) Paradoxes de hauteur. Tech. rep., IRCAM - Centre Georges Pompidou
Robert P, Escoufier Y (1976) A unifying tool for linear multivariate statistical methods: the RV-coefficient. J R Stat Soc 25:257–265
Rodriguez A, Gasc A, Pavoine S, Grandcolas P, Gaucher P, Sueur J (2014) Temporal and spatial variability of animal sound within a neotropical forest. Eco Inform 21:133–143
Rossing TD (ed) (2007) Handbook of acoustics. Springer, New York
Royer JY, Chateau R, Dziak R, Bohnenstiehl D (2015) Seafloor seismicity, Antarctic ice-sounds, cetacean vocalizations and long-term ambient sound in the Indian ocean basin. Geophys J Int 202:748–762
Rumsey F, McCormick T (2002) Sound and recording. An introduction. Focal Press, Boston
Russell DA (2000) On the sound field radiated by a tuning fork. Am J Phys 68:1139–1145
Russell DA, Junell J, Ludwigsen DO (2013) Vector acoustic intensity around a tuning fork. Am J Phys 81:99–103
Sakoe H, Chiba S (1978) Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans Acoust Speech Signal Process 26:43–49
Schafer RM (1977) The soundscape: our sonic environment and the tuning of the world. Destiny Books, Rochester
Schloerke B, Crowley J, Cook D, Briatte F, Marbach M, Thoen E, Elberg A, Larmarange J (2017) GGally: Extension to 'ggplot2'. https://CRAN.R-project.org/package=GGally, R package
Senin P (2015) jmotif: tools for time series analysis based on symbolic aggregate discretization. http://CRAN.R-project.org/package=jmotif, R package
Shannon CE (1949) Communication in the presence of noise. Proc Inst Radio Eng 37:10–21
Shannon CE, Weaver W (1949) The mathematical theory of communication. Illinois University Press, Urbana
Sharan VR, Moir TJ (2016) An overview of applications and advancements in automatic sound recognition. Neurocomputing 200:22–34
Shepard RN (1964) Circularity in judgments of relative pitch. J Acoust Soc Am 36:2346–2353
Shumway RH, Stoffer DS (2006) Time series analysis and its applications with R examples. Springer, Cham
Snell RC, Milinazzo F (1993) Formant location from LPC analysis data. IEEE Trans Speech Audio Process 1:129–134
Soetaert K (2014) diagram: functions for visualising simple graphs (networks), plotting flow diagrams. https://CRAN.R-project.org/package=diagram, R package
Speaks CE (1999) Introduction to sound. Singular Publishing Group, San Diego/London

Staszewski WJ, Robertson AN (2007) Time-frequency and time-scale analyses for structural health monitoring. Philos Trans R Soc A: Math Phys Eng Sci 365:449–477
Stevens SS, Volkmann J, Newman EB (1937) A scale for the measurement of the psychological magnitude pitch. J Acoust Soc Am 8:185–190
Sueur J, Aubin T (2006) When males whistle at females: complex fm signals in cockroaches. Naturwissenschaften 93:500–505
Sueur J, Farina A (2015) Ecoacoustics: the ecological investigation and interpretation of environmental sound. Biosemiotics 26:493–502
Sueur J, Aubin T, Simonis C (2008a) seewave: a free modular tool for sound analysis and synthesis. Bioacoustics 18:213–226
Sueur J, Pavoine S, Hamerlynck O, Duvail S (2008b) Rapid acoustic survey for biodiversity appraisal. PLoS ONE 3:e4065
Sueur J, Mackie D, Windmill JFC (2011) So small, so loud: Extremely high sound pressure level from a pygmy aquatic insect (Corixidae, Micronectinae). PLoS ONE 6:e21089
Sueur J, Farina A, Gasc A, Pieretti N, Pavoine S (2014) Acoustic indices for biodiversity assessment and landscape investigation. Acta Acustica United Acustica 100:772–781
Sylvander M, Ponsolles C, Benahmed S, Fels JF (2007) Seismoacoustic recordings of small earthquakes in the Pyrenees: experimental results. Bull Seismol Soci Am 97:294–304
Teetor P (2011) R Cookbook. O'Reilly, Sebastopol
Tokuda IT (2017) Nonlinear dynamics and temporal analysis. In: Comparative bioacoustics: an overview. Bentham Science, Oak Park, pp 336–357
Towsey M, Planitz B, Nantes A, Wimmer J, Roe P (2012) A toolbox for animal call recognition. Bioacoustics 21:107–125
Towsey M, Wimmer J, Williamson I, Roe P (2014) The use of acoustic indices to determine avian species richness in audio-recordings of the environment. Eco Inform 21:110–119
Urbanek S (2013) audio: audio interface for R. https://CRAN.R-project.org/package=audio, R package
Venables WN, Ripley BD (2002) Modern applied statistics with S, 4th edn. Springer, New York
Villanueva-Rivera LJ, Pijanowski BC (2016) soundecology: soundscape ecology. https://CRAN.R-project.org/package=soundecology, R package
Villanueva-Rivera L, Pijanowski B, Doucette J, Pekin B (2011) A primer of acoustic analysis for landscape ecologists. Landsc Ecol 26:1233–1246
Welch PD (1967) The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. IEEE Trans Audio Electroacoust 15:70–73
Wickham H (2009) ggplot2. Elegant graphics for data analysis. Springer, Dordrecht
Wickham H (2014) Advanced R. Chapman & Hall/CRC Press, Boca Raton
Xie Y (2013) Dynamic documents with R and knitr. The R Series. Chapman & Hall/CRC, Boca Raton
Zwicker E (1961) Subdivision of the audible frequency range into critical bands (frequenzgruppen). J Acoust Soc Am 33:248–248

Index

J. Sueur, *Sound Analysis and Synthesis with R*, Use R!,
https://doi.org/10.1007/978-3-319-77647-7