extratitle

Speech Signal Processing with Praat

David Weenink

June 9, 2014

http://www.speechminded.com

ISBN-13:. . .

dedication

Contents

Part I.
Introduction to Praat

1. Introduction

The aim of this book is to give the non-mathematically oriented reader insight into the speech processing facilities of the computer program Praat. This program is freely available from Praat's website at http://www.praat.org and versions of the program exist for all major platforms: Linux, Windows and Mac OS. Versions for mobile platforms are not supported yet.

The Praat computer program has been developed by Paul Boersma and David Weenink, both at the Phonetics Institute of the University of Amsterdam. The program is used world wide by phoneticians, phonologists and speech researchers. Besides for the analysis of speech sounds, it is also used to analyze singing voices, music and even the vocalizations of animals like for example birds, dolphins, apes and elephants.

Although the program is still under heavy development the users interface has been relatively stable for a long period of time now, and the way of working that Praat enforces neither has changed much over the years. Development mainly concentrates on adding new functionality, extensions and under the hood improvements in stability. The current version of the program is 5.3.64.

The interface of Praat facilitates the manipulation of objects that model the world of speech signal analysis or the world of phonology. These objects, like for example a Sound, only exist in the memory of the computer and disappear when leaving the program.

Besides the analysis part, you may also make drawings of high quality that can be send to a printer or to a file and which can also be included in your document.

For automating your work you can extend the program with scripts. These scripts can be used either interactively or from batch, i.e. you can direct the program what to do from a text file. This comes in handy if you want to analyze a large number of files in a standard way. Besides automating your work, scripting is more powerful since it also facilitates live simulations by using a special *demo window* that can only be accessed via scripting.

Of course there is also a help facility in the program and Praat also comes with a number of tutorials.

Hopefully this book will give the reader more insight into the program and it also hopes to clarify some of the underlying signal processing theory.

1.1. Conventions used in this book

Because some actions in Praat are not directly visible and can only be reached via other menus we use a straightforward notation: we list all the menu items you have to pass to get to the command and use the ">" character as a separator. For example, the path to the command "Create Sound from formula..." may be given by the string "New > Sound > Create Sound from formula...". More explicit: you first have to click on the "New" menu, next you click on the "Sound" item inside this menu, and then you click on the wanted command.

1.2. Starting and leaving Praat

Normally one uses the program in a desktop environment. This can be on a machine with Linux, Windows or Mac OS. A (double) click on the icon will start the program. You quit Praat by either selecting the Quit command from the Praat menu or typing Ctrl-Q.

If you quit the program, all objects that have been created in your session will be destroyed. Of course, Praat will ask you if you want to save your data first before actually destroying these objects.

After you have started Praat, two windows appear on the screen as is show in figure 1.1. The actual decorations of these windows are platform specific and may depend on the chosen desktop theme. *For Windows and Linux users the **Praat** menu item appears at the top left position in the object window while for Macintosh users this menu item does not appear in the object window at all but is placed left on the menu at the top of the display.*

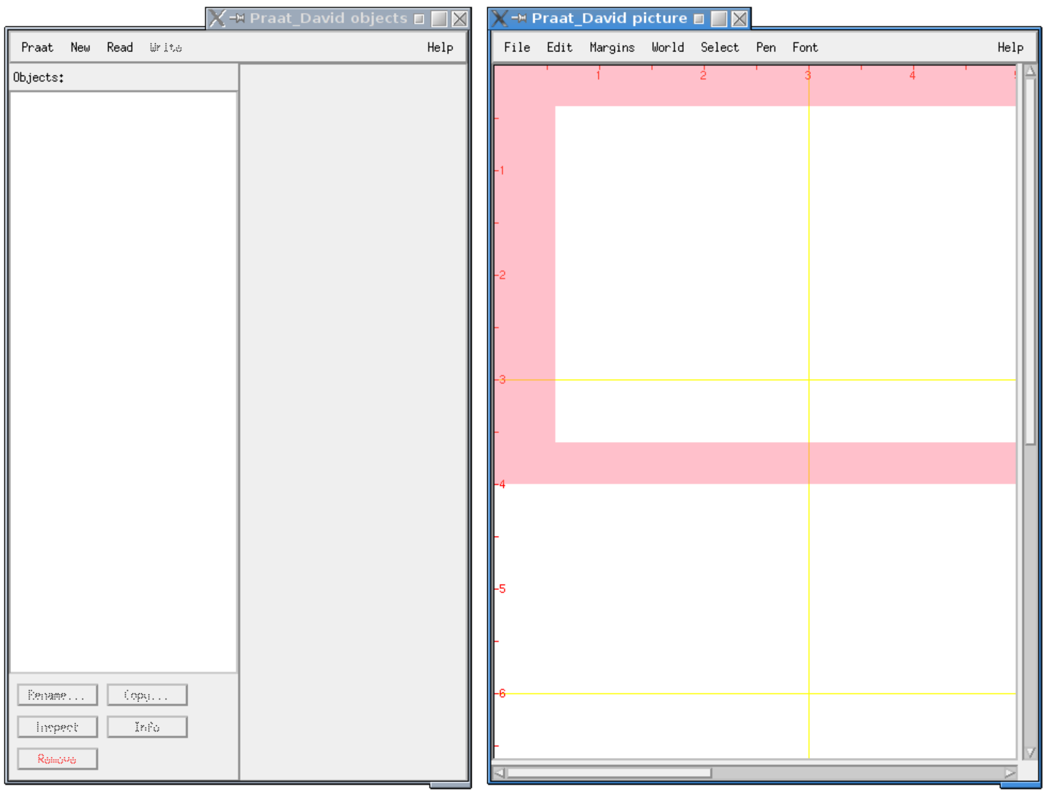
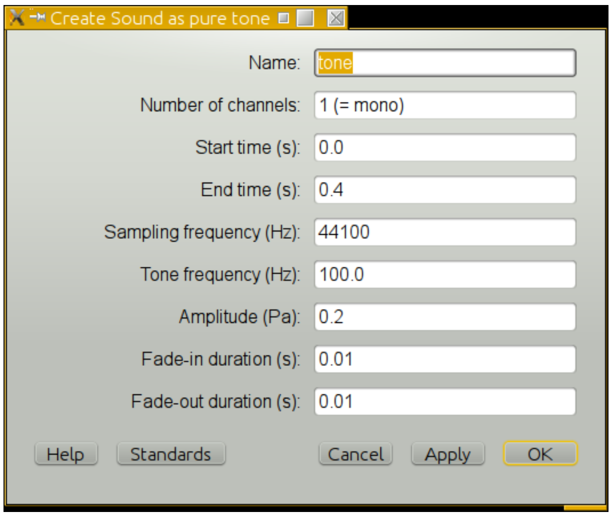


Figure 1.1.: The initial Praat object and picture window. On a Macintosh computer the top left Praat menu is not in the object window but always appears at the top left in the menu bar at the top of the display.

If this is your first try with Praat you could try now to create a new sound signal by choosing New > Sound > Create Sound from formula.... In figure 1.2 the form that appears on the screen is shown. Now just push the OK button and a new sound object appears in the list of objects. This sound object is displayed in the list of objects and starts with a number. This number is a unique identification number, its “id”. After a fresh start of Praat, the id always starts with number 1. With every newly created object the number is increased by one. Later in the book where scripting is explained we learn how to identify objects by this number.



Name:	tone
Number of channels:	1 (= mono)
Start time (s):	0.0
End time (s):	0.4
Sampling frequency (Hz):	44100
Tone frequency (Hz):	100.0
Amplitude (Pa):	0.2
Fade-in duration (s):	0.01
Fade-out duration (s):	0.01

Figure 1.2.: The New > Sound > Create Sound as pure tone... command.

1.3. How we specify a command in this book

There are a great many commands in Praat, too many to show or to explain in this book. For some of these commands we will show in two distinct ways how we can interact with them. We can show the complete form of a command explicitly as for example happens in Fig. 1.2 for the “Create Sound as pure tone...” command. We will not show this explicit form too often because of limitations in space and generally we will use an alternative, more compact, way. We show the command as if used in the scripting language that goes with Praat. Here we anticipate chapter 4 which tells more about scripting. For example, the form in Fig. 1.2 will be represented as follows:

```
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100,
... 100, 0.2, 0.01, 0.01)
```

We write down the command as on the button and replace the three dots (...) with a colon (:), followed by the arguments of the command separated with comma’s (,). The order of the arguments on the line is from left to right as how they appear in the form from top to bottom. The first argument of this Praat command is the name you want the new sound to have. Because this name is a text it has double quotes around it. Then follows a number (1) because the form shows that it expects a number in this field. The following arguments are also numbers (0.0, 0.4, 44100, 100.0, 0.2, 0.01, 0.01) as they correspond, respectively, to the “Start time (s)”, the “End time (s)”, the “Sampling frequency (Hz)”, “The tone frequency (Hz)”, the “Amplitude (Pa)”, the “Fade-in duration (s)” and the “Fade-out duration (s)” fields of the form. Because this particular command is too long to fit on one line we had to break it up in two lines. The three dots that start the second line mean that the rest of this line is a continuation of the previous line. For other commands we follow the same procedure: we always start with the name of the command and the rest of the arguments follow the fields of the form from top to bottom as we showed above. If a command does not have an associated form, as happens with all commands that do not end with three dots “...”, only the command itself is displayed. For example a Sound object has a Play command which is immediately executed if clicked. We can mimic this one in a script as:

```
Play
```

In the sequel we will mostly use this unambiguous notation to represent Praat commands.

1.4. How to make pictures

All picture drawing in Praat takes place in the picture window. The following steps may be involved:

1. Select an area in the picture window where you want to draw;
   Default an area for drawing has already been selected. This area is called the *viewport* and is indicated with a pink colored border. You can see this pink selection in the right of figure 1.1, where the viewport extends beyond the visible display area of the picture window. The white area *inside* is where the actual drawing takes place, the pink *outside* is used to garnish the drawing with, for example, tick marks, axis labels etc. The size of the outside area depends on the font size chosen: the larger the font the larger the outside area. You can manipulate the drawing area with the mouse and/or a menu command.

2. Draw the picture with one of the selected objects’ drawing methods;
   Many objects like a sound or a spectrum have drawing methods, i.e. if you select a sound object, a “Draw...” command exists in the dynamic menu.

3. Refine the picture with commands from the picture window’s menus;
   For example, you might want to add extra marks to the axes.

4. Save the picture.

The last two steps are optional.

1.4.1. Drawing examples

In the first example we start by selecting “Font size... 10” from the Font menu. Drag the mouse in the picture window and select a viewport drawing area. Chose the “Draw inner box” command that you can find in the Margins menu of the picture window. This command draws a rectangle at the border between the inner and outer part of the viewport. Change the font size

to 12 and draw a new rectangle. Change the font size to 18 and draw another rectangle. Three boxes of different sizes have appeared in the drawing area. Besides making your first drawing this has shown you the influence of font size on the drawing area. The larger the font the smaller the actual drawing area becomes because larger fonts need more space for garnishing a drawing.
In the following example we like to draw a sound of one second duration and put a vertical mark at 0.7 s. First erase everything in the picture window by selecting the “Erase all” command from the Edit menu.

1. Select an area where you want to draw, i.e. the viewport.
   Set the font size to 10.

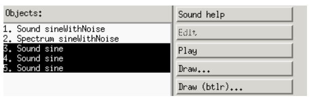
2. Create a sound of 1 s duration. If you don’t know how you can use the New > Sound > Create Sound from formula... command. This command will create a noisy sound of 1 s duration.

3. (Select the sound,) click the Draw... command and accept the defaults by clicking OK. Now the sound has been drawn in the picture window.

4. In the Margins menu in the picture window select the “One mark bottom...” command. Type the number 0.7 in the field labeled “Position”. Select “Write number”, “Draw tick” and “Draw dotted line” and leave the text field empty. After the click on OK a dotted vertical line appears in the drawn sound.

1.5. How to make a selection in Praat

An important aspect of working with Praat is its object oriented interface. Object oriented means that you first have to select an object before you can do anything (with it), i.e. *selection precedes action*. Making a selection of an object involves moving the mouse in the list of objects area to a location above the particular object and then clicking the (left) mouse button. If you want to select more than one object in the list of objects two possibilities exist:



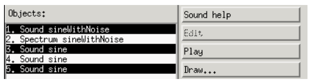


Figure 1.3.: On the left a contiguous selection of the objects numbered 3, 4 and 5. On the right a discontiguous selection of the objects numbered 1, 3 and 5.

1. A *contiguous* selection. This is exemplified by the left display in figure 1.3 where the consecutive objects numbered 3, 4 and 5 have been selected. If you have several objects in the list of objects then dragging the mouse over the list may result in a contiguous selection. Another way to obtain a contiguous selection of objects is to select the first object with a mouse click, release the mouse, move the mouse to the last object in the list that you want to include in you selection, push the Shift button on the keyboard and

click with the mouse on the object. This is called a Shift-click. It results in a contiguous selection that includes all the objects from the first selected one up to and including the last Shift-clicked one.

2. A *discontiguous* selection as shown in the right display of figure 1.3 where the objects numbered 1, 3 and 5 have been selected. The selection is interrupted by one or more gaps of deselected objects. The way to achieve a discontiguous selection is by Ctrl-click extension. Holding the Ctrl button down while clicking on an object in the list of objects toggles a selection, i.e. if the object was not selected then it will be selected, if the object was already selected it will be deselected.

With the Shift-click and the Ctrl-click options there are many ways to make a discontiguous selection. For example, the selection in figure 1.3 could be accomplished in several ways. We mention two of them. Firstly, we could have started by selecting the object with number 1 and subsequently, with two Ctrl-clicks, extend the selection with the objects numbered 3 and 5. Secondly, we might have started from a contiguous selection by dragging the mouse from object 1 to 5 and subsequently deselecting the objects numbered 2 and 4 by Ctrl-clicking them.

1.6. Terminology and book layout

A lot of different words are used to draw attention to certain aspects of sound signals. The most important sound signal in this book is the *speech* signal. We will visualize and analyse speech sound signals. Speech normally is a transient pressure wave in the air and it disappears after being uttered. One of the things we cover is how to *record*, *copy* and *store* speech sounds on a computer. This will involve terms like *microphone*, *oversteering*, *analog*, *digital*, *quantization*, *sampling*, *aliasing*, *bits*, *bytes* and *file formats*. We cover this part in chapter 3. A speech sound can be analysed as a *function* of time. The essential part of the mathematics of functions will be treated in appendix A. We will talk about the *amplitude* of a sound signal and how to visualise a sound in chapter 2. Aspects of the *pitch* of a speech sound are the topic of chapter 5. The *intensity* of a sound as a function of time is explained in chapter 6. We will talk about *frequency* and its associated visualization the *spectrum* in chapter 7. The mathematical functions that we need here are *sine* and *cosine* and these functions are associated with *pure tones* that have *amplitude*, *frequency* and *phase*. *Time-frequency displays* like the *spectrogram* are explained in chapter 8. Formant analysis is the topic of chapter 13. A very important part of this book is devoted to *scripting* because we feel that this is essential for speech analysis where one, most of the time, is confronted with a huge number of speech sound files that have to be analysed. Scripting involves making a list of commands that can, hopefully, be executed without intervention on a selected sample of these files. This necessarily also involves collecting analysis results in an appropriate format form like for example a table. Chapter 4 is devoted to scripting.

2. The sound signal

2.1. Introduction

A human who speaks, produces sounds which can be heard by anyone nearby. These kind of sounds we will mostly study in this book, i.e. the sounds that we can hear. We must realize however that there are many more sounds than the ones *we* can hear. The sounds a human being can hear are limited to a subset of all possible sounds. For example a dog can hear *ultrasound*, i.e. sounds that are too "high" for us to hear. Sounds that are too "low" for us to hear are called *infrasounds*. Infrasounds can travel large distances and this is why whales and elephants can communicate over very large distances using infrasounds. The descriptions "low" and "high" are connected to the physical concept of *frequency*. The branch of physics that studies sound is called *acoustics*.

2.2. Acoustics

From acoustics we learn that sound is defined as a *mechanical disturbance from a state of equilibrium that propagates through an elastic medium*. An elastic medium is capable of resuming its original shape after stretching or compressing. A disturbance may be produced in several ways like plucking a guitar string, pinching a tuning fork or by the opening and rapid closing of the vocal folds. This disturbance produces a sudden local increase in pressure, i.e. a local compression of air. Since the medium is elastic, the compression is not permanent and therefore the compressed region will rebound. In doing so it will compress an adjacent region, which will then again rebound and so on. The result of this cycle of repetition is a *compression wave* followed by a *rarefaction wave* as each region rebounds. The waves thus generated propagate through the medium with a speed that depends on several factors like the elasticity, density, and the temperature of the material. The propagation *velocity* of a sound pressure wave in air at room temperature is approximately 340 m/s. [1] It is important to realize that it is not the air *particles* in the air that carry the sound to your ear but it is the compression *wave* that propagates and carries the sound. Therefore in order to study the physical aspects of sounds we don't have to study particle physics but instead we have to know some things about waves, in particular sound waves.

In many textbooks sound waves are illustrated with the sounding of a tuning fork and we will continue in this tradition. A tuning fork is an acoustic resonator in the form of a two-pronged U-shaped fork of elastic metal (usually steel). It resonates at a specific constant pitch

[1]This corresponds to 1224 km/h which may appear like a very large velocity but it is nothing compared to the speed of light which is nearly 300,000 km/s. This means that after you see a lightning flash at one kilometer distance, it will take the sound another three seconds to reach your ear.

when set vibrating by striking it against a surface or with an object, and emits a pure musical tone. The pitch that a particular tuning fork generates depends on the length of the two prongs. Its main use is as a standard of pitch to tune other musical instruments. In the upper panel of

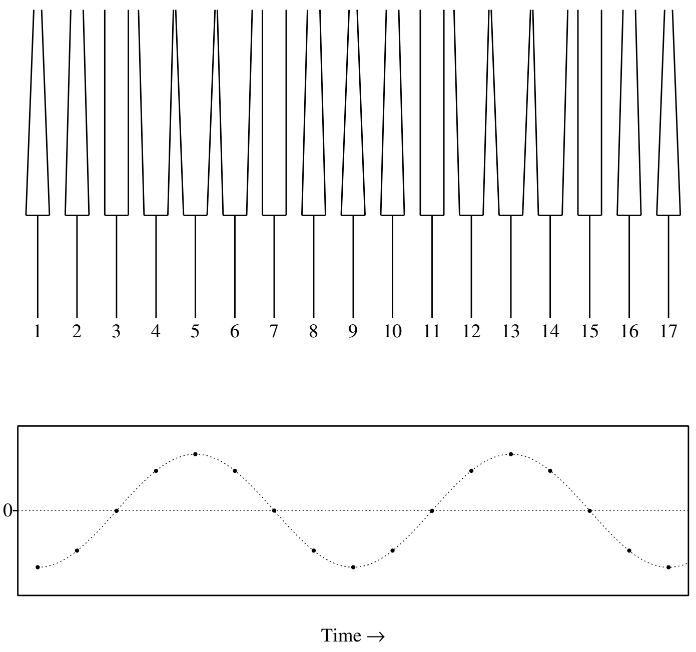


Figure 2.1.: Upper panel: seventeen phases of the displacement of the prongs of a tuning fork after it has been made to sound by pinching the two prongs together. Lower panel: the displacement of the right-hand prong as a function of time. Inward displacement is negative, outward displacement positive. The baseline corresponds to the rest position.

figure 2.1 we have tried to visualize the movement of the prongs of a tuning fork at seventeen regularly spaced time points. The tuning fork was made to sound by pinching the two prongs together at time point 1. Now the two prongs of the fork are most close together and because of the elasticity of the material, they will immediately start to move back to their neutral position. However, when they reach the neutral position, at time point 3, they have so much velocity and consequently overshoot, moving more outward. The prongs therefore pass the neutral position at time point 3 and reach a maximum outward displacement, at time point 5, and from there they start moving inwards again. At time point 7 they pass the neutral position again, overshoot, and move further inward. At time point 9 they have completed one cycle and are back at approximately the same displacement as at time point 1. They now move outward again and a new cycle of outward and inward movement starts. This goes on until eventually the movement dies. The movement of each prong is called a *periodic* movement because the movement repeats itself in regular intervals or periods.

In the lower panel of figure 2.1 we have plotted the displacement of one of the prongs, in this case the right prong, as a function of time. Relative to the neutral position, a displacement inward has been given a negative sign while a displacement outward has been given a positive sign. The baseline at 0 therefore corresponds to the neutral position. The dots correspond to the time points displayed in the upper panel. A smooth curve has been drawn through the displacement points to show the intermediate values of the displacements. The curve starts at the most negative value for the displacement since the prongs are most closely together here. It then moves upwards towards less negative values because the displacement w.r.t. to the neutral position becomes less, crossing the neutral position at time point 3. It then reaches the most positive value at time point 5 because the prongs are at their maximum distance apart which is at their maximum outward position. The curve goes down again because the prong starts moving inward again. At time point 7 the prong passes the neutral position and the value of the displacement curve is zero. At time point 9 the value of the curve is equal again to the value at time point 1. Now the cycle starts all over again.

This curve, that describes the motion of the prong, is called a *sinusoid*, which means like-a-sine or sine-like. It is one of the most important curves that exist in science. The displacement of the prong of the tuning fork follows a sinusoidal curve and the prong is said to move in a *simple harmonic motion*. The sound produced by a tuning curve is called a *pure tone*.

A sinusoidal curve describes the motion of many *oscillating* bodies like a spring or a pendulum. As the lower panel visualizes, the sinusoid curve is periodic: a sinusoid repeats itself. The curve that starts at time point 1 repeats starting at time point 9. A closer look at the curve reveals that the periodicity of the curve could start at *any* place. For example, if we had drawn an 18th and a 19th point, the shape of the curve from time points 10 to 18 would equal the shape of the curve from time points 2 to 10 and the shape from time points 3 to 11 would equal the shape from time points 11 to 19. In fact we could pick any random point at the time axis follow the curve and discover the next point from which the shape of the curve equals the traced curve. From all the possible starting points two of these curves have gotten a special name: the curve that starts at time point 3 is called a *sine* curve, and the curve that starts at time point 5 is called a *cosine* curve. As the figure shows both sine and cosine are periodic *functions*, that have a lot of similarities. In fact, the only difference between the two is that the sine curve starts with a zero value and the cosine curve starts at the positive extreme. In words, one period of a sine starts at zero, goes up to an extreme value and then goes down below zero to a negative extreme and then goes up again to zero. In the chapter A, the mathematical introduction, we give a lot more information about the sine and cosine functions. Because these functions are so important a special branch of mathematics called *trigonometry* is involved dealing with them.

What is very nice about the sinusoid functions is that they also describe the motion of air particles immediately in the neighborhood of the prongs of the tuning fork. Let us focus on the right-hand prong as we have before. In the left panel of figure 2.2, the changing positions of air "particles" are shown during the transmission of a simple sound from the right-hand prong. The rows represents*, at different times,* the position*s* of the air particles along a certain distance on a virtual straight line that extends from the sound source origin outwards. We must take "particle" not too literally because we do not mean the individual molecules; we mean something like the average position of a small volume of air. At the bottom of the left panel, a wave is started by a local compression of air particles by some external disturbance, like for

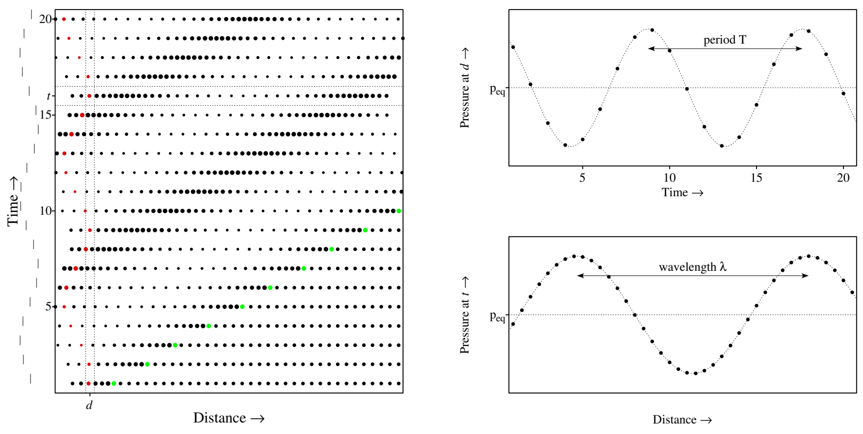


Figure 2.2.: Left panel: Propagation of sound in air visualized by the changing position of air "particles" during the transmission of a simple periodic sound wave. The rows represent the position of the air particles at successive points in time along a distance on a straight line that extends from the sound source. The wave front is indicated by a particle marked with green color. The displacement of one single air particle is displayed in red color. The changing position of the right-hand prong of the tuning fork is shown on the left with small vertical bars. Upper right panel: the pressure at distance d as a function of time. Bottom right panel: pressure variation at time t as a function of distance.

example the right-hand prong of a tuning fork. The tuning fork sets air particles in motion and these particles set other particles in motion and so on resulting in a local compression of air particles. Because of this compression the air particles at the left in the first row are closer to each other than in the rest of the row where the particles are still in their normal position. There is only a compression at this time point at the start of the row, the rest of air particles farther away from the sound source are still in their neutral position. The green colored air particles mark what is called the *wavefront*. The picture makes clear that in the course of time the wavefront continues to travel to the right, further away from the source. The velocity of travel is the *velocity of sound*. Near the wavefront the particles due to the compression are closer together and therefore the pressure is larger than average. At positions where there are less particles than normal the pressure is less than average. We emphasize again that although the wave travels to the right the air particles do not. Each particle simply follows a harmonic motion; to show this we have given one particle a red color which enables us to track its position as time develops. It shows that the particle's displacement nicely moves along the neutral position. This illustrates that the sound not travels with the particles but with the wave that creates the harmonic movement of these particles.

As the left panel shows, the amount of air particles at a certain position varies with time. We know that the amount of air particles within a certain space is directly related to the physical measure called *pressure*; the more particles are present, the higher the pressure is. Pressure is

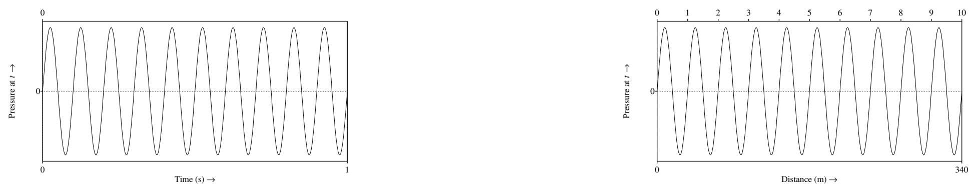


Figure 2.3.: Left pane: The pressure variation during an interval of 1 s at a certain position due to a 10 Hz sound. Right pane: the pressure variation over a distance of 340 m for a 10 Hz sound.

expressed in units of pascal, abbreviated as Pa.[2] The pressure in a small volume at a distance d from the sound source has been plotted in the top right panel of figure 2.2. The pressure as a function of time shows a sinusoidal curve and its period has been marked with T. The number of wave periods that fit into one second is called the wave's *frequency*; the dimension or the unit of frequency is called the hertz, abbreviated as Hz.[3] The hertz unit is equivalent to cycles per second and has dimension one over time (1/s). If a period would last one second its frequency would be one hertz. There is an inverse relation between the frequency and the period of a wave:

$$f = \frac{1}{T}. \tag{2.1}$$

In this formula f is the frequency in hertz and T is the period in seconds. Note that the units on both sides are the same because the unit of the period T is in seconds and therefore the unit of $1/T$ is 1/s which is the same unit as the hertz. The left panel of figure 2.3 shows the pressure variation during an interval of 1 s at a certain position due to a 10 Hz sound. A frequency of 10 Hz means that during this 1 s the sound pressure completes 10 cycles of pressure variations, i.e. 10 periods will fit in this time interval. Consequently each period will last 0.1 s (= $1/10$). In the figure the starting point was chosen to be at zero pressure variation but this does not matter. Had we chosen another starting position then also exactly 10 cycles would have been completed. A frequency of 20 Hz has a period of 0.05 s (= $1/20$) because exactly 20 periods fit in one second. The sound frequencies that we are normally concerned with in speech sounds are higher than these. Although the human ear is sensitive to sounds with frequencies between 20 Hz and approximately 20000 Hz (=20 kHz), the most important information for speech sounds is limited to the range from say 200 Hz to some 5000 Hz. For analog telephone conversations the frequency range is deliberately limited to frequencies between 300 and 3300 Hz.

The right bottom panel of figure 2.2 shows the pressure as a function of *distance* at the time point t (bounded on the vertical axis with horizontal dotted lines). Again a sinusoidal curve appears whose period now is called the *wavelength*. Because the horizontal scale is now distance instead of time, the dimension of the wavelength is also expressed in the unit of dis-

[2]The normal air pressure is approximately 10^5 Pa.

[3]The hertz unit is in honor of the German physicist Heinrich Hertz (1857–1894). Many units have been named to honor eminent scientists, we name the watt (unit of power), joule (unit of energy), newton (unit of force), tesla (unit of magnetic field), pascal (unit of pressure), ampere (unit of electrical current) and volt (unit of electric potential).

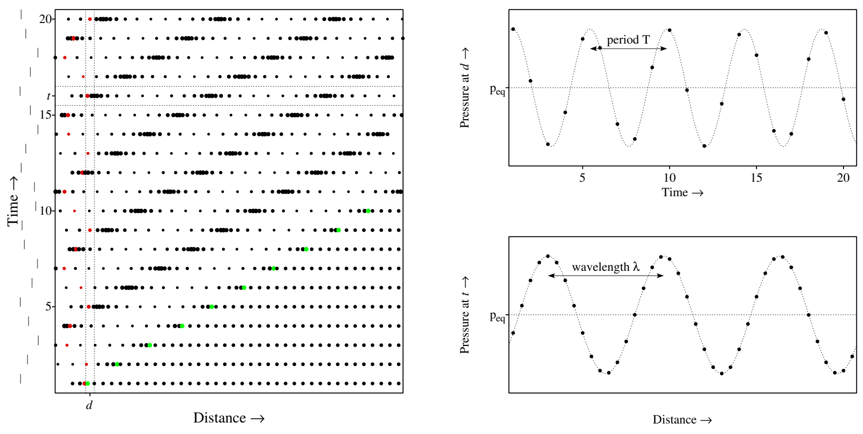


Figure 2.4.: Propagation of sound in air as in figure 2.4.

tance, i.e. meters. Wavelength is often indicated with the Greek character λ (lambda). There is a relation between the wavelength of a wave and its frequency which says that wavelength times frequency is a constant. The constant happens to be the velocity of wave propagation in the medium, in our case the velocity of sound in air. For frequency f, wavelength λ en velocity v the formula reads:

$$\lambda . f = v. \tag{2.2}$$

This formula shows just like equation 2.1 an inverse relationship, but now between wavelength and frequency (we can rewrite the equation above as $\lambda = v/f$). Note that the units are correct since the left side has the unit of the wavelength (m) multiplied by the unit of frequency (1/s) which results in a combined unit of m/s, the unit of velocity. We can easily see that this relation is true: if we assume that the velocity v of a wave does not depend on frequency but is a constant, then we know that in one second the wave will have traveled a distance of v meters. Because the frequency equals f the wave has oscillated f times during this second and so the distance traveled equals f wavelengths. Therefore f wavelengths should equal the distance covered, and therefore $f . \lambda = v$. If we would measure during two seconds the distance covered would be twice as large and equal $2v$ meters and twice as many wavelengths would fit in so $2f . \lambda = 2v$ which reduces to $f . \lambda = v$ again. We could repeat this argument for any length of time. If the frequency of a tone increases, its wavelength decreases, or, if the frequency decreases its wavelength increases. In the right panel of figure 2.3 the pressure variation of a 10 Hz sound has been plotted over a distance of 340 m, i.e. the distance a sound would travel in a one second interval. As the figure makes clear the 10 periods of the sound cover a distance of 340 m and therefore the wavelength of a 10 Hz sound will be one tenth of the distance traveled in this one second and equal 34 m. If we increase the sound's frequency to 100 Hz then a hundred periods have to fit into the same distance and the wavelength decreases to 3.4 m. Given the formula above we now can easily calculate that a wavelength of 0.17 m, which happens to be the average male's vocal tract length, corresponds to a frequency of 2000 Hz.

In figure 2.4 we show again the propagation of sound in air. All panes have exactly the same scales as in figure 2.2, however, the frequency of the source signal has been doubled. This results in a doubling of the number of compressions in the same amount of time as can be judged by comparing the left panels in both figures. Because of the *doubling* of the frequency, the period has been *halved*, as a comparison of the upper-right panels in figures 2.2 and 2.4 shows. As for the wavelength, the corresponding lower panels shows that because of the frequency doubling the wavelength has been halved, in accordance with the formula above.

We have now covered how a sound is transmitted in a medium. Two further things have to be said. The first is that without a medium there will be no sound, in other words, in a vacuum there can be no sound because sound, in contrast to light, needs a transport medium. The second is that besides the frequency also the loudness of a simple sound can vary. The only way loudness information can be transmitted is by a variation of the pressure, i.e. the *density* of the air particles.

The sound of a tuning fork is one of the most simple sounds possible because it is a sound wave with a fixed frequency; it is called a pure tone. Most of sounds we encounter, however, are not pure tones but more complex sounds. But however complex they may be, they still are transmitted in the same way as a pure tone: by a wave that propagates as a compression and rarefaction of air particles. These compressions and rarefactions can be recorded as local changes in pressure by a microphone and subsequently transduced into an electrical signal. The electrical signal can be stored in a computer as numbers. In chapter 3 we will go into more detail how this can be accomplished. For now it suffices to say that when we say "*the sound object in Praat represents the acoustical pressure variations of a sound in air*" you hopefully will know from the preceding discussion what we are talking about.

2.2.1. The nasty details of pure tone specifications

In the preceding sections we have seen that the pressure variation of the sound wave from a tuning fork can be modeled with a sinusoid. However, we did not specify the sinoid in mathematical terms. In this section we will explain how the mathematical sine function can specify a tone in terms of frequency and time.

Let us first find out more about the sine function. From the mathematical introduction we know that the sine function can be written as $\sin(x)$ where the argument of the sine, x, is some dimensionless variable. In the top left pane of figure 2.5 we show how the value of the function $\sin(x)$ varies with its argument x in the interval from 0 to 6.283. The end of the interval could also have been displayed as 2π but instead we have displayed it as the number 6.283 to emphasize that 2π really is a number and not something magical. The period of $\sin(x)$ is 2π. This means that the value of $\sin(x)$ for some value x and for another value $x + 2\pi$ are always equal, whatever the value of x may be. In a formula this says that always

$$\sin(x) = \sin(x + 2\pi). \tag{2.3}$$

We can reformulate this formula as: $\sin(x) = \sin(x + 2m\pi)$ for all integer values of m.[4] The

[4]It is easy to see that this must be true by noting that if $\sin(y) = \sin(y + 2\pi)$ for all y then if $y = x + 2\pi$ we get $\sin(x + 2\pi) = sin(x + 2\pi + 2\pi) = sin(x + 2 \cdot 2\pi)$. With the help of equation (2.3) this results in $\sin(x) = \sin(x + 2 \cdot 2\pi)$. In the same way we can show that every increase of the argument x by 2π results in the same value.

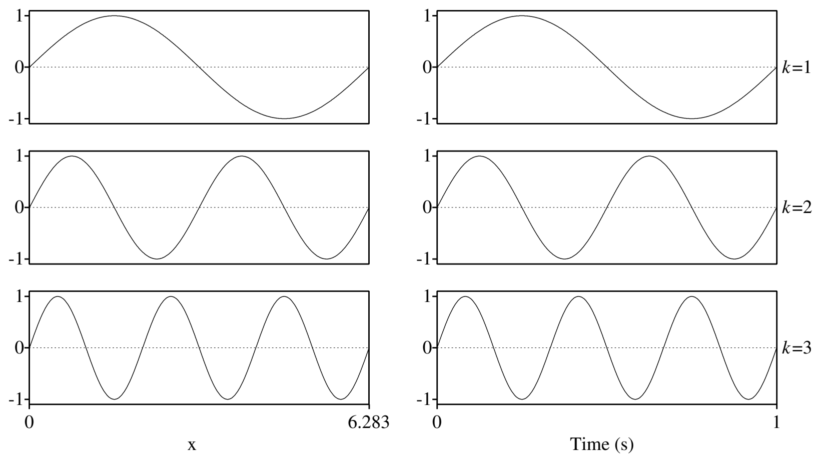


Figure 2.5.: Different representations of sinusoidal functions. In the left column we display the function $\sin(kx)$, where the variable x runs from 0 to approximately 6.283 (2π). From top to bottom the values for the parameter k are 1, 2 and 3, respectively. In the right column the displayed function is $\sin(2\pi kx)$ where x runs from 0 to 1.

legend of the figure says that we display functions $\sin(kx)$ which equals $\sin(x)$ if k equals 1. The most important thing to note from the top left panel for the function $\sin(x)$ is that *exactly one period fits in the interval from 0 to* 2π and that the length of one period therefore equals 2π. The amplitude of the sine varies between $+1$ and -1. The next function in the left column is $\sin(2x)$. This function has two periods on the same interval because when x runs from 0 to 2π, the argument $2x$ runs from 0 to 4π which means that when x is halfway at the value π, the argument $2x$ has already reached the value 2π and we know that one period has been tracked. Since the sine is a periodic function we know that when x runs through the second half, i.e. the interval from π to 2π another period of the sine will be traced. In the third panel in the left column where the function $\sin(3x)$ is displayed there are three periods on the interval 0 to 2π because the argument of $\sin(3x)$, i.e. $3x$ now runs from 0 to 6π. From the arguments above we can make the following generalization: the function $sin(kx)$ shows k periods when x runs from 0 to 2π. In the figure k was an integer number but this was just to get a nice display. You may now be able to understand that k may be any real number. For example if the value of k were 2.788 then the function $\sin(kx)$ would not show an integral number of periods when x runs from 0 to 2π but only show 2.788 periods.

From the preceding paragraph we have learned that the function $\sin(kx)$ shows k periods when x runs from 0 to 2π. We further know that if the x variable were time then k periods in a segment of duration 2π seconds would correspond to a frequency $f = \frac{k}{2\pi}$ since the frequency is the number of periods per *second*. This shows that k can almost be interpreted as frequency apart from a factor of 2π. So why not include this factor in the argument? To show the effect, we display in the right column of figure 2.5 how the function $\sin(2\pi kx)$ varies if k varies from 1 to 3. In the display we have reduced the interval for x from 0 to 1. For $k = 1$ we see 1 period, for $k = 2$ we see 2 periods and for $k = 3$ there are 3 periods during the 1 second interval. This is very nice because if x were time than k would correspond to frequency! So why not change the notation and write $\sin(2\pi f t)$ instead, where f is the frequency in Hz and t is the time in seconds. Although frequency and time have different dimensions or units, the product ft is dimensionless since the dimension of the product of "1/s" for the frequency and "s" for the time cancels out. The formula $\sin(2\pi f t)$ expresses nicely how the amplitude of a wave with frequency f varies as a function of time.[5] Let us use this formula to express the pressure variation as a function of time at a certain point due to a sounding tuning fork of frequency f. We write:

$$p(t) = \sin(2\pi f t).$$

Would this be the correct formula? Well, . . .almost. Two things have still have to be arranged in the formula, *amplitude* and *phase*.

As figure 2.5 shows, the amplitudes of all the sines in all the panes vary between the values +1 and −1. Or put in another way: no matter what the argument of a sine function is, the result is always a number between +1 and −1. The unit of pressure is the pascal, abbreviated as Pa. The formula above therefore only describes pressure variations between +1 and −1 Pa.[6] We want to be more flexible than this and be able to describe variations between say +0.001 and −0.001 Pa. This can be easily fixed by extending the formula above with a *scale factor*. We now write:

$$p(t) = A\sin(2\pi f t).$$

The pressure now varies between the numbers $+A$ and $-A$ and by choosing an appropriate value for A we can allow for any range of pressure variations. This shows how we can vary the amplitude of a sound. The amplitude correlates with the *loudness* of a sound, the larger the amplitude the louder the sound. We are almost there, one tiny step to take. Take a look at the upper right panel of figure 2.2 again where the pressure variation as a function of time was displayed. The curve certainly looks sinusoidal but the initial amplitude is definitely not zero as the sine functions in figure 2.5. Clearly we have to be able to manipulate the amplitude value at the start, i.e. the phase in the cycle of the sine where it should start. We know that for $t = 0$ the argument of $A\sin(2\pi f t)$, i.e. the term $2\pi f t$, is also zero. The only way to guarantee that for $t = 0$ the argument is unequal zero is by adding a constant number to it. This constant is called the *phase* and the most used symbol for it is the Greek letter ϕ. Finally we can now write the complete general description of the pressure variation at a certain point in space due to the sound of a tuning fork:

$$p(t) = A\sin(2\pi f t + \phi). \qquad (2.4)$$

[5] An alternative formulation is based on the following. First we make a purely cosmetic change and move the 2π in $\sin(2\pi k x)$ closer to the x to get $\sin(k2\pi x)$. Then we make the obvious conclusion that if x varies from 0 to 1 then $2\pi x$ varies from 0 to 2π. Therefore the curve of the function $\sin(k2\pi x)$ when x runs from 0 to 1 is identical to the curve of $\sin(kx)$ when x runs from 0 to 2π.

[6] The pascal is used as the unit of air pressure. It is a derived unit of measurement and defined as newtons per square meter (N/m^2). The ambient air pressure is about 100,000 Pa. Our ear can detect air pressure variations as small as approximately 0.00002 Pa for a sine wave with a frequency of 1000 Hz. The Praat help page on "sound pressure level" will show additional information on the subject.

The phase parameter ϕ facilitates the opportunity to let the sine curve start at any amplitude value between $+1$ and -1, the parameter A can scale these values and the frequency parameter f scales the number of cycles per second. In order to be able to start at any amplitude of the sine at $t = 0$, the value of ϕ must be able to vary over a range of 2π. In figure 2.6 we display the function $\sin(x + \phi)$ *as a function of* ϕ at $x = 0$ where ϕ runs from -2π to $+2\pi$. Although for $x = 0$ the function $\sin(x + \phi)$ reduces to $\sin(\phi)$, we have explicitely written $\sin(x + \phi)$ to emphasize that we are interested in the *starting* amplitude. The figure reconfirmes what we already know now, a range of π for the phase ϕ is sufficient to cover all posible starting amplitudes. An infinite number of intervals of length π are possible, however, the most convenient one is the interval from $-\pi/2$ to $+\pi/2$. As the figure makes clear, this interval for ϕ cover all possible amplitude variations of a sine and therefore equation 2.4 is sufficient to model all possible pressure variations of frequency f. We repeat that it is not the *absolute* pressure that we are interested in, only the pressure variations around the ambient pressure.[7]

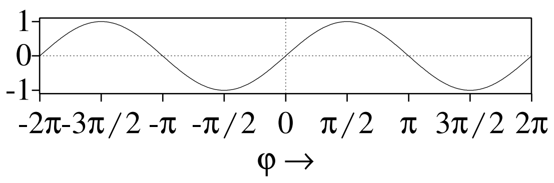


Figure 2.6.: The function $\sin(x + \phi)$ for $x = 0$ where ϕ runs from -2π to $+2\pi$.

2.3. The sound editor

In figure 2.7 we show the sound editor, it is the window that appears when you click the Edit command of a selected sound. The sound editor shows vertically the sound amplitude as a function of the time which is displayed horizontally. The most important parts of the editor have been numbered in the figure:

1. The title bar. It shows, from left to right, the identification id of the sound, the type of object being edited, i.e. "Sound" and the name of the sound being edited. This is typical for the title bar of all editors in Praat: they always show the id, the type of object and the name of the object, respectively.

2. The menu bar. Besides options for displaying the sound amplitudes we can display various other aspects of the sound like the spectrogram, pitch, intensity, formants and pulses. In figure 2.7 we have explicitly chosen to display none of these. On the other hand in figure 2.8 all these aspects of the sound are displayed:

[7]The absolute pressure for the sound of a tuning fork can be written as $p'(t) = p_0 + A\sin(2\pi ft + \phi)$, where p_o is the ambient pressure.

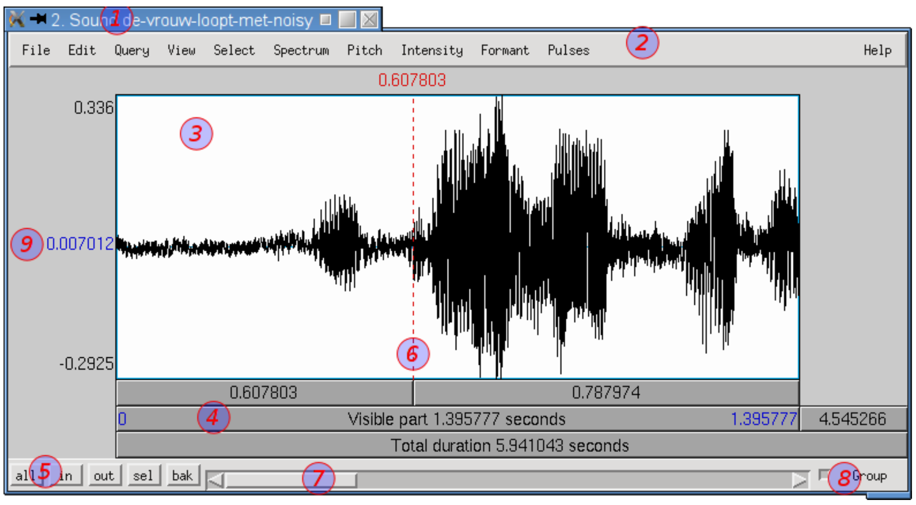


Figure 2.7.: The basic sound editor. The numbered parts are explained in the text.

- The spectrogram is shown as grey values in the drawing area just below the sound amplitude area. The horizontal dimension of a spectrogram represents time. The vertical dimension represents frequency in hertz. The time-frequency area is divided in cells. The strength of a frequency in a certain cell is indicated by its blackness. Black cells have a strong frequency presence while white cells have very weak presence. The maximum and minimum frequencies represented in the spectrogram are displayed on the left. here they are 0 Hz and 6000 Hz, respectively. The characteristics of the spectrogram can be modified with options from the Spectrum menu. In chapter 8 you will find more information on the spectrogram.
- The pitch is drawn as a blue dotted line in the spectrogram area. The minimum and maximum pitch are drawn on the right side in blue color. Here they are 70 Hz and 300 Hz, respectively. The specifics of the pitch analysis can be varied with options from the Pitch menu. More on pitch analysis in chapter 5.
- The intensity is drawn with a solid yellow line in the spectrogram area too. The peakedness of the intensity curve are influenced by the “Pitch settings...” from the Pitch menu, however. The Intensity menu settings only have influences on the display of the intensity, not on its measurements. The minimum and maximum values of the scale are in dB’s and shown with a green color on the right side inside the display area (normally the location depends on whether the pitch or the spectrogram are present too). More on intensity in chapter 6.
- Formant frequency values are displayed with red dots in the spectrogram area. The Formant menu has options about how to measure the formants and about how many formants to display. Formant analysis is treated in chapter 13.
- Finally the pulses menu enable the pitch glottal pulse moments to be displayed.

More on pulses in the chapter on pitch.

3. The sound display area. It shows the sample "amplitudes" as a function of time. The two numbers at the left near the top and the bottom line of the sound display area show the maximum and the minimum sound amplitudes, respectively.

4. Rows of rectangles that represent parts of the sound with their durations. In the figure only five rectangles are displayed and each one when clicked plays a different part of the sound. For example, the rectangle at the top left with the number "0.607803" displayed in it plays the first 0.607803 s of the sound, left from the cursor. The rectangle in the second row on the right plays the 4.545266 s of the sound that are not displayed in the window. A maximum of eight rectangles are displayed when the part of the sound displayed in the window is somewhere in between the start and end and at the same time we have a selection present. In case of a selection which is indicated by a pink part in the display area, a rectangle also appears above the sound display area. In this rectangle the duration and the inverse duration are displayed. This is shown in figure 2.8 where a the duration "0.1774555" with its inverse "5.729 / s" is displayed. The times of the left and right cursor of the selection are also shown on top. The selection starts at time 0.982446 which is of course equal to the sum of the durations in the two rectangles below the display with the values 0.111519 and 0.870928 as it should be.

5. Global selections. After clicking the "all" button the complete signal is displayed in the display area. The "in", "out" or "back" button zoom in or out or back. By successively clicking "in" you will see ever more detail of the sound. This goes on and on until the duration of the display area has become smaller than the sample time. In this case there will be no signal drawn anymore. By successively clicking "out" you will less detail. This goes on until the complete sound can be drawn within the display area. Further zooming out is not possible.

6. A cursor. The cursor position is indicated by a vertical red-colored dashed line. Its time position is written above the display area.

7. A horizontal scroll bar. We can either drag the scroll bar or click in the part not covered by the scroll bar to change the part of the sound that is displayed.

8. A grouping feature. Grouping only applies if you have more than one time-based editor open on the same sound. If grouping is on, as is show in both figures, all time actions act synchronously in all grouped windows. For example, if you make a selection in one of the grouped editors the selection is replicated in all the editors. If you scroll in one of them all other editors scroll along.

9. A number displaying the average value of the sound part in this window. For this speech fragment the value happens to be "0.007012". In general this number has to be close to zero, as it is here, otherwise there is an offset in your signal. Any offset can be removed with the "Modify>Subtract mean" command.

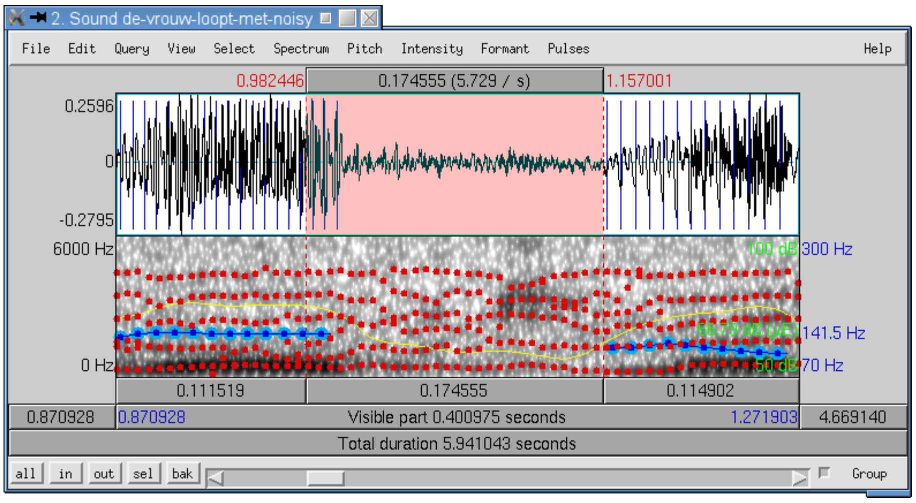


Figure 2.8.: The sound editor with display of spectrogram, pitch, intensity, formants and pulses.

2.4. How do we analyse a sound?

First we have to deal with a fundamental limitation of our analysis methods: most of our analysis methods are not designed to analyse sounds whose characteristics are changing in time, although some special methods have been developed that can analyse signals that change in a *predictable* way. Speech however is not a predictable signal. We have a dilemma here. The practical solution is to model the speech signal as a *slowly varying* function of time. By slowly varying we mean that during intervals of say 5 to 25 ms the speech characteristics hopefully don't change too much and we consider them to be almost constant. The parts of the speech signal where this model holds best is for monophthong vowels. This is because the articulators do not move as fast here as in other parts of speech (for example during the release of a plosive the articulators move faster). Therefore, to analyse a speech sound we cut up the sound into small segments and analyse each interval separately and pretend it has constant characteristics. The analysis of a whole sound is split up and becomes a series of analyses on successive intervals. Such intervals are termed *analysis interval*s and all sound analyses give you the opportunity to set the duration of the analysis intervals. The optimal analysis interval length will depend on what kind of information you want to extract from the speech signal. Some analysis methods are more sensitive to sound changes during the analysis interval than others. As you may know by looking carefully at the speech signal, intervals where the sound signal characteristics do not vary at all, probably may not exist in real speech. Therefore the analysis results always represent some kind of average of the analysis interval.

In the analysis we do not really cut up the signal into separate consecutive pieces but we use overlapping segments, i.e. successive analysis segments have signal parts in common: the last part of a segment overlaps the begin part of the next segment. In figure 2.9 we present the generic analysis scheme. In the upper left part, the figure shows (the first part of) a sound signal and in the upper right part the first part of the analysis results. The next lines visualize

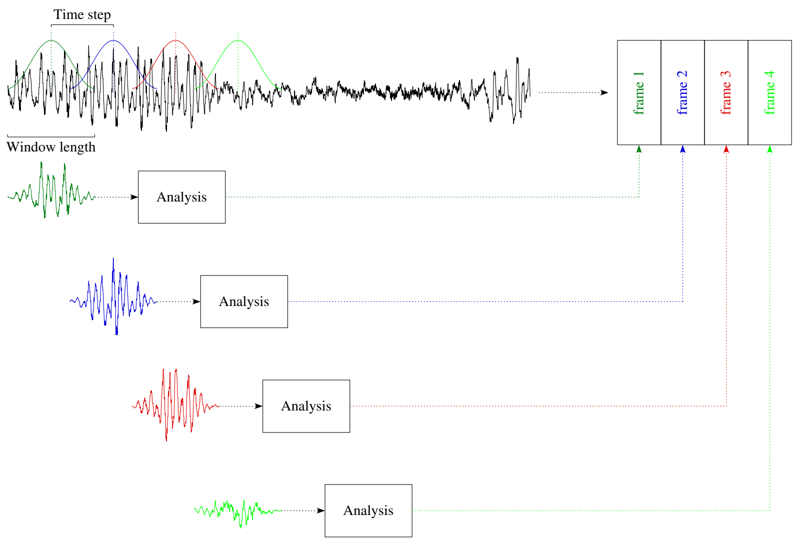


Figure 2.9.: General analysis scheme.

the cutting up of the sound into successive segments. Each segment is analysed in the rectangular block labeled "Analysis", the results of the analysis are stored in an *analysis frame* and the analysis frame is stored in the output object. An analysis frame is also called a *feature vector* in the literature. What happens in the "Analysis" block depends of course on the particular type of analysis and necessarily the contents of the analysis frames also depends on the analysis. For example, a pitch analysis will store pitch candidates and a formant analysis will store formants. Before the analysis can start at least the following three parameters have to be specified:

1. The "window length". As was said before, the signal is cut up in small segments that will be individually analysed. The window length is the duration of such a segment. This duration will hold for all the segments in the sound. In many analyses "Window length" is one of the parameters of the form that appears if you click on the specific analysis. For example if you have selected a sound and click on the "`To Formant (burg)...`" action a form appears where you can chose the window length (see figure 13.6). Sometimes you don't have to give the window length explicitly, instead it can be derived from other information that you need to supply. For pitch measurements window length is derived from the lowest pitch you are interested in.
   There is no one optimal window length that fits all circumstances as it depends on the type of analysis and the type of signal being analysed. For example, to make spectrograms one often choses either 5 ms for a wideband spectrogram or 40 ms for smallband spectrogram. For pitch analysis one often choses a lowest frequency of 75 Hz and this implies a window length of 40 ms. If you want to measure lower pitches the window

length has to increase.

2. The “time step”. This parameter determines the amount of overlap between successive segments. If the time step is much smaller than the window length we have much overlap. If time step is larger than the window length we have no overlap at all. In general we like to have at least 50% overlap between two succeeding frames and we will chose a time step smaller than half the window length.

3. The “window shape”. This parameter determines how a segment will be cut from the sound. In general we want the sound segment’s amplitudes to start and end smoothly. A *windowing function* can do this for us. Translated to the analog domain a windowing function is like a fade-in followed immediately by a fade-out. In the digital domain things are much simpler: we copy a segment and multiply all the sample values by the corresponding value of the window function. The window values are normally near zero at the borders and near one in the middle. In figure 2.9 the window function is drawn in the sound. The form of the window function will not change during the analysis. The coloured sound segments are the result of the windowing of the sound, i.e. the multiplication of the sound with the window of the same color. A lot of different window shapes were popular in speech analysis, we name a square window (or rectangular window), a Hamming window, a Hanning window, a Bartlett window. In Praat the default windowing function is the Gaussian window. When we discuss the spectrogram in chapter 8 we will learn more about windowing functions.

Now the analysis algorithm scheme can be summarized as follows:

1. Determine the values of the three parameters `windowLength`, `timeStep` and `windowShape` from values specified by the user. The duration of the sound and the values of these three parameters together determine how many times we have to perform the analysis on separate segments. This number therefore equals the number of times we have to store an analysis frame, we call it `numberOfFrames`.

2. Calculate the midpoint t_0 of the first window.

3. Copy the windowed sound segment, centred at t_0 with duration `windowLength` from the sound and analyse it.

4. Store the analysis frame in the output object.

5. If this is the last frame then stop. Else calculate a new $t_0 = t_0 + \text{timeStep}$ and start again at 3.

Once the analysis has finished, a new object will appear in the list of objects, for example a Pitch or a Formant or a Spectrogram.

2.5. How to make sure a sound is played correctly?

Whenever you want to play a sound there are a number of things that you need to know. First of all: for the sound to play correctly it is *mandatory* that the amplitudes of the sound always stay within the limits -1 and $+1$. You can easily check this in the sound editor by looking at the numbers displayed at the left of the sound display area near the top and the bottom line as they represent the sound amplitude extrema. If these numbers do not exceed $+1$ or -1 this will guarantee that the sound will not be corrupted during the transformation from its sampled representation to an analog sound signal. In section 3.6.2 a more extensive explanation will be given about what might happen if a sound doesn't conform to this amplitude range. If the amplitude happens to be larger than one, a save way to guarantee that all amplitudes of a sound will stay within a given range is to use the "Modify>Scale peak... 0.99" command. This commands multiplies all amplitudes with the same factor to ensure that the maximum amplitude does not exceed 0.99.

In the second place, fast and large amplitude variations in the sampled sound have to be avoided because these will be audible as clicks during play back and these will create all kinds of perceptual artefact's. In section 7.3 some spectral effect of discontinuities will be explained.

There are several things you can do to avoid clicks.

- When you have to listen to parts of a mono speech sound in the sound editor, you can set the start time and the end time of the selected part on zero-crossings. A zero-crossing is a time point in the sound where the amplitude on one side is above zero and on the other side below zero. At the zero-crossing itself point the amplitude is zero. The best strategy is to let Praat find the nearest zero-crossings at the start and the end time of the selected part. If you don't set your start and end markers on zero-crossings you will probably hear clicks.

- Use fade-in and fade-out functions to let the sound amplitudes grow or shrink. A fade-in or fade-out duration of 5 milliseconds is very common. In section A.1.5 an example of fading is shown.

2.6. Removing an offset

Sometimes it may happen that due to errors in the recording equipment a sound signal shows a nonneglectable offset. By this we mean that the average value of the signal is not close to zero. An artificial sound with offset is shown in figure 2.10 where the artificial signal is drawn with a solid line. The average value of this signal is clearly not around zero but around 0.4, i.e. this signal has an offset of 0.4. We want to remove this offset and have a signal where, on the average, the parts of a signal above and below zero are approximately equal. This can be arranged by subtracting the average value, i.e. the mean, from the signal. This is exactly what the Praat command "Modify > Subtract mean" does. The signal drawn with the dotted line is the result of applying the "Subtract mean" command to the sound drawn with the solid line. For a mono signal it is equivalent to the following script[8]:

[8]We anticipate here on section 4.5.3 on scripting. There is some special syntax to execute Praat commands from within a script. The first line of this script queries a selected sound object for its mean value and assigns it to a variable with the name "mean". In the next line we subtract the mean from all samples of the sound. The "Get mean..." command in this script is the same command that you can find in the sound's dynamic menu "Query -" list. Because the "Get mean..." command needs three arguments we have to supply these on the script line.

```
mean = Get mean: "All", 0, 0
Formula: "self - mean"
```

For a stereo sound means are determined for each channel separately.

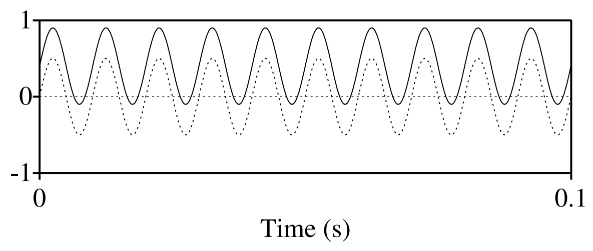
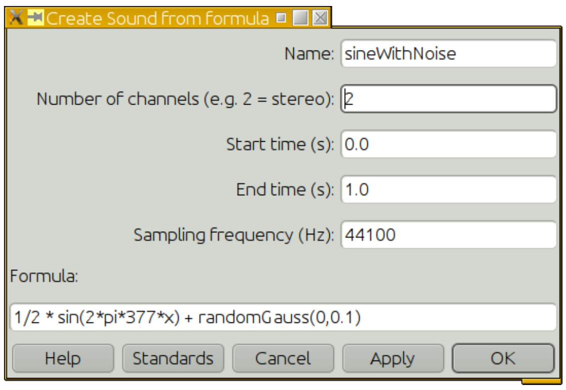


Figure 2.10.: A sound with an offset of 0.4 before (solid line) and after (dotted line) a correction with "Subtract mean".

2.7. Special sound signals

The "Create Sound from formula..." command as displayed in figure 2.11, offers the possibility to create all kinds of sounds by varying the formulas in the Formula field. In the following sections we will give some examples of sounds that can be generated with a couple of lines of script. The **Name** argument in the form specifies the name the new sound object will have



Name: sineWithNoise
Number of channels (e.g. 2 = stereo): 2
Start time (s): 0.0
End time (s): 1.0
Sampling frequency (Hz): 44100
Formula:
1/2 * sin(2*pi*377*x) + randomGauss(0,0.1)

Figure 2.11.: The Create Sound from formula... command form.

in the list of objects. The **Number of channels** argument specifies the number of channels of the sound. This argument may be a any natural number like 1 or 2 or 3. You may also specify either "Mono" or "Stereo" which will be translated to the numbers 1 or 2, respectively. The **Start time** and **End time** specify the domain of the sound. Most of the time you will specify a duration by only supplying the **End time**. The **Sampling frequency** specifies the number of samples the sound will reserve for every second of its duration. For example, a mono sound that lasts 2 s and has a sampling frequency of 44100 Hz will consist of 88200 samples. Finally, the **Formula** argument specifies the amplitudes of the sound. Here you can use all the formulas that are available in Praat. The example form creates a tone of amplitude 0.5 with a frequency of 377 Hz with random Gaussian noise of zero mean and 0.1 standard deviation. The resulting sound is calculated by evaluation of this formula for each sample value (where for the variable x the time at the corresponding sample value is substituted). More details of how a Formula works on a sound will be given in section 4.7.1.2.

2.7.1. Creating tones

The following line creates a mono sound with a tone of 800 Hz with a sampling frequency of 44100 Hz and a duration of 0.5 s.

```
Create Sound from formula: "s", 1, 0, 0.5, 44100, "0.9*sin(2*pi*800*x)"
```

This sound is displayed in the following figure 2.12 with a dotted line.

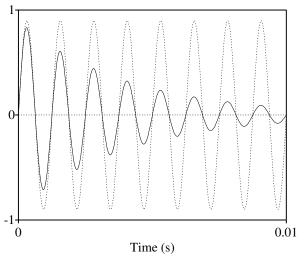


Figure 2.12.: The solid line shows a damped sine with a frequency of 800 Hz and a bandwidth of 80 Hz. The dotted line shows the (undamped) tone of 800 Hz.

An alternative way to create a tone is with the specialized command “Create Sound as pure tone...” from the Sound menu. You don’t need a sine formula to specify the tone, you only specify the frequency and the amplitude and, additionally, you can specify fade-in and fade-out times to guarantee that no clicks can be heard at the start and the end of the sound. The following command will create a tone with an amplitude of 0.9 Pa, a duration of 0.5 s and a frequency of 800 Hz.

```
Create Sound as pure tone: "tone", 1, 0, 0.5, 44100, 800.0, 0.9, 0.01, 0.01
```

2.7.2. Creating a damped sine (formant)

A damped sine can be created from the formula $s(t) = e^{-\alpha t} \sin(2\pi F t)$, where F is called the frequency and α is a positive number called the *damping constant*. As one can see, damping reduces the amplitude of $s(t)$ as time t increases. If α is very small there is hardly any damping at all and if α is large the amplitude goes to zero very fast. The damping constant α is often expressed as $\alpha = \pi B$, where B is called the *bandwidth*. The damped sine can then be written as $s(t) = e^{-\pi B t} \sin(2\pi F t)$. The following script creates a damped sine with a frequency of 800 Hz and a bandwidth of 80 Hz. In figure 2.12 we show the damped sine with a solid line. As a comparison the undamped tone of 800 Hz is shown with a dotted line. A damped sine is also called a *formant* and it is one of the building blocks of speech synthesis. We will learn more about formants in chapters 12 and 13.

```
f = 800
b = 80
Create Sound from formula: "f", 1, 0, 0.1, 44100, "exp(-pi*b*x)*sin(2*pi*f*x)"
```

2.7.3. Creating noise signals

In many experiments noise sounds are required. Noise sounds can be made by generating a sequence of random amplitude values. Different kinds of noises exist, the two most important ones are called *white* noise and *pink* noise. In white noise the power spectrum of the noise is flat on a linear frequency scale: all frequencies have approximately the same strength. In pink noise the power spectrum is flat on a logarithmic frequency scale. This means that on a linear frequency scale the power varies as a $1/f$ function (see section A.7). Both types of noise can be made easily with Praat.

To create white noise we can use two functions that generate random numbers from a random number distribution.

- The function `randomGauss (mu,sigma)` generates random numbers from a Gaussian or normal distribution with mean `mu` and standard deviation `sigma`.

  ```
  Create Sound from formula: "g", 1, 0, 0.5, 44100, "randomGauss(0,0.2)"
  ```

  A mono sound labeled g will appear in the list of objects. It's duration will be 0.5 s and it is filled with random Gaussian noise of zero mean and 0.2 standard deviation.

- The function `randomUniform(lower,upper)` generates random numbers between `lower` and `upper`. It has the advantage as compared with the randomGauss function that all amplitudes are always limited to lie within the predefined interval.

  ```
  Create Sound from formula: "u", 1, 0, 0.5, 44100, "randomUniform(-0.99,0.99)"
  ```

  A mono sound labeled u wil appear in the list of objects. It's duration will be 0.5 s and it is filled with uniform random noise. All amplitudes will be smaller than or equal to 0.99.

In the spectra of both types of noise all frequencies are equally present, and no audible spectral differences can be heard between the two sounds. For practical use we often prefer the randomUniform noise because we have better sound amplitude control since for random Gaussian noise some extreme amplitudes outside the (-1,1) interval might always occur. In section 4.7.1.3 we will learn how to create pink noise.

2.7.4. Creating linear sweep tones

Up till now we have only created tones with constant frequencies. Suppose we want a tone whose frequency changes linearly from a frequency f_1 at time t_1 to a frequency f_2 at time t_2. We can write such a function as $sin(\phi(t))$, where the *phase function* $\phi(t)$ should regulate the frequency at any time t. How should this phase function $\phi(t)$ look like? We already know that for a tone with a *constant* frequency F it can be written as $\phi(t) = 2\pi Ft$ because $\sin(2\pi Ft)$ is the formula for such a tone. Here the frequency does not depend on time. Now, a frequency f that *increases or decreases linearly* as a function of time can in general be written as $f(t) = at + b$, where the coefficients a determines the slope and b the offset. With the boundary conditions, i.e. the start and end frequencies we determine these coefficients as $a = \frac{f_2 - f_1}{t_2 - t_1}$ and $b = f_1 - at_1$. It can be shown[9] that the corresponding phase $\phi(t)$ for this case is $\phi(t) = \pi at^2 + 2\pi bt + \phi_0$, where ϕ_0 is the phase at time $t = 0$. The following script creates a sweep tone of 1 s duration that starts at 500 Hz and ends at 1500 Hz.

```
f1 = 500
f2 = 1500
t1 = 0
t2 = 1
a = (f2 - f1) / (t2 -t1)
b = f1 - a * t1
Create Sound from formula: "sweep", 1, t1, t2, 44100,
... "0.99*sin(pi*a*x^2 + 2*pi*b*x)"
```

2.7.5. Creating a gammatone

A gammatone is a sound that can be described as the product of a gamma distribution with a sinusoidal tone. It is the impulse response of the gammatone filter. It is no problem if you don't know what a gamma distribution is, just continue. The gammatone is important because it is often used as a model of the auditory filter. We create a gammatone as follows:

```
f = 500
bp = 150
gamma = 4
phase = 0
Create Sound from formula: "gammatone", 1, 0, 1, 44100,
  ... "x^(gamma - 1) * exp(-2 * pi * bp * x) * sin(2 * pi * f * x + phase)"
Scale peak: 0.99
```

In left panel of figure 2.13 we show the gammatone that results from the above script. The formula for a gammatone is $g(t) = t^{\gamma-1}e^{-2\pi bt}\sin(2\pi Ft + \phi_0)$, where γ determines the order of the polynomial part of the gamma distribution. The parameters F and b are the carrier frequency and the "bandwidth" parameters and ϕ_0 is the starting phase. The figure shows that at the start of the tone, the polynomial rising part of the gammatone function is stronger than the exponentially decaying part but eventually the exponential decay takes over and makes the amplitude vanish. Compared to the formant of section 2.7.2 we note that the gammatone

[9] For a function $\sin(\phi(t))$ the instantaneous frequency $f(t)$ is defined as $f(t) = \frac{1}{2\pi}\frac{d\phi(t)}{dt}$. From this we deduce that the phase $\phi(t) = \int f(t)dt$. Given a linear function for the frequency $f(t) = at + b$, the phase then follows as $\phi(t) = 2\pi({}^1\!/\!{}_2at^2 + bt + c)$, where the integration constant c can be adapted to account for the phase at $t = 0$.

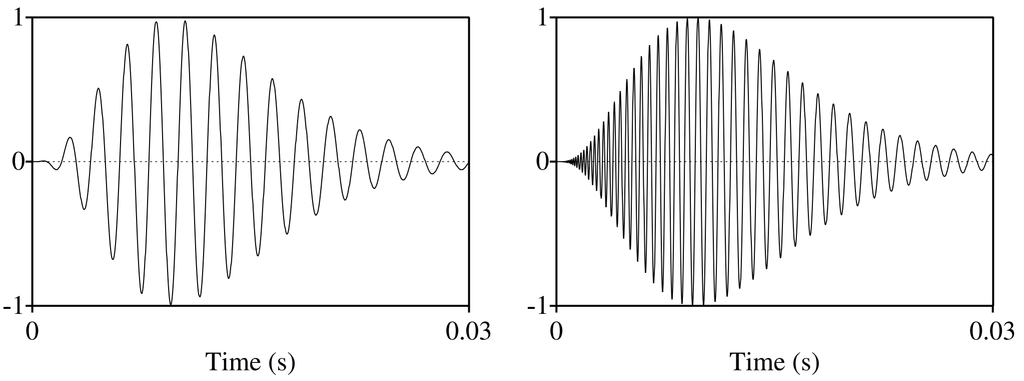


Figure 2.13.: On the left a gammatone with $\gamma = 4$, $F = 500\,\text{Hz}$ and $b = 50\,\text{Hz}$. On the right a gammachirp with $\gamma = 4$, $F = 500\,\text{Hz}$, $b = 50\,\text{Hz}$ and an addition factor $c = 50$.

is like a formant with the starting part modified by the rising polynomial. The parameter b results in an actual bandwidth that is twice this value, i.e. $B = 2b$. Besides auditory modeling, many other physical phenomena, like for example knocking on wood, can be modeled with gammatones.

2.7.6. Creating a gammachirp

The gammachirp introduces a frequency modulation term to the gammatone auditory filter to produce a filter with an asymmetric amplitude spectrum [Irino and Patterson, 1997]. It can be used in an asymmetric, level-dependent auditory filterbank in time-domain models of auditory processing [Irino and Patterson, 1997]. The formula for a gammachirp is $g_c(t) = t^{\gamma-1}e^{-2\pi bt}\cos(2\pi Ft + c\ln(t) + \phi_0)$ which shows that for $c = 0$ it reduces to the ordinary gammatone. From the gammachirp's formula it follows that its frequency varies as a function of time like $f(t) = F + \frac{c}{2\pi t}$. So, theoretically, the frequency at the start of the sound ($t = 0$) is infinite and subsequenty at a $1/t$ rate approaches the frequency F. In contrast with a linear sweep tone the frequency change is not constant in time but is largest in the first part of the sound. In order to avoid aliasing, the first part of the gammachirp, where the frequency is larger than the Nyquist frequency has to be suppressed.

```
f = 500
bp = 50
gamma = 4
c = 50
phase = 0
Create Sound from formula: "gammachirp", 1, 0, 0.1, 44100,
  ... "if (f + 1/(2 * pi * x)) < 22050
  ... then x^(gamma - 1) * exp(-2 * pi * bp * x) * cos(2 * pi * f * x + c * ln(x) + phase)
  ... else 0 fi"
Scale peak: 0.99
```

The chirp generated with the script sounds like the tweet of a bird. Gammachirps and gammatones are so important that they deserve a special command to generate them: the New > Sound > Create Sound from gammatone... command can generate both gammatones as well as gammachirps.

2.7.7. Ceating a sound with only one pulse

A sound with only one pulse can be used to study the impuls response of a digital filter. To create such a sound where for example the first sample value is one and the rest are all zeros, the following script suffices:

```
Create Sound from formula: "1", 1, 0, 0.1, 44100, "if col=1 then 1 else 0 fi"
```

This script line can easily be modified to allow the 1-puls at any sample number.

If we want a pulse at a time that not exactly matches a sample time, things get a bit more complicated. Because the time is not at a sample point, a sound like this is not really band-limited, i.e. its spectrum will have frequencies above the Nyquist frequency. To correctly represent such a sound with a certain sampling frequency, we first have to band-limit it by low-pass filtering. Praat takes care of this if we start by creating a PointProcess object[10], then add the point at the desired time (s) to the PointProcess and then create the final sound from this PointProcess. The final part, the creation of a sound from a point process takes care of the necessary filtering.

```
Create empty PointProcess: "pp", 0, 0.1
Add point: 0.01234567
To Sound (pulse train): 44100, 1, 0.05, 2000
```

Figure 2.14 shows the low-pass filtered pulse as (part of) the sound generated by the script above.

[10]See chapter 15 for more information about a point process.

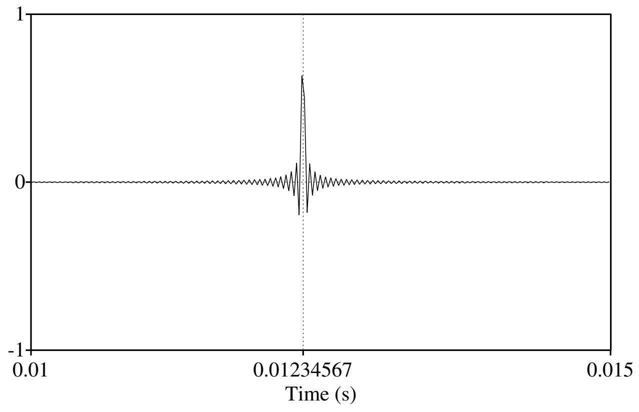


Figure 2.14.: A pulse at time 0.01234567 represented in a sound of 44100 Hz sampling frequency.

3. Sound and the computer

This chapter is about getting sound into the computer. It is *not* about downloading a sound file from the Internet. It is about the process described in figure (3.1).

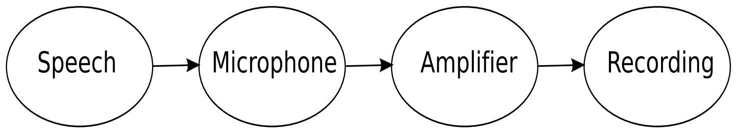


Figure 3.1.: The sound recording process.

A speaker produces a speech signal, the acoustic wave is picked up by a microphone and transduced into an electrical signal. This electrical signal is amplified and the amplified signal is recorded either in analog or in digital form. Nowadays the recording will be in digital form only.

The conditions at a recording session can vary widely, to name some:

- We are recording at home with a microphone directly attached to the computer. This is often the most simple environment for making a recording. All we need is a (good) microphone and a computer with a sound card. However, we don't control the environment, which means that noises could start at any time. For example, somebody enters our room, a car honks outside or one of the neighbors starts drilling a hole. Another source of noise is the computer itself: a fan might start to blow or a disk might spin up or the electrical circuit of the computer interferes with the circuitry of the input of the sound card.
- We are recording in a studio. This is clearly the preferred way because we have almost complete control over the recording environment. Environmental noise, the main disturbing factor in making a recording, can be minimized here. We are also not limited by the complexity, weight or size of all the necessary recording equipment.
- We are recording in the field. This might be at extreme places like somewhere in the Amazon basin where you have to record the sounds of a very uncommon Indian language, or in the tundra of Siberia where the last sounds of some extinguishing language have to be recorded or in a Chinese city making recordings for a language change project. These kind of field recordings clearly limit the amount of control that we can exercise on the recording environment and they also limit the size and weight of our recording equipment.

In all these different recording situations, the main goal is to store the volatile analog information of the sound pressure wave into an unchangeable digital format that can be played back at later times. This is accomplished by first transducing the information in the analog sound pressure wave to an electrical signal that can be processed more easily. In former times this signal was stored, after preparation, on a magnetic tape or on a record or even on wax-roles.[1] Nowadays everything is stored in a digital memory device like a USB stick, a computer disk, a compact disc, DVD or on flash memory cards, to name the most current digital audio storage media.

Before we reach the digital memory stage, some processing is required. We will examine these processing steps in the sequel when we discuss the electronic equipment that we need to make a sound recording.

3.1. Making a recording at the computer

To make a recording at the computer, we need a computer with a sound card and a microphone. We connect the microphone with the sound card in the computer. Before we start recording, we must be sure that the microphone is selected as the only sound input device and any other input devices like the CD or line-in are deselected. The input mixer takes care of this (see section 3.6.3): select the microphone and deselect all other devices as is shown in figure 3.8 for the Windows XP mixer. While making a recording it is best to also deselect *all* outputs to avoid possible reverberations between the loudspeaker in the computer and the microphone.

Now with the sound streams in place, we use Praat for the actual recording. We choose from the “New” menu the “Record mono Sound...” option: the sound recorder pops up, it may look like figure 3.2 shows. The mono channel is automatically set and you check “Microphone” and the “44100 Hz” boxes. The actual recording starts as soon as the “Record” button is clicked and stops when you click “Stop”. If you click “Save to list” a new Sound object appears in the list of objects. You can inspect the new sound in the sound editor and save it to a file on disk. If you close the sound recorder without having saved the sound the recording will be lost.

3.2. Making a recording in the studio

In general a studio is optimal for making a recording. In a well designed studio all kinds of disturbing sounds have been eliminated or are at a minimal level. If possible, making recordings in a studio is to be preferred above anything else.

[1]To digitize a collection of wax-rolls one needs specialized equipment. The old machines to play these rolls are hardly available these days. And even if they were, one runs the risk of damaging the old rolls because physical contact between a needle and the roll is needed. Preferably one uses the reflections of a laser beam to read out this old material.

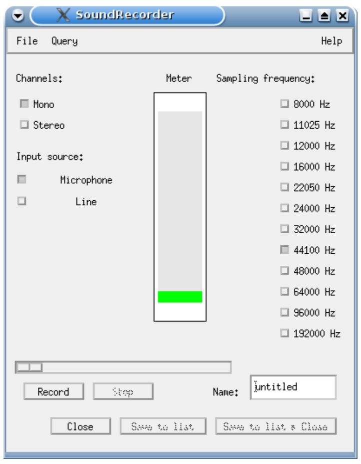


Figure 3.2.: The sound recorder.

3.3. Making a recording in the field

3.4. Guidelines for recording speech

- Try to avoid making recordings in reverberant rooms (a church is very reverberant).
- Try to avoid making recordings at places where uncontrollable noises like traffic noise, airplanes flying over, trains passing by, other people interrupting etc. might turn up.
- Write the recording protocol beforehand and inform the speaker about how and what you want from her.
- Arrange a comfortable reading/speaking position for the speaker. If she needs to read from paper, take care that no paper cracklings take place while reading and the speaker only moves the paper during breaks. These sounds get unnoticed during the recording but afterwards, with you carefully listening with a headphone, they will sound awfully loud.
- To avoid large intensity variations in the recording, the distance from the speaker's mouth to the microphone should remain as constant as possible. Use head-mounted microphones whenever you can.
- Beware of lip noises: many speakers get dry lips during talking, their lips start sticking together and then they open with a smacking sound that you probably won't notice during the recording, only afterwards when you listen with headphones. Try to avoid this from happening by having a glass of water standing by.
- Take care to comfort the speaker, she should speak as natural as possible. If you record read-aloud speech then embed the words or sentences you want recorded in other

words and sentences. Try to avoid repeating rhythmic patterns in consecutive sentences: speakers will adapt their speaking rate to this rhythm.

- Test extensively the sound level before making the recording. Try to avoid that your recordings are either too loud or too soft.
- Always start with the speaker saying her name and date and time of the recording.
- Legal concerns. Depending on the use of the recorded material the speaker has to give her written consent. This means that you have to state explicitly how the speech is going to be used and/or stored or distributed. If you want to share, distribute, or publish the recordings, you need a copyright licence signed by the speaker. You can use a licence as used by the big corpus projects. When recording conversations be aware of possible privacy concerns.
- Do not use audio compression like MP3, as it introduces artifacts that can show up in analysis results.
- .... and more.

3.5. Audio file formats

Audio file formats describe the structure of a sound on a storage medium. The sound file itself is just a bunch of bytes on the computer disk. The file format gives information for a computer program how to interpret these bytes. For example, the program needs to find out whether the file contains a mono or a stereo sound. The program also needs to know at what sampling frequency the sound was recorded. The following questions have to be clarified too. What is the precision of the audio data in the file? Are they stored with 8, 12, 16 or perhaps 24 bits of precision? Are multi-byte entities stored in big-endian or little-endian format?[2] Is the sound data directly readable or is it only available in a compressed form and if so, what compression algorithm has been used? The answers to these kind of questions have to be added as extra information to the sound signal and should preferably be stored together with the sound within one file container in an unambiguous format, otherwise we do not know how to process the data in the file.

Because there are many ways to store these *meta* data in a file, a plethora of audio file formats exists today. We will give a short overview of the most important audio file formats that Praat can handle.

3.5.1. The aiff and aifc format

AIFF stands for Audio Interchange File Format. The sound is stored uncompressed in big-endian format. It is the native audio sound file format of Apple Macintosh computers. The AIFF-C or AIFC format has the same structure as the AIFF format but *may* store the sound data in a compressed form. Because the concept of these file types is so elegant, we will

[2]See section D on terminology.

give a short description here. For an extended overview we refer to the wikipedia article and its references. AIFF and AIFC both are container formats, this means that the various type of information are not intermingled but can be found in separate chunks in the file. A chunck always starts with an identification string of four ASCII characters, followed by a four byte (unsigned) long number, the chunk size, that gives the total number of bytes of storage needed for the rest of the chunk. This number has been incorporated to be able to read (or to skip) a chunk. An AIFF file is also one big container chunk, it starts with 'FORM' followed by the total number of bytes that remain in the file. Then follows the 'AIFF' identifier which identifies it as an AIFF file. An AIFF file contains at least two different chunks the 'COMM' chunk and the 'SSND' chunk. The 'COMM' chunk contains information *about* the sound like the number of channels (1: mono, 2: stereo, etc), the number of samples per channel, the number of bits per sample and the sampling frequency. The 'SSND' chunk contains the sample values of the sound. For AIFC files the identifying chunk is 'AIFC' instead of 'AIFF'. For example if we dump the first part of an AIFF file as ASCII then we might get a string like "FORM....AIFFFVER........COMM......................SSND....", where all bytes that are not of character type have been replaced by a dot. We see that the file starts with 'FORM', followed by four bytes that make up the chunk size, next follows identification of this file as 'AIFF'. Next there is also an 'FVER' chunk, we deduce from the string that it is a very short chunk with only 4 bytes of information besides it size. The 'COMM' chunk has 18 bytes for its information. Next comes the 'SSND' chunk where only four size bytes are indicated. This chunk contains all the sample values. The interpretation of its contents depends on the information in the 'COMM' chunk.

The actual speech data in a file can be read by Praat only if the data is uncompressed and in PCM format, i.e. it sample values are stored and not some derivative of these values like DPCM. For most of the available files in AIFF/C format this is luckily the case. This means, however, that it might occasionaly happen that Praat cannot *read* a file.[3] The files written by Praat in this format always contain 16-bit PCM values.

3.5.2. The wav format

The wav format is the native audio file format for the Microsoft Windows operating system. Its file structure is derived from and very similar to the container format of AIFF, however, it stores the data in little-endian format. One weak spot of this format is that sampling frequency has to be an integer number wich poses problems in the conversion of some older formats with non-integer sampling frequencies.

3.5.3. The FLAC format

The Free Lossless Audio Codec is an open source lossless audio codec. A codec is a piece of software that can code and/or decode a piece of data. It supports streaming, seeking and archiving. The compression is approximately 50%.

[3]The cross-platform Sound exchange program program SoX opens and saves audio files in most popular formats and can optionally apply effects to them; it can combine multiple input sources, synthesise audio, and, on many systems, act as a general purpose audio player or a multi-track audio recorder. It also has limited ability to split the input in to multiple output files.

3.5.4. Alaw format

This is a sound compression format used in European telephony. The amplitudes in a file are compressed with a logarithmic transform and quantized with 8 bits. This ensures that the lower amplitude signals (where most of the information in speech takes place) get the highest bit resolution while still allowing enough dynamic range to encode high amplitude signals. The sampling frequency is 8 kHz.

3.5.5. μlaw format

This is also an 8-bit format like alaw but for American and Japanese telephony. It uses a similar logarithmic transform for quantization and also 8 kHz sampling frequency.

3.5.6. Raw format

Sometimes sound is available in a format with incomplete data about the sound. Maybe, only the numer of samples are known or can be guessed from its file size. These sound files are called *headerless* files or raw files. Because the information is not *in* the file we have to supply the information *afterwards*. This may end up into a trial and error game. If we know that each speech sample is a one-byte number then we might try the Open > Read from special sound file > Read Sound from raw Alaw file..." command. If this does not help, or, we know that the speech data is in a two-byte format we might try one of the two commands "Read Sound from raw 16-bit Little Endian file..." or "Read Sound from raw 16-bit Big Endian file...". The little endian and big endian terminology is explained in appendix D and refers to the way a number has been stored in the computer. If we have no idea which one to chose then simply try both ways. If you have chosen the wrong endianness and play the sound it will always sound terrible and be completely incomprehensible. The one that sounds best is the one you want. Once you got the endianness right, however, the sound's sampling frequency still might be wrong. If it sounds as if the speaker speaks very slow and with a very low pitch then the sampling frequency is too low and you have to increase it. By increasing the sampling frequency, you speed up the sound, because, since the number of samples in a sound is fixed and you play more of them in each unit of time, the duration of the sound will decrease. This means that by changing the sampling frequency of a sound you implicitly change its duration. Note that we are not recording a sound here, we already have all the sample values and we know how many samples we have, the only thing we change is the sound's playing characteristics. If we know the correct sampling frequency we can use the "Modify > Override sampling frequency..." command to impose the correct frequency on the sound. If we really don't know what the sampling frequency of the raw file was, we have to guess it by setting a frequency and then listen whether it "sounds well". Most of the time raw sound files date from a very long time ago because nobody would nowadays save data in this way. Popular sampling frequencies in these days were 8 kHz, 10 kHz and 16 kHz.

3.5.7. The mp3 format

A lossy file format developed mainly at the Fraunhofer institute. The mp3 file format stores a compressed lossy version of the original. Data-compression rates of 10 to 1 are obtained easily with only small audible artifacts. The mp3 format is not a free standard, there are licences that limit availabilty.

3.5.8. The ogg vorbis format

A better lossy compression format than mp3 and a completely free standard. With a lossy compression format you will never be able to restore the original sound wave while a lossless format allows exact reconstruction. In other words: with a lossy compression format we lose some information about the original signal. It is a pitty that this standard has not enough followers yet.

3.6. Equipment

3.6.1. The microphone

The microphone transduces the acoustical sound pressure waves into an electrical signal. Sound pressure variations cause a mechanical movement of a thin membrane, the *diaphragm*, and this mechanical movement is transduced to an electrical signal. Various type of microphones exist that can be distinguished by the type of physical effect they use to generate the electrical signal.

- Electro-magnetic induction microphones, also called dynamic microphones use the well known fact that if a magnet moves past a wire, the magnet induces current to flow in the wire. There are two basic dynamic microphone types: the moving coil microphone and the ribbon microphone. The moving coil microphone uses the same principle as is used in the loudspeaker but instead of generating a movement of the coil in response to a changing current, it generates a current in response to a moving coil. The ribbon microphone uses a ribbon placed between the poles of a magnet to generate voltages by electro-magnetic induction.

- Piezoelectric microphones. Certain crystals change their electrical properties as they change shape. By attaching the diaphragm to such a crystal, the crystal will create a signal when sound waves hit the diaphragm.

- Carbon microphones. These are the oldest and simplest microphones. They were used in the first telephones and are still in use today. They use carbon powder in a container that has a thin metal or plastic diaphragm on one side. Sound air pressure variations change the degree of compression of the carbon powder and this changes the powder's electrical resistance. By running a current through the carbon, the changing resistance changes the amount of current that flows.

- Condenser, or electrostatic microphones. The diaphragm acts as one plate of a capacitor, and the vibrations produce changes in the distance between the plates. Because the charge on the plates is fixed, the distance change produces a voltage change.
- Laser microphones. When sound pressure waves hit an object like a glass window, the window reacts a little bit on the varying pressure. With the reflection of a laser beam the tiny movements of the glass can be detected.

3.6.2. The sound card

The sound card is the sound-processing unit, the audio work horse of the computer. The card is used, among others, for audio recording and audio generation. It is the interface between the analog world outside the computer and the digital world inside. If you play an audio-CD or a DVD in your computer, the sound of the player is sent in analog or digital form to the sound card where it will be processed further. If you attach a microphone to the computer, the microphone signal is processed by the sound card. If you order Praat to play a sound the digital representation of the sound is sent to the sound card where it will be put in analog form and sent to an external headphone or speaker. In figure 3.3 a typical year 2000 sound card is shown. There is a lot of electronics on this card! In many modern computers the sound processing unit is not on a separate card anymore but is integrated on the motherboard of the computer. However, for a description of the sound processing that goes on in this unit, this sound card gives a better view. This card is inside the computer in a so called PCI-slot. The PCI-slot offers a way for the card to communicate with the CPU of the computer. It is a two-way communication channel. Digital signals can travel to and from the card to the CPU. The metal colored plate on the left connects to the outside of the computer. In this picture you cannot see how it shows on the outside. But we can deduce how it would look. At the top left we see four colored blocks, orange, light blue, pink, lime green. These blocks are connectors with the outside world. The colors have been standardized:

- lime green is for the line-out jacket. It *delivers* an analog sound signal that can be made audible with head phones or a loudspeaker. The maximum amplitude of a line-out signal varies from card to card but approximates 1 volt (V).
- light blue is for the line-in jacket. It *accepts* an analog sound signal from devices like a CD-player, a cassette deck or a tape recorder. The maximum line-in sensitivity is not standardized but normally is around 200 mV. There is controlling circuitry right after the line-in jacket on the sound card to be able to accept a wider range of input amplitudes, sometimes up to 2 V.
- pink/red is for a microphone jacket. The sensitivity of the microphone input is not standardized and may vary between approximately 20 mV and 120 mV, depending on the type of sound card. Sensitivity is defined here as the minimum voltage that produces full scale level at the analog to digital converter if the microphone volume control is set to its maximum (i.e. the slider in the input mixer at maximum). Immediately after the microphone jacket a special microphone pre-amplifier amplifies the microphone signal to bring it on par with the other line level signals. This pre-amplifier is generally not

externally controllable and delivers a fixed amplification like 20 or 30 dB. The actual signal presented at the microphone jacket may have voltages much larger than 20 or 120 mV. For these cases the volume control slider of the microphone can be used to attenuate these signals to stay within bounds acceptable for the ADC. Now the only place where things still might go wrong is in the microphone pre-amplifier if the microphone signal presented at the jacket overloads the pre-amplifier.

- orange is the S/PDIF jacket for digital sound output. Sometimes this jacket is also used as an analog line output for another loudspeaker.

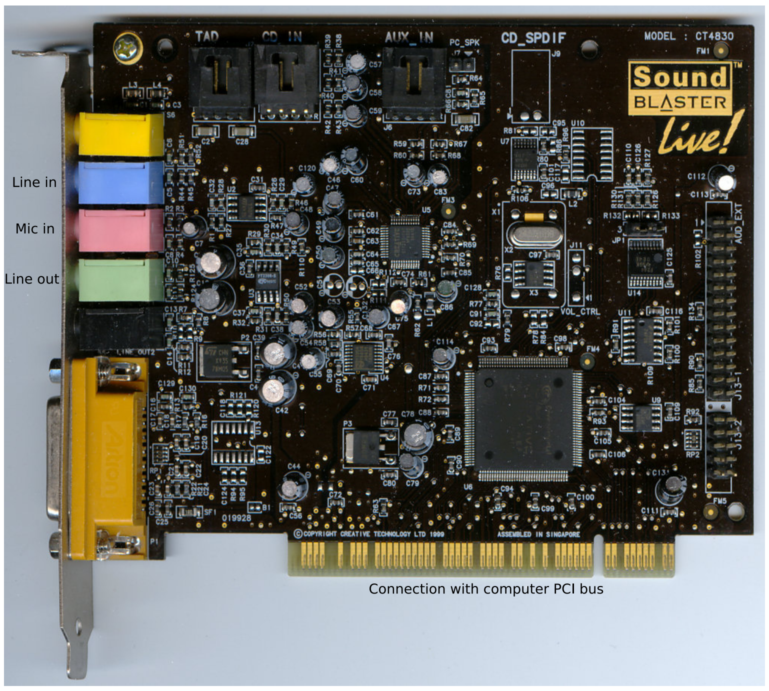


Figure 3.3.: A typical sound card.

In theory, the line-in and line-out are standardized in such a way that one should be able to connect a signal from the line-out of a device to the accepting line-in of the sound card without causing any signal loss or degradation. In practice this is not always the case and one should carefully check whether the connection behaves as it should. Sometimes a kind of adaptor device is necessary in-between. For example if the amplitude of a signal coming out of an external device is too large for the input of the sound card the signal will be clipped. An example is shown in figure 3.4 where the input sound on the left is severely clipped. We clearly see that tops and valleys are flattened as if cut off by a razor blade.

To avoid this clipping taking place, there is some electronic circuitry needed between the line-in jacket and the ADC which has a fixed maximum allowed input amplitude. This additional circuitry, lets call it an amplifier,[4] must be capable of changing the amplitude of the input signal before it reaches the ADC. In the mixer panel you can move the sliders to arrange for this amplitude. However, in most sound cards this electronic circuitry only behaves as desired when its input amplitude, i.e. the signal coming directly from the line-in jacket is not too big. So we are confronted with the bizarre situation that the circuitry that has to guarantee that the ADC cannot be overloaded can be overloaded itself!

Even worse, if this amplifier gets overloaded and you cannot regulate the amplitude of your external device, there is nothing else you can do but connecting some kind of attenuator or amplifier between the output of your external device and the sound card's line-in!

3.6.2.1. Oversteering and clipping (do it yourself)

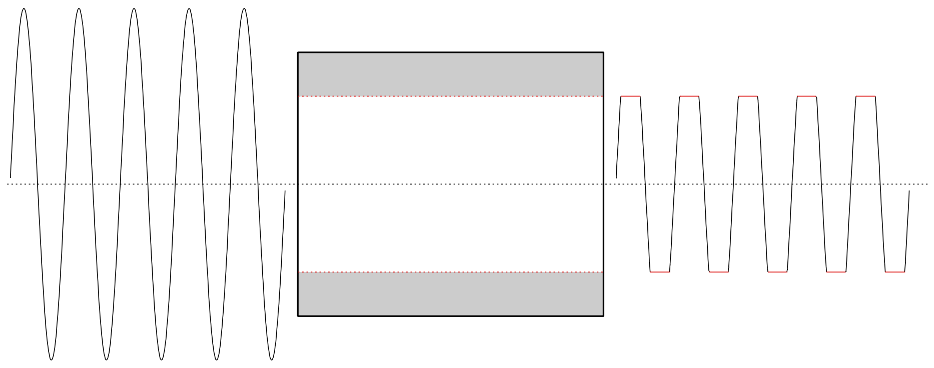


Figure 3.4.: An example of clipping. The signal at the left has too large amplitudes for the electronic device in the middle. The output signal on the right will be clipped (clipped parts shown in red).

If you want to listen to a clipped sound then run the following script in Praat and listen to the two sounds. Both sounds are 500 Hz tones.

```
Create Sound from formula: "s500", 1, 0, 1, 44100, "sin(2*pi*500*x)"
Create Sound from formula: "s500o", 1, 0, 1, 44100, "2*sin(2*pi*500*x)"
```

The sound labeled s500 has an amplitude of 1 while the sound labeled s500o has an amplitude of 2. If you look at both sounds in the sound editor, they show as perfect sines. However, s500 sounds like a perfect 500 Hz tone, while s500o sounds harsh. For the explanation we have to know how a sound is played in Praat, i.e. what happens if you push the Play button when a sound is selected. Praat then translates the selected sound to a series of numbers that are sent to the sound card and then the DAC will use these numbers and convert them to an analog signal that is used to drive the loudspeaker. If the amplitude of the sound varies between -1 and $+1$, the range of numbers that Praat sends to the DAC guarantees that the analog signal of the DAC varies between its minimum and its maximum amplitude. I.e. the analog signal range

[4] The main use of this amplifier is to be able to weaken the signal.

that the DAC produces, is optimally used if the amplitude of a sound is in the $(-1, 1)$ interval. When Praat sends the numbers to the DAC it has to make all sound amplitudes that are larger than 1 equal to 1 and all amplitudes that are smaller than -1 equal to -1. This is clipping and it produces the same effect as if we had generated the sound s500o with the following script.[5]

```
Create Sound from formula: "s500o2", 1, 0, 1, 44100, "2*sin(2*pi*500*x)"
Formula: "if self > 1 then 1 else if self < -1 then -1 else self fi fi"
```

3.6.2.2. Sound card electronic circuitry

As can be seen in the picture 3.3, a sound card contains a lot of electronic circuitry. The really interesting parts of the card, however, are invisible because they are all integrated in specialized chips like the big square chip that rests at the bottom of the card near the PCI connector. In this chip all the digital sound processing takes place.

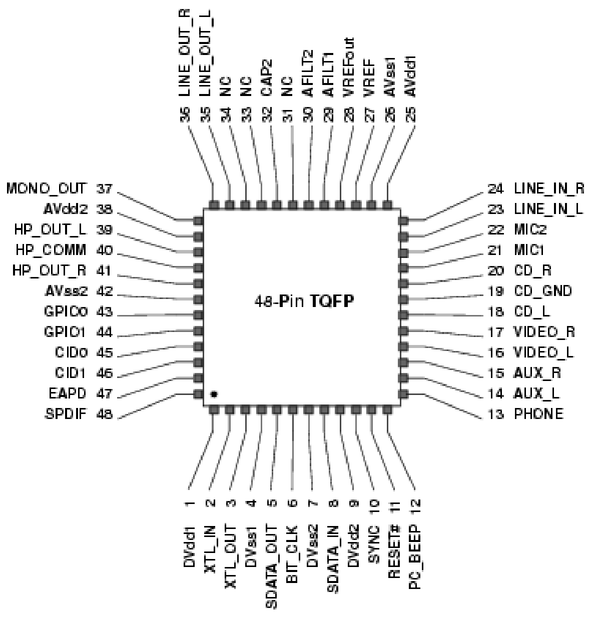


Figure 3.5.: The layout of the 48 pens of the STAC9750 sound processing chip.

As an example of what might take place in such a chip the pin layout of a much simpler one, the mid 1990's STAC9750 is shown in figure 3.5. The pen layout shows what kind of

[5]This is the formula we have used to produce the clipped sound in figure 3.4.

signal has to be connected to which pen of the chip. For example, the two pens on the top side on the left have to be connected with the right and the left channel of the line-out jacket, respectively, while the two top pens at the right side have to be connected with the right and the left channel of the line-in jacket, respectively. Further more, we see pens to connect the signals from the CD player and auxiliary (AUX), and a lot more. This chip integrates all the necessary components, like the mixer, the ADC's, the DAC's as we can see in the chip's block diagram in figure 3.6.

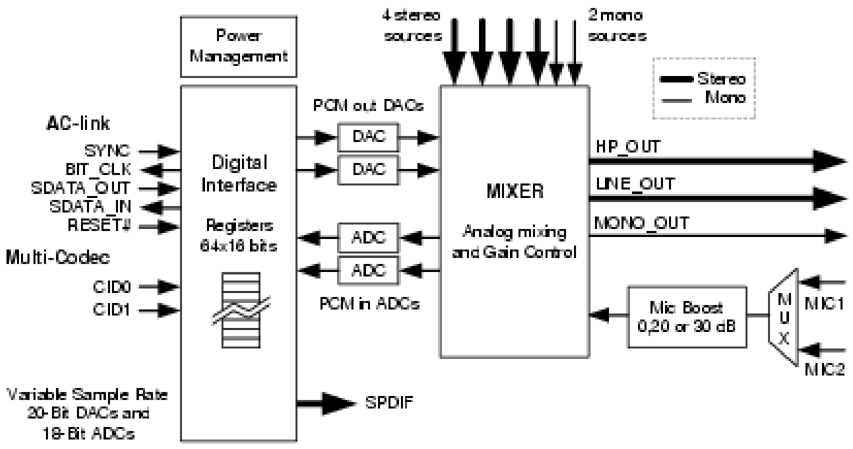


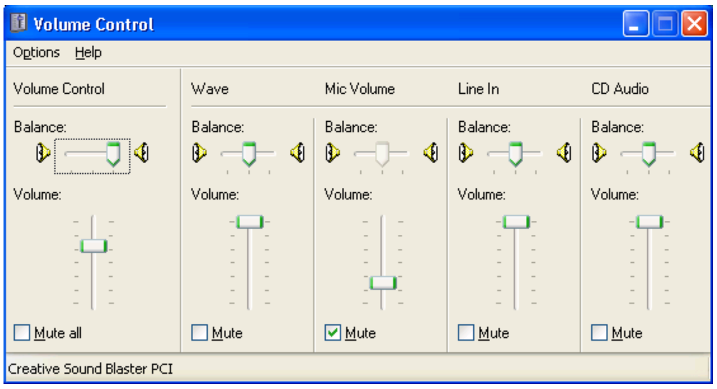
Figure 3.6.: Block diagram of the STAC9750 sound processing chip.

In the diagram we see the different functional units and how they are interconnected. The interface to the computer is displayed on the left while the connections with the outside world are displayed on the right. Arrows show the direction of signal travel. From the direction of the arrows we deduce that the mixing takes place while all signals are analog. In modern chips the mixing takes place in a very powerful Digital Sound Processor (DSP).

3.6.3. The mixer

As was shown before, a sound card contains a signal mixer. This signal mixer is controlled by a piece of software which is also called a mixer. This (software) mixer pops up an interface that lets you control the various inputs and outputs of the hardware. If you click on the volume control icon in the Windows tool bar, i.e. the small loudspeaker icon, the Windows output mixer pops up as is displayed in figure 3.7.[6] You raise the Windows input mixer, i.e. that part of the mixer that controls the inputs to the computer, by selecting the "Properties" button in the Options menu followed by choosing "Recording": figure 3.8 shows up.

[6]If there is no little loudspeaker icon in the toolbar you can also get the mixers via the Windows start menu at the bottom left corner of the screen. The exact sequence to reach the mixers may vary somewhat but the following choices have to be made: "Control panel" followed by "Sounds and audio" where you choose the "Audio" tab. Now you can reach the output- and the input mixers via the "Volume..." buttons in "Sound playback" or "Sound recording", respectively.

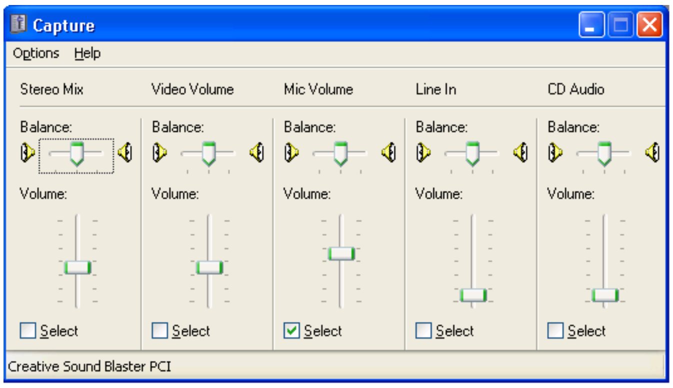

Volume Control

Options Help

Volume Control	Wave	Mic Volume	Line In	CD Audio
☐ Mute all	☐ Mute	☑ Mute	☐ Mute	☐ Mute

Creative Sound Blaster PCI

Figure 3.7.: The Windows output mixer.


Capture

Options Help

Stereo Mix	Video Volume	Mic Volume	Line In	CD Audio
☐ Select	☐ Select	☑ Select	☐ Select	☐ Select

Creative Sound Blaster PCI

Figure 3.8.: The Windows input mixer.

For making a recording on the computer we use *The Golden Mixer Rule*: deselect all input devices except the one you want to use. [7]

[7]The standard mixer software in Windows does not allow for mixing INPUT sources, only for PLAYING! Selection of an input source deselects all other sources automatically. So, it's not really a mixer but an input selector!

3.6.4. Analog to Digital Conversion

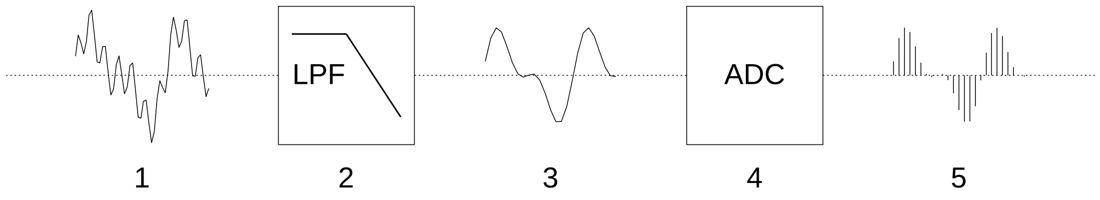


Figure 3.9.: The analog to digital conversion process.

In figure 3.9 the analog to digital conversion process is shown. The signal 1 enters the low-pass filter 2 where frequencies that are above the Nyquist frequency are filtered out.[8] The low-pass filtered signal 3 then enters 4, the analog to digital converter (ADC). The ADC converts the analog signal 3 into a series of numbers 5.

The ADC uses a clock that ticks at a regular time interval. This time interval is the inverse of the sampling frequency. For example, if the sampling frequency is 44100 Hz, then we have clock ticks at intervals of 1/44100 s. For each small time interval the ADC first measures the average amplitude of the signal 3. This is called *sampling*. In the next phase, called *quantization*, this analog average value is converted to a binary number and immediately sent to the output of the ADC. In assigning this number, the ADC can not exactly represent all possible analog values. Giving the precision of its internal circuitry, it quantises the value with a certain precision. This precision is expressed as a number of bits that the ADC uses for the representation of the value. In figure 3.10 we show how this number of bits influences the precision of the representation.

In the top left in panel (a) we see the analog signal. Panel (b) shows the crude quantization with the two possible levels allowed in a one bit quantizer. Panel (c) shows the four possible levels of a 2-bit quantizer. As the number of bits (n) increases the numbers of levels increases as 2^n. In this way we can make the digital representation approximate the analog representation as closely as we want. Modern ADC's use at least 16 bits for the representation of analog values. This 16-bit precision is also used on a CD-audio on which every second of audio in each of the two channels is represented by 44100 numbers of 16-bit precision.

3.6.4.1. Aliasing

The low-pass filtering step 2 in figure 3.9 is essential for the digitized signal to be a faithful representation of the original (in the frequency interval we are interested in). Shannon and Nyquist proved in the 1930's that for the digital signal to be a faithful representation of the analog signal, a relation between the sampling frequency and the bandwidth of the signal had

[8]The frequency that is at half the sampling frequency is called the *Nyquist frequency*. For example, if the sampling frequency is 44100 Hz, the Nyquist frequency is 22050 Hz.

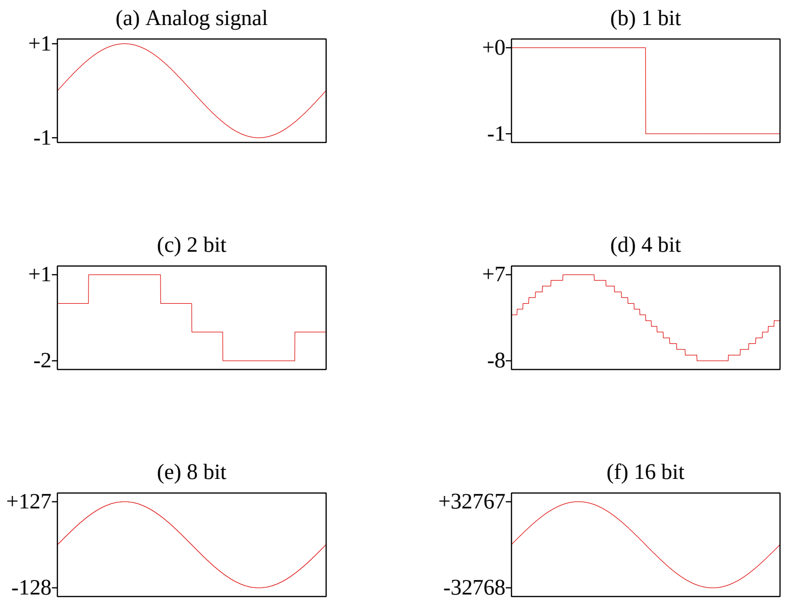


Figure 3.10.: The influence of the number of bits in the quantization step on the faithfulness of the digital representation.

to be maintained. For speech and audio sounds this relation is expressed by the following theorem.

The *Nyquist-Shannon sampling theorem*: A sound $s(t)$ that contains no frequencies higher than F hertz is completely determined by giving its sample values at a series of points spaced $1/(2F)$ seconds apart.

The number of sample values per second corresponds to the term *sampling frequency*. Sample values at intervals of $1/(2F)$ s translate to a sampling frequency of $2F$ hertz. A variant of the above theorem is: if the highest frequency in a sound is F hertz then the minimal sampling frequency we need is $2F$ hertz. We know that the highest frequencies human beings can hear is nearly 20 kHz. To faithfully represent frequencies that high we have to use a sampling frequency that is at least twice as high. Hence the 44100 Hz sampling frequencies used in CD-audio. All DAC's have a fixed highest sampling frequency and to guarantee that the input contains no frequencies higher than half this frequency we have to filter them out. If we don't filter out these frequencies, they get aliased and would also contribute to the digitized representation. A famous non-audio example of aliasing is in cowboy movies where the spokes in the wheels of the stage coach sometimes seem to turn backwards; the aliasing here is caused by not having taken enough pictures per second. In figure 3.11 we see an example of aliasing. The figure shows with black solid poles the result of sampling a sine of 100 Hz with

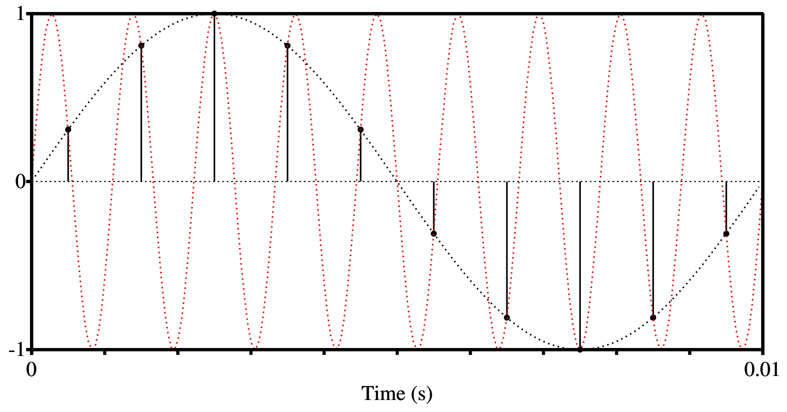


Figure 3.11.: Aliasing example. The red dotted analog 900 Hz tone gets aliased to the black dotted 100 Hz tone after analog to digital conversion with 1000 Hz sampling frequency.

a sampling frequency of 1000 Hz. This sampling frequency can be easily checked from the figure: we have 10 sample values in 0.01 s which make 1000 sample values in one second. As a reference, the analog sine signal is also drawn with a black dotted line. Therefore, the black dotted line could represent the analog signal before it is converted to a digital signal, while the black dotted poles are the output of the ADC. The red dotted line shows nine periods of an analog sine in this same 0.01 s interval and accordingly has a frequency 900 Hz. The figure makes clear that if the red dotted 900 Hz signal were offered to the ADC instead of the black dotted 100 Hz signal, the analog to digital conversion process would have resulted in the same black poles. This means that from the output of the ADC we can not reconstruct anymore whether a 900 Hz or a 100 Hz sine was digitized: if we have a signal that contains besides a sine of 100 Hz also a sine of 900 Hz then, after the analog to digital conversion with a sampling frequency of 1000 Hz, only one frequency is left, namely the 100 Hz frequency. From this frequency we can not reconstruct how much the 100 Hz component contributed and how much was aliased from the 900 Hz frequency. This is a very undesirable situation and therefore we have to take care to avoid it. It could happen because the 900 Hz frequency is above the 500 Hz Nyquist frequency. The solution is called low-pass filtering. Before the ADC we install a filter that lets frequencies lower that the Nyquist pass and blocks frequencies higher than the Nyquist.

The following script makes aliasing audible.

```
Create Sound from formula: "s1", 1, 0, 1, 11025, "0.5*sin(2*pi*500*x)"
Create Sound from formula: "s2", 1, 0, 1, 11025, "0.5*sin(2*pi*44600*x)"
Create Sound from formula: "s3", 1, 0, 1, 11025, "0.5*sin(2*pi*500*x)
  ... +0.5*sin(2*pi*44600*x)"
```

The script creates three sounds of 1 second duration, all with a sampling frequency of 11025 Hz. For the first sound, s1, the formula says that a tone of frequency 500 Hz is to be generated. The second sound is generated from a frequency of 44600 Hz. The third sound is the sum of the two previous sounds. The 44600 Hz is a frequency so high that a human being can not hear it, you have to be a dolphin to be able to hear these frequencies. However, if you listen to the three sounds all are very audible pure tones of one frequency. What is going on? Why do we hear sounds we are not supposed to hear? The functions that we use in the formula in the `Create Sound from formula...` command are defined for all values of x. It is the fifth argument of this command, the value for the sampling frequency, that orders that the formula is to be evaluated at 11025 different values of x in the interval from 0 to 1 s, i.e. the analog formula is sampled with a sampling frequency of 11025 Hz. There is an analog to digital converter working in the `Create Sound from formula...` command! It is not a real hardware converter that you can touch: it is going on in the software. This command therefore simulates an analog to digital converter.[9] The 44600 Hz tone has not been represented faithfully because the sampling frequency, 11025 Hz, was too low. To faithfully represent a 44600 Hz tone we need at least a sampling frequency of 89200 (= 2 × 44600) Hz. What happened to the 44600 Hz signal is the same as what happened to the 900 Hz signal in figure 3.11: aliasing. The 44600 (= 4 × 11025 + 500) Hz was aliased to a 500 Hz frequency.

3.6.5. Digital to Analog Conversion

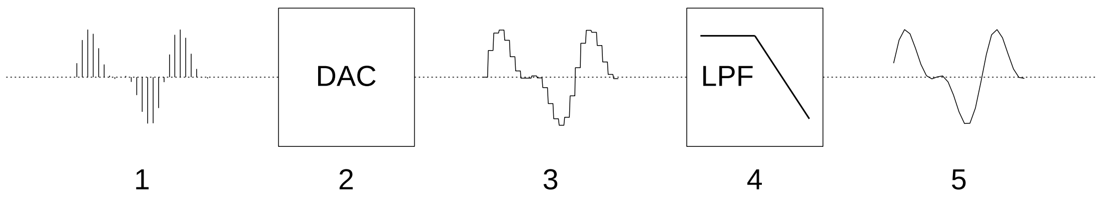


Figure 3.12.: The digital to analog conversion process.

In figure 3.12 the digital to analog conversion process is shown. This process is almost the reverse of the analog to digital conversion process. We start at 1 with a series of numbers as input to the digital to analog converter 2. At each clock tick, the DAC converts a number to an analog voltage and maintains that voltage on its output until the next clock tick. Then a new number is processed. This results in a not so smooth step-like signal 3. If made audible this signal would sound harsh. In step 4, this step-like signal is low-pass filtered to remove frequencies above the Nyquist frequency.

[9] In fact the simulation of the analog to digital conversion is much better than any hardware device now and in the foreseeable future can deliver. The quantization in Praat is only limited by the precision of the floating point arithmetic units. Sounds are represented with double precision numbers: this roughly corresponds to a 52-bit precision. The best hardware nowadays quantizes with 24 bits of precision.

3.6.6. The Digital Signal Processor

Many sound chips nowadays contain a special purpose processor for digital sounds, the Digital Signal Processor (DSP). Without going into details, these processors are specialised for digital signal processing.

4. Praat scripting

A script is a text that controls the actions of one (or more) programs, here Praat. The format of this script text is not completely free but must confirm to certain syntax rules. For example, a script text may have Praat menu and Praat action commands. When the actions in the script text are performed by Praat we speak of "running a script". When you run a script, the text of the script will be *interpreted* and the corresponding actions will be performed. The part of Praat that reads and interprets the script text and then initiates these actions is called an *interpreter*. In short: a script is run by the interpretor.

A script can be useful for various reasons.

- To automate repetitive actions. You have to do the same series of analyses on a large corpus and you don't want to sit for months at the computer clicking away to do your analyses on those thousands of files. Instead, you write a script that performs all necessary steps, for example, reading a sound file from disk, performing a pitch analysis and saving the results. For these cases you first test the script thoroughly on a small number of files and then order Praat to run the script on all the files in the corpus. You sit back and relax while all the analyzes are carried out automatically.
- To log actions. If you want to know *later* what you are doing *now* to achieve a certain result, you can save your actions in a script and save that script to a file. If you want to repeat the actions at a later moment you open the script file in Praat and run it.
- To communicate unambiguously to other people what *you* have done and how *they* may achieve the same results. We have already seen many example of this use of scripting in previous chapters. In this book many examples will be accompanied by a script that you can download.
- To make drawings in the picture window. Especially for articles you want to add all kinds of additional info in a drawing. A script lets you successively add more and more to a picture in the drawing window. Nearly all drawings in this book were produced with a script.
- To add a new button in the menu. For example, you have a series of actions on a selected sound that have to be performed in a prescribed order. You may script these actions and define a new button in the dynamic menu so every time you have a sound selected and you click that button, the actions in the associated script will be carried out in the prescribed order.

Examples of the various uses of scripting will be given in the sequel. We will start by showing you how simple it is to add functionality to Praat once you know how to script. To start scripting, in its most elementary form, you do not have to learn any new commands to address the functionality of Praat: you just use the commands that you already know, i.e. a simple script is a sequence of one or more lines that are copies of the text that is on a command button. For example if you have created a sound and want a command in the script to play this sound, a single script line with only the text `Play` suffices.

4.1. Execution of Praat commands in a script

As we recall from section 1.3, the prefered way to use a Praat command in a script is simply copying the command, replacing the three dots (...) with a (:) and separation of the arguments with a comma (,) while surrouding each text argument with double quotes ("). For a command that doesn't have arguments like for example the Play command for a sound, we simply type `Play`. To mimic the form in figure 1.2 we used

```
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100,
... 100, 0.2, 0.01, 0.01
```

If we want to use the outcome of a Praat command in for example a complicated numerical expression we have to use a somewhat more extensive formulation (10 characters more). Say we want to calculate halve the duration of a selected sound. The following scriplet does so:

```
duration = Get total duration
mean2 = 0.5 * duration
```

We can use the following one-liner instead:

```
mean2 = 0.5 * do ("Get total duration")
```

The `do` is a function that accepts as first argument the command as it is on the button between double quotes, followed by the comma separated arguments. For example, the following two queries of a sound have the same effect:

```
minimum = Get minimum: 0, 0, "Sinc70"
minimum = do ("Get minimum...", 0, 0,  "Sinc70")
```

The first line saves you 10 characters in typing a command that has three dots and 7 characters for a command without dots. In this book we will mostly use the first notation without the explicit `do`. Besides the `do` there is `do$` which returns a string instead of a number.

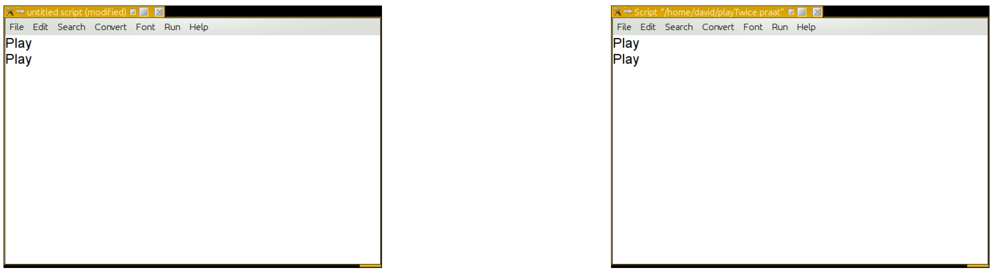
4.2. Defining a new button in Praat

In this section you will learn how to extend the dynamic menu of a sound with a new button. This may sound like a very complicated task but we will show you that it is very easy to do in Praat. The procedure to achieve this consists of three steps:

1. You edit a script that has the desired functionality.
2. You save the script as a file on the computer's disk.
3. You connect the saved script to a new button in the dynamic menu.

The example we use is to add a new button with the text “Play twice” that is positioned just underneath the “Play” button in the dynamic menu of a sound. When pushed a selected sound will be played twice. This is a toy problem, but once you know how to script it, you have mastered the basics and can go on to do more complex and interesting things. In order to do so we first need a sound. If you don’t have any sounds in the Object list yet, first create one. Use the New → Sound → Create Sound as pure tone... command,[1] click OK and a newly created sound appears in the list. How do we play a sound twice? Well, select the sound and click the “Play” button twice. If we have more sounds that we want to have played twice, we select the next sound, click twice on “Play”, et cetera.

Start by opening the script editor. Click on the “New Praat script” command that you find in the leftmost menu item labeled “Praat”. A script editor window pops up and you type on two consecutive lines the same `Play`. The left panel in figure 4.1 gives an impression of the script editor with the two typed lines. Because the script has not been saved yet and we have added


untitled script (modified)
File Edit Search Convert Font Run Help
Script "/home/david/playTwice.praat"
File Edit Search Convert Font Run Help

```
Play
Play
```

```
Play
Play
```

Figure 4.1.: The script editor before and after saving.

two lines, the window bar is titled “untitled script (modified)”. We save the script as a file on the computer disk, with the name `playTwice.praat`, by choosing the “Save as...” command from the File menu. After saving, the title of the script editor will reflect this name change as the right panel in figure 4.1 shows.

Now try out the script: first select a sound from the Object window. Then choose the “Run” command from the Run menu in the script editor. If you have typed the Play commands correctly you will hear the sound played twice. The actions you just performed are the basics of scripting: opening the script editor, typing in some script lines and trying them out by using the “Run” command. As you click the “Run” command you saw another option “Run selection” whose function might be obvious now: it only runs the part of the script that you have selected. For example, if you select one of the Play commands and next click on the “Run selection” command you will only hear it played once. Clicking on the “Run” command will always result in the execution of the complete contents of the script editor (even if there is a selection). The “Run selection” command gives you the opportunity to only execute part of the script.

[1]This is a shorthand notation for the path to find the `Create Sound from formula...` command: click on the New menu, this opens a new list of possible commands. Click on the Sound option in this list. This opens a new list from which you can choose the given command. This notation is sometimes necessary so *you* can find a command, Praat knows where to find its commands.

We have the script working and now it is the time to add the new button with the text “Play twice”. Defining a new button in Praat is done by associating a script with the new button; clicking the button will then run the script. This is easy to accomplish: from the script editor’s file menu as shown in figure 4.2 choose the ”Add to dynamic menu...” command and then

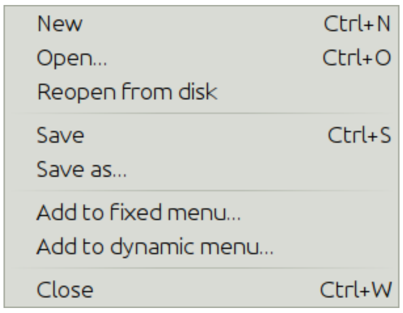


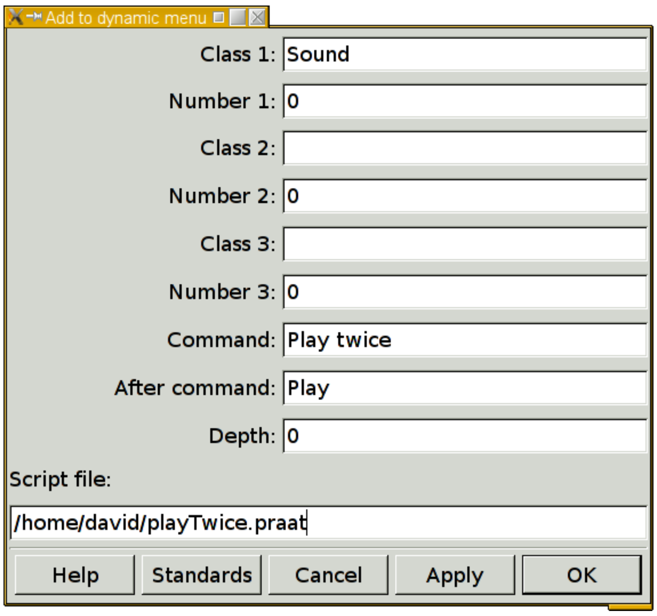
Figure 4.2.: The script editor’s File menu.

a form like the one displayed in figure 4.3 pops up. We can now add the new functionality to the dynamic menu. In the figure we have already modified two fields. The first modified field labeled “Command” originally showed “Do it...” and now “Play twice”. The text in the “Command” field will be displayed on the new button. The contents of the second modified field, “After command”, directs Praat to place the new button after the command you give here. By filling out “Play” here, the new “Play twice” button will appear in the list after the existing “Play” button. If you leave the “After command” field empty, or you gave a non-existing command name, Praat will place the new button at the bottom of the dynamic menu list. The last field “Script file” contains the complete file name of your script. Its content is automatically placed there by Praat. Be aware that file naming is different on Linux, Macintosh and Windows. For example, on Linux, the maps in the path are separated by ’/’ symbols. On Windows, maps are separated by the ‘\’ symbol.

After you have clicked the OK button, the dynamic menu changes and a new button with the “Play twice” text appears below the “Play” button. The right pane in figure 4.3 shows what the upper part of the sound’s dynamic menu looks like with the new button added. The newly defined button has the functionality of the script that you associated with it. You may close the script editor now.

4.3. Removing the newly defined button

The newly defined button works, it has served its only purpose, i.e. showing how easy it was to define a new button in the dynamic menu, and now it is time to remove it. To do this we open the buttons editor which you can reach via the path Praat > Preferences > Buttons.... The buttons editor allows you to determine which buttons will show up in the interface and which won’t. This editor creates the possibility to *hide* buttons from view that you either don’t


Add to dynamic menu

Class 1:	Sound
Number 1:	0
Class 2:	
Number 2:	0
Class 3:	
Number 3:	0
Command:	Play twice
After command:	Play
Depth:	0

Script file:

/home/david/playTwice.praat

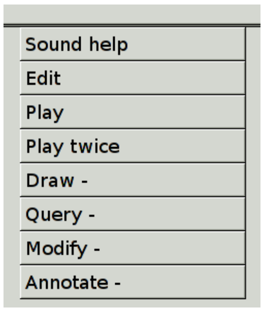
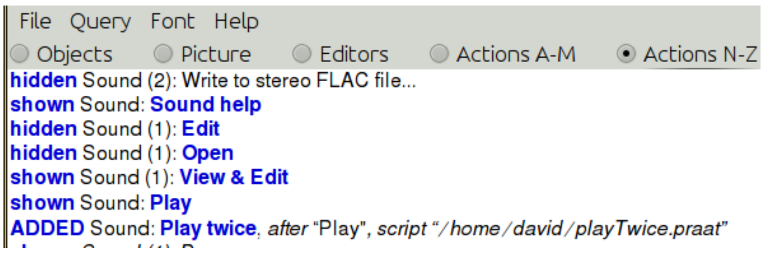


Figure 4.3.: The script editor's "Add to dynamic menu..." form and its effect on the dynamic menu of a sound.

want to see or never want to use and also to *remove* buttons that you have added yourself.[2] In this case you want to remove the "Play twice" button. The buttons editor cannot show you all buttons at once because there are thousands of defined buttons in Praat. Therefore, buttons are grouped into categories and these are shown in the editor just above the scrollable part, they are labeled "Objects", "Picture", "Editors", "Actions A-M" and finally "Actions N-Z". The last two categories concern among others the dynamic menu buttons. Selecting one of them displays all the buttons available for object types that start with a character in the range "A-M" or "N-Z", respectively. Because the "Play twice" button only works if a Sound object is selected, and "Sound" starts with an "S", we have to choose the category "Actions N-Z" as is displayed in figure 4.4. Lots of lines are shown in the buttons editor. Most lines start with the word "shown" in blue colour, followed by the object type and, again in blue colour, an action that can be performed with this object. The text of the action is the text you will also find on the corresponding button. We have to scroll down until we see the part that resembles the part shown in figure 4.4. The bottom line shows the characteristics of the "Play twice" button. If we click on the blue "ADDED", the text in the buttons editor immediately changes to "REMOVED" and the "Play twice" button instantly disappears from the dynamic menu. Clicking that same line again, will change the text back to "ADDED" and will make the button reappear.

Some final remarks on adding/removing/hiding buttons.

[2]And the maintainers of Praat can guarantee that old scripts keep working by hiding, for example, buttons whose names have changed.



hidden Sound (2): Write to stereo FLAC file...
shown Sound: Sound help
hidden Sound (1): Edit
hidden Sound (1): Open
shown Sound (1): View & Edit
shown Sound: Play
ADDED Sound: Play twice, after "Play", script "/home/david/playTwice.praat"

Figure 4.4.: Part of the buttons editor after first choosing "Actions N-Z" followed by scrolling to the actions for a Sound object.

- The first line in figure 4.4 shows a command that is normally hidden from the dynamic part of the Save menu. If you click on the word "hidden" this word will change to "SHOWN" and the action will be available in the Save menu of a sound. The first word on a line is a toggle and switches between either "hidden" and "shown" or between "ADDED" and "REMOVED". Actually there is semantics in the capitalization of the words too, as a capitalized word indicates that *you* changed a setting, while normal characters indicate the setting was done by the authors of Praat.

- You can use *all* the actions in the buttons editor as if they were real buttons, i.e. clicking on the "Sound help" in the second line of the editor in figure 4.4 will display the help window.

- A number between parenthesis after the object type indicates the exact number of objects that have to be selected in order to make the command available. The figure shows that to invoke the "View & Edit" command only one sound may be selected and to invoke the (hidden) "Save as stereo FLAC file..." command exactly two sounds have to be selected together.

- We have chosen to add a new button to the dynamic menu and *not* to a fixed menu because we only want our new command to be available if a *sound is selected*. Adding the new button to one of the fixed menus would always make the new button visible no matter which object were selected. The "Play twice" command does not make sense to all types of objects defined in Praat and, therefore, does not belong in any of the fixed menus of Praat.

- Our script was very simple and only contained the command "Play", however, we can use any other Praat command in the script because pushing the new button simply runs the associated script. If the script conforms to the syntax and semantics of Praat it will be run. All actions in Praat, i.e. all the buttons and all the forms, can be addressed in a script, and much more.

- You don't need to define a button if you want to run a script. The "Run" command in the script editor can take care of running a script. We just defined the new button to show you how easy it is to do so.

4.4. Create and play a tone

The previous "Play twice" example was a very simple one. We now create a substantially more powerful script that

- pops up a form to choose a frequency,
- creates a sound of a constant frequency with a duration of 0.2 seconds (such a sound is called a *tone*).
- plays the tone,
- write the frequency of the tone to the info window
- removes the tone.

In this example we first show how you can use Praat's history mechanism to help you during scripting. From this recalled history we can start with a crude version of the script and in a number of small steps improve upon this script until we reach the final version in section 4.4.5. During these improvements we introduce the use of *variables* and we learn how to create simple *forms* (a form is a window that delays the execution of a script until you supply the information that the form queries). In later sections we will elaborate more on variables and how to make use of them.

We already showed in section 2.7.1 how to create a tone with the familiar New → Sound → Create Sound as pure tone... command. We will start with the default settings of this command, click OK and a new sound appears in the list of objects named tone". It is a mono sound with a duration of 0.4 second and a sampling frequency of 44100 Hz. We listen to this sound by clicking on the "Play" button. This is how a pure tone sounds. We remove the sound by clicking on the red "Remove" button at the bottom of the Object window.

We start up the script editor and immediately choose from its Edit menu "Paste history". Now all the commands that we executed in the current Praat session, show up in the script editor because from the moment we start up Praat, the program maintains a history of all the commands and actions we that perform. In the script editor we have access to this history. In the Edit menu of the script editor we can see two commands that address the history. With one command we can paste the complete history in our script. With the other command we clear the complete history and start a fresh history recording. This history mechanism comes in handy if we want to try out a succession of commands. In this case we can start up the script editor, clear the history, perform a number of actions, paste the actions in the script editor and then start editing these commands if necessary. [3]

In the script that we start working on now, remove all lines except for the (last three) lines that show in the following script.

```
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, 440.0, 0.2, 0.01, 0.01
Play
Remove
```

[3] This is how we could have started in the first place but then you would have missed the experience of a script editor full of previous commands.

If we run this script it repeats the three actions we have just carried out.

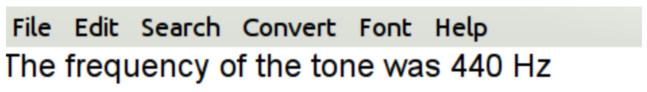
Beware: the first two lines in this script were actually one long script line. We have split this line to fit on the paper. This is something we are always allowed to do with long script lines that don't fit in the normal width of the editor (or paper). There are two things to keep in mind when splitting lines: (1) we can only split on positions where adding extra white space wouldn't matter and (2) we have to start the continuation part with three consecutive dots (...). These three dots signal that this line is a continuation of the previous line.[4] We may use as many continuation lines as we wish and white space before the three dots is allowed as the above script shows.

Note that the fields that show up from top to bottom in a form, are shown from left to right in a script line. We must always maintain this correspondence between the position of a field in a form and the position of the field in a line of text in a script.

We will edit these lines until they have the form that we want. We only have to change the **End time** to 0.2 and we add a line after the Play command that writes the frequency to the info window. The script is now:

```
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, 440.0, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was 440 Hz"
Remove
```

The `writeInfoLine` first clears the info window and then writes text to it. If you run the new script, the duration of the sound is now shorter than in the previous version. The info window will be as in figure 4.5



The frequency of the tone was 440 Hz

Figure 4.5.: The info window after executing a simple script.

After running this script a couple of times we are getting bored. The script plays a tone but it is the same tone all the time. We like to vary the tone's frequency. We can do so by typing another number instead of the "440.0" in the **Tone frequency** field and changing the number in the `writeInfoLine` function. Let's say we change both numbers "`440`" in the script above to "`1000`". If we run the script now, you will hear a higher tone, one of 1000 Hz and the info window will also show the new frequency.

By using this script we have actually saved some time. Instead of the three actions: creating a tone, playing the tone and removing the tone, we only have one action now: running the script.

[4]All commands in Praat that put a form on the screen also end with three dots.

4.4.1. Improvement 1, introducing a variable

To change the frequency of the tone we had to change the number "440" at two places, in the formula and in the line the writeInfoLine. This, however, is error prone. For instance we could make a typing error or forget to change the second occurence. If a typing error makes the syntax incorrect then Praat, of course, generates an error message with detailed information about the line number and the contents of the line where things went wrong, but nevertheless, we have to carefully check at two different positions. We can achieve a more simple version by introducing *a variable* for the frequency as the following script shows.

```
frequency = 1000
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was ", frequency, " Hz"
Remove
```

The first line introduces a variable with the *name* `frequency` and assigns the *value* `1000` to it. A variable is something like a labeled box in the memory of the computer: the label on the box is the variable's name and the contents of the box is the variable's value. Every time that you use the name of a variable in a script, the computer will use its value. When the second line is executed the value of the frequency variable is used. The `writeInfoLine` now also writes the value of the `frequency` variable to the info window. If you run the modified script, the results will be exactly as they were in the previous section. However, by using the variable `frequency` we have achieved something important: if we want want to change the frequency of the tone and simultaneously the information displayed about this frequency in the info window, we now only need to change the frequency value at *one* place in the new script instead of the *two* occurrences in the previous script.

Keep in mind that variables always have to *start* with a lowercase character and only the characters a-z, A-Z, 0-9 and the underscore "_" are allowed in the rest of the variable name. Therefore `frequency`, `a34` and `bcX3` are valid variable names while `Frequency`, `$a` and `_frequency` are not. The characters "." and "$" are special in a variable name. Only Praat menu commands (have to) start with an uppercase character.

4.4.2. Improvement 2, defining a minimum form

Now we like to skip editing the script each time we want a tone with a different frequency. We like the script to raise a form in which we can type the desired frequency. The following script improves on what we had.

```
form Play tone
positive frequency
endform
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was ", frequency, " Hz"
Remove
```

When run, the script raises the form displayed in figure 4.6. This form is defined in the first three lines of the script. The first line defines the title for the form, i.e. the text "Play

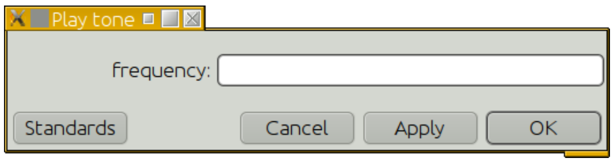


Figure 4.6.: The first form.

tone" at the top. You can choose your own text, you can even have no text at all.[5] The text of the window should describe or summarize the actions of the script in a compact way. The second line in the script defines a numeric field named **frequency** that allows only *positive* numbers. If you run the script and type a number in the **frequency** field that is less than or equal to zero, a message is generated that will inform you that you made an error. The second line of the script serves two things. In the first place it defines the field of the formed labeled "frequency" and at the same time it guarantees that a new variable is created that also has the name "frequency". This new frequency *variable* will receive the value of the **frequency** *field* once OK is clicked. In this way the script and its form communicate: *the field name in the form corresponds to a variable that bears the same name in the script.*[6] The `endform` closes the definition of the form.

Note that, again, all these form definitions (form, endform and positive) *start* with a *lowercase* character. Only Praat commands and actions start with an *uppercase* character.

4.4.3. Improvement 3, default value in the form

A minor annoyance of the previous script is that when the form pops up you have no idea what you should type in the **frequency** field. If you click OK without typing anything, an error message pops up. It would be nice when the script supplied a default value that guarantees that the script runs if the user just clicks OK. Error messages should pop up only if you do something wrong, there should be nothing wrong in just accepting defaults. The following script preloads a default value in the **frequency** field. This number happens to be `440.0`.[7]

```
form Play tone
positive frequency 440.0
endform
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was ", frequency, " Hz"
Remove
```

The form that pops up is like the form in figure 4.6 but now shows the number 440.0 in the **frequency** field.

The basic elements for constructing a form are now in place: each user supplied argument needs at least a line in the script that starts with `<argument_type>` followed by `<argument_name>`

[5]In the latter case there has to be at least one white space after the `form` text in the script.

[6]Use the underscore '_' to create white space. For example, the field name "frequency_value" with associated variable "frequency_value" shows as "frequency value" in the form.

[7]To inform the user that real numbers are allowed, it is better to preload with a number that makes this explicit. We therefore used "440.0" instead of "440" for the default value.

and optionally followed by `<argument_default_value>`. In the example above we have used `positive` as `<argument_type>`, `frequency` as the `<argument_name>` and `440.0` as `<argument_default_value>`. Of course more argument types than `positive` are possible and you can use the following: `real`, `positive`, `integer`, `natural`, `word`, `sentence`, `text`, `boolean`, `choice` and `comment`. See the scripting manual in Praat for more explanation about argument types.

4.4.4. Improvement 4, field names start with uppercase character

The next improvements are only cosmetic, but nevertheless important. First, we want to see **Frequency** as the title of the field instead of **frequency**, as all field names in a Praat form start with an uppercase character. As we saw earlier, variables can only start with lowercase characters, therefore, to avoid a conflict, Praat automatically converts the first character of the associated variable to lowercase. In this way the field name **Frequency** can start with an uppercase character and the associated variable `frequency` will start with the lowercase character.

The other cosmetic change is that the second line is indented now, to let the `form` and the `endform` stand out. The first three lines of the script now read as follows.

```
form Play tone
    positive Frequency 440.0
endform
```

4.4.5. Final form, input values have units

The final improvement is cosmetic again. We want to communicate that the unit for the **Frequency** field is hertz. In this script the name of the field has been changed to **Fre-**

Script 4.1 The final Play tone example.

```
form Play tone
  positive Frequency_(Hz) 440.0
endform
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was ", frequency, " Hz"
Remove
```

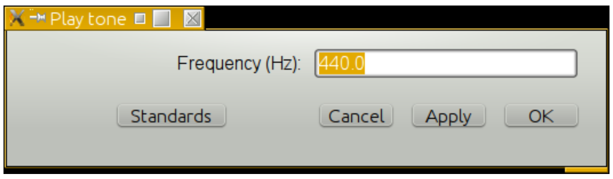


Figure 4.7.: The final Play tone form.

quency_(Hz). Despite this change, the associated variable is still named `frequency`. During the creation of the form, Praat chops off the last part **_(Hz)** to create the variable. Actually, Praat chops off **_(** and everything else that follows from the field name.

4.4.6. Variation

Suppose you want to keep the Sounds that were created by the script. You remove the last line in the script and all the newly created Sounds will be kept in the list of objects. However, they all carry the same name. You want them to have a meaningful name that enables you to easily identify the Sounds. The new script:

```
form Play tone
  positive Frequency_(Hz) 440.0
endform
Create Sound as pure tone: "s_" + string$ (frequency), 1, 0, 0.4, 44100,
    ... frequency, 0.2, 0.01, 0.01
Play
writeInfoLine: "The frequency of the tone was ", frequency, " Hz"
```

There are two new things in this script: the `string$ (frequency)` part converts the `frequency` number to a string and the "+" operator adds i.e concatenates two strings. All your sounds will have names now that start with `s_` and have the frequency attached. For example if you run this script and type "1000" in the **Frequency** field, the sound appears with the name `s_1000`. Note that only the integer part of a number will be in the name because dots are not allowed in object names.

4.5. More on variables

A variable is a named location in the memory of the computer that holds a value. This value can be used and modified in the script. A variable is like a labeled box, the variable's name is the label on the box and the variable's value is the contents of the box. If we use the variable on the *left* side of an equal sign we put new contents in the box, if we use the variable's name on the *right* side of an equal sign we use the value in the box.

```
duration = 0.2
start = 0.1
end = start + duration
```

In the first line the variable `duration` is given the numeric value 0.2 and in the second line variable `start` is given the value 0.1. In other words we put the numeric values 0.2 and 0.1 in the boxes labeled `duration` and `start`. Finally, in the third line we use the contents of the boxes `start` and `duration`, add these values and put the result, which is the numeric value 0.3, in a box labeled `end`.

The name of a variable has to start with a lowercase character. Although nothing forces you to do so, in general, you want to give a variable a meaningful name like for example `time` or `frequency` or `duration` and not something like `xDS24_hY`. Variable names are case sensitive, i.e. all the variables with names `xyz`, `xYz`, `xyZ` and `xYZ` are different variables.

There are two kind of variables in Praat: variables that hold numbers and variables that hold text. The variables that hold text must end with a dollar sign `$` and are called *text variables* or *string variables* to differentiate them from *numeric variables*. Variables like `frequency` and `duration` therefore hold numeric values while the variables like `name$` and `expression$` will hold text values.

Why do we need variables, can't we do without them? Of course we can, for very simple scripts we don't need variables. Some reasons why variables are used.

- Because it saves us some typing. Instead of calculating the perimeter of a circle with radius 1.5 as "2*3.1415926535897932384626433*1.5" we might use the predefined constant `pi` and write it as `2*pi*1.5`. If you use a certain constant value over and over again in a script it might be more convenient to use a variable for it too. We already saw an example with the frequency variable in previous scripts.

- To better express what a calculation is used for. For example consider the following expression for the perimeter of a circle where the chosen variable names express what you are calculating.

```
# instead of 2*3.1415926535897932384626433*1.5
radius = 1.5
perimeter = 2 * pi * radius
```

- Sometimes we just have to. If we want the user to modify a field in a form the user input has to be *variable*. For example, the input form used in the create and play a tone example of section 4.7 necessarily needs a variable to store the user input for the **Frequency** field.

- It makes scripting a lot easier, more powerful, and, more fun to do. Without variables the only thing we could do is the repetition of a number of Praat commands.

4.5.1. Predefined constants

There are two predefined *mathematical constants* in Praat, `pi` and `e`, which are used as an approximation for the mathematical numbers π and e.[8] Because they are constants `pi` and `e` can only appear on the right-hand side of an equal sign in your script, i.e. their values *cannot* be changed. Praat will issue an error message if you try to. The number π is defined as the ratio between the circumference and the diameter of a circle. The number e is Euler's number and is commonly defined as the base of the natural logarithm. Several other definitions for e exist. These numbers are used so often in scripting that they deserve special treatment. Table 4.1 shows the relation between the mathematical symbol, the name used in Praat and its numerical approximation. This means that if you use `pi` in a numerical context, for example in the calculation of the number `2*pi`, the interpreter will substitute for `pi` the value in the third column resulting in the number 6.2831853071795864769252867665590057683944.

[8]The numbers π and e have an infinite number of decimals. Therefore, they cannot be represented exactly on a digital computer whose computing elements only allow calculations with a finite number of digits.

Table 4.1.: The predefined mathematical constants in Praat.

Mathematical symbol	Variable	Approximation in Praat
π	`pi`	3.1415926535897932384626433832795028841972
e	`e`	2.7182818284590452353602874713526624977572

4.5.2. Predefined variables in Praat

A *predefined variables* is a variable that already was given a value by Praat. You do not need to assign a value, although it is not forbidden and the interpreter will therefore not complain most of the time. [9] We consider three types of predefined variables in Praat: predefined string variables, predefined numerical variables associated with matrix types like sound and spectrum and finally booleans associated with operating system identification.

4.5.2.1. Predefined string variables.

The following predefined variables can be used in a string or text context; of course they have to be quoted to be effective.

`newline$`: the newline character starts a new line.

`tab$`: the tab character insert a tab in a string

`shellDirectory$`: the directory you were when you launched praat. This information may be handy if you work in a DOS shell or in a Unix shell. If you don't know what I'm talking about you will never need to know about this variable and probably the following information in this section may be not relevant to you either.

`homeDirectory$`: your home directory.

`preferencesDirectory$`: the directory where Praat stores your preferences.

`temporaryDirectory$`: the directory available for temporary storage.

`defaultDirectory$`: the directory where the script resides.

The last five variables all are preloaded with, not necessarily different, directory names. For example the output of the following script

```
writeInfoLine: homeDirectory$
appendInfoLine: preferencesDirectory$
appendInfoLine: temporaryDirectory$, newline$
appendInfoLine: defaultDirectory$
```

on my system shows in the info window the following text:

[9]However, if you assign to the numerical variables associated with matrices (`xmin, xmax, nx, x1, dx, ymin, ymax, ny, y1, dy`) and you also use, for example, Formula... to manipulate a Matrix the interpreter will complain.

```
/home/david
/home/david/.praat-dir
/home/david

/home/david
```

Because of the `newline$` at the end of the third line there is an extra blank line in the info window. Three of the predefined string variables happen to identify the same directory on my system.

4.5.2.2. Numerical variables associated with matrices

The following predefined variables only make sense if you are working with a Matrix or object types derived from Matrix like a Sound. They are useful when you use the `Formula...` command.

`self`: the value of the current matrix element.
For example, for a selected sound applying the Modify → Formula... command with argument `2*self` multiplies all sound sample values by a factor of 2.

`row`, `col`: the current row and column number.
The following script creates a stereo sound with a tone of 200 Hz in the first channel and a tone of 400 Hz in the second channel.

```
Create Sound from formula: "stereo", 2, 0, 1, 44100,
  ... "0.5*sin(2*pi*row*200*x)"
```

`xmin`, `xmax`: start value and end value of the row domain. For a sound they represent the start time and the end time, for a spectrum the start frequency and the end frequency, for example.
Suppose you want to define a sound signal whose *amplitude* starts at a value of zero and then runs linearly to 1 at the end of the sound. Now, if the sound starts at 0 s and ends at 1 s it is simple. The **Formula** field of `Create sound from formula...`, which as we know specifies the amplitudes of the sound, can be as simple as: `x`. Because if the `x` which stands for time runs from 0 to 1 second the amplitude will also run from 0 to 1. However, if we don't want a 1 s duration but say 0.5 s we change the **End time** to 0.5 and now the **Formula** field must read `2*x`, because we have to reach an amplitude of 1 when `x` reaches 0.5. For a duration of 0.2 s the Formula field has to read `5*x`. Each time we change the End time, i.e. the duration, we have to change the formula too. This is very frustrating, and we have not even changed the starting time! Luckily there is an elegant way to solve this problem. We can use the time domain information, i.e. the `xmin` and the `xmax`, to construct a formula that will *always* give the correct amplitude behaviour, irrespective of the duration, and what's even better, it is also independent of the starting time. If the **Formula** field reads `(x-xmin)/(xmax - xmin)`, then we are done. Check: at the start of the sound `x` equals `xmin`, the numerator will be zero as will be the amplitude, at the end `x` equals `xmax` and the numerator and denominator are equal and the amplitude will equal one. For values of `x` in-between these extremes the increase in amplitude will be linear as the formula says.

`nx`: the number of samples in a row.

`dx`: the sampling period in the row domain. For a sound this value is the sampling time and its inverse, `1 / dx`, will give you the sampling frequency.

`x1`: the x-value of the first point in a row.

`x`: the x-value of the current point in a row.

`y`, `ymin`, `ymax`, `ny`, `dy`, `y1`: analogous to the `x` variables (i.e. the column).

For a mono sound of 1 second duration that starts at time 0 s with a sampling frequency of 16000 Hz the values of these predefined variables are

$$\begin{array}{lllll} \texttt{xmin} = 0; & \texttt{xmax} = 1; & \texttt{nx} = 16000; & \texttt{x1} = 3.125\cdot10^{-5}; & \texttt{dx} = 6.25\cdot10^{-5}; \\ \texttt{ymin} = 1; & \texttt{ymax} = 1; & \texttt{ny} = 1; & \texttt{y1} = 1; & \texttt{dy} = 1. \end{array}$$

For a stereo sound of 2 second duration that starts at 1 s with a 16000 Hz sampling frequency the values of these variables are

$$\begin{array}{lllll} \texttt{xmin} = 1; & \texttt{xmax} = 3; & \texttt{nx} = 32000; & \texttt{x1} = 3.125\cdot10^{-5}; & \texttt{dx} = 6.25\cdot10^{-5}; \\ \texttt{ymin} = 1; & \texttt{ymax} = 2; & \texttt{ny} = 2; & \texttt{y1} = 1; & \texttt{dy} = 1. \end{array}$$

For a more complex signal like the spectrum derived from the first sound by means of the FFT algorithm, the values are

$$\begin{array}{lllll} \texttt{xmin} = 0; & \texttt{xmax} = 8000; & \texttt{nx} = 8193; & \texttt{x1} = 0; & \texttt{dx} = 8000/8192; \\ \texttt{ymin} = 1; & \texttt{ymax} = 2; & \texttt{ny} = 2; & \texttt{y1} = 1; & \texttt{dy} = 1. \end{array}$$

For a Matrix object created by the command Create simple Matrix... xy 10 10 row*col, the values are

$$\begin{array}{lllll} \texttt{xmin} = 0.5; & \texttt{xmax} = 10.5; & \texttt{nx} = 10; & \texttt{x1} = 1; & \texttt{dx} = 1; \\ \texttt{ymin} = 0.5; & \texttt{ymax} = 10.5; & \texttt{ny} = 10; & \texttt{y1} = 1; & \texttt{dy} = 1. \end{array}$$

Warning: Try to avoid the variable names described in this section in an assignment, i.e. don't use them on the left-hand side of an equal sign, because of possible ambiguities. The following scriptlet is illegal in Praat and the interpreter will issue a warning that the variable dx is ambiguous. The variable is assigned the value of 10 but in the formula context the value of dx is equal to 1/16000.

```
Create Sound from formula: "s", 1, 0, 1, 16000, "1/2 * sin(2*pi*377*x)"
# illegal assignment of dx before using it in array context!
dx=5
Formula: "self*dx"
```

The following script, although not recommended, is legal because there is no possible conflict.

```
Create Sound from formula: "s", 1, 0, 1, 16000, "1/2 * sin(2*pi*377*x)"
Formula: "self*dx"
dx=5
```

4.5.2.3. Operating system identification variables.

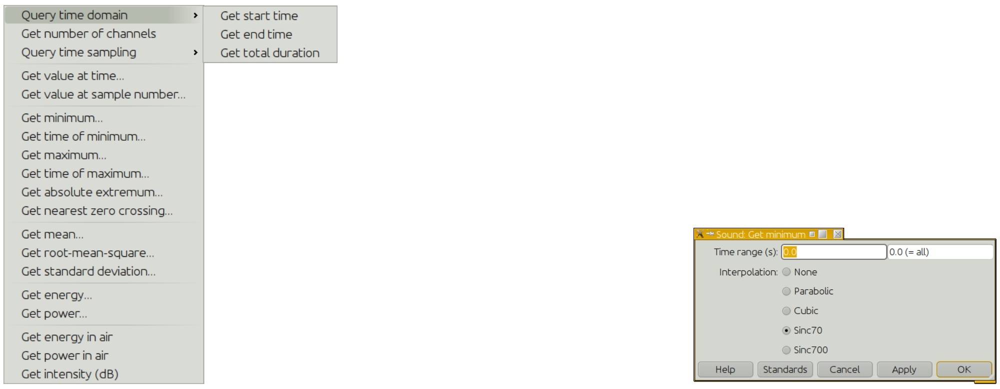
The following variables can be used to differentiate between operating systems and you will hopefully never have to use them. It is hardly necessary to write operating system dependent scripts, but just in case:

`macintosh`: this variable has the value 1 on a Macintosh and is 0 elsewhere.

`windows`: 1 on Windows, 0 elsewhere.

`unix`: 1 on Unix 0 elsewhere.

4.5.3. Accessing queries



Time range (s): 0.0 | 0.0 (= all)

Interpolation: None / Parabolic / Cubic / [✓ Sinc70] / Sinc700

Figure 4.8.: Left: the query menus of a Sound. Right: the Get minimum query form.

Most objects in Praat can be queried. In the left part of figure 4.8 we show queries for a sound. Needless to say that a sound must have been selected for these queries to appear. You note from the figure that some of the queries have been grouped like “Query time domain”. This query expands to three separate queries “Get start time”, “Get end time” and “Get total duration”. Now if you click for example on the “Get total duration” query, the result of the query will show itself in the info window: a new line appears that shows the total duration of the sound. If we want to have access to the total duration *from within a script*, we can simply do this by assigning the output of the query command to a variable like in the following scriptlet:

```
duration = Get total duration
writeInfoLine: "The duration is ", duration, " s."
```

The first line queries a selected sound for its total duration and assigns the output of this query to the variable `duration`. The last line then writes the information to the info window.

For queries that show a form, like for example the “Get minimum...” query whose form is shown in the right part of figure 4.8 you fill out the parameters like in the following scriptlet where we query for the minimum value in a sound.

```
minimum = Get minimum: 0, 0, "Sinc70"
```

For a time interval the default zero values generally mean that the whole domain is used (the `Sinc70` method uses a very precise interpolation to attain the real minimum of the sound).

4.6. Conditional expressions

One of the first things you need in a script is to be able to vary its execution path. Sometimes you only want a particular part of the script executed, if a certain condition is fulfilled. Suppose we want to check whether the frequency number a user supplies via a form is a valid frequency. A valid frequency for example has to be a number larger than or equal to zero. As we know from the previous section 3.6.4, a frequency for a sampled tone has to be lower than the Nyquist frequency of the sampled sound otherwise aliasing will occur. Here the positivity condition can be automatically maintained by the form and we don't have to test it explicitly in the script. However, if you happen to fill out a frequency larger than the Nyquist frequency, aliasing will occur and the frequency of the generated tone will not be as you typed. If we want to prevent this from happening we have to test whether the input is correct. In case the input is incorrect exit the script with an appropriate error message. Increasing the sampling frequency of the sound above 44100 Hz in order to faithfully represent the desired frequency is probably not an option. Most likely you will need a special sound card to create Sounds with a sampling frequency above 44100 Hz. Given the right hardware, *you* would not be able to hear the *ultrasonic* sound at all. If the frequency does not exceed 45 kHz a dog or a cat can hear it. If your tone is even higher, maybe a nearby swimming dolphin could hear it.[10] We change the script and include the following *conditional expression* in the Play tone script of section 4.4.5 after line 3:

```
if frequency >= 22050
    exitScript: "The frequency must be lower than 22050 Hz."
else
    Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
endif
```

In this way the generated tone will always have a correct frequency. A notational variant that would have the same effect is

```
if frequency >= 22050
    exitScript: The frequency must be lower than 22050 Hz.
endif
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
```

Frequencies too low for us to hear, say lower than 30 Hz, are called *infrasonic* frequencies. Elephants use infra sound to communicate. You could extend the script with an extra test for infrasound:[11]

```
if frequency >= 22050
    exitScript: "The frequency must be lower than 22050 Hz."
elsif frequency <= 30
```

[10]According to George M. Strain's website on hearing at Louisiana State University (http://www.lsu.edu/deafness/HearingRange.html).

[11]Note that the loudspeakers of PC's and especially laptops most of the time are very bad at representing low frequencies (say lower than 100 Hz).

```
    exitScript: "The frequency must be higher than 30 Hz."
endif
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
```

We can combine tests with “and” and “or” like in the following:

```
if frequency <= 30 or frequency >= 22050
  exitScript: "Frequency must be larger than 30 and smaller than 22050 Hz."
endif
Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
```

or in another variant

```
if frequency > 30 and frequency < 22050
    Create Sound as pure tone: "tone", 1, 0, 0.4, 44100, frequency, 0.2, 0.01, 0.01
else
    exitScript: "The frequency must be higher than 30 Hz and lower than 22050 Hz."
endif
```

4.6.1. Conditional expressions within a Formula...

For the conditional expression in a formula such as occur in the “Create Sound from formula...” command we have to use a syntactical variant of the `if`. Because a formula is essentially a one-liner we have to use the form

```
if <test> then <something> else <something else> endif
```

or

```
if <test> then <something> else <something else> fi
```

in which the `<test>`-parts are expressions and the else part is *not* optional. Instead of the closing `endif`, we could also have used the shorter `fi` as in

```
if <test> then <something> else <something else> fi
```

For example, the following one-liner creates a noise with a gap in it (or two noises if you like).

```
Create Sound from formula: "gap", 1, 0, 0.3, 44100,
  ... "if x > 0.1 and x < 0.2 then 0 else randomGauss (0,0.1) fi"
```

If you select an interval you can do this by combining the lower limit and upper limit with `and`, as the previous script does, or with `or`, as in following one. Both scripts result in exactly the same sound.

```
Create Sound from formula: "gap", 1, 0, 0.3, 44100,
  ... "if x <= 0.1 or x >= 0.2 then randomGauss (0,0.1) else 0 fi"
```

Another variant involves the use of the variable `self`.

```
Create Sound from formula: "gap", 1, 0, 0.3, 44100, "randomGauss (0,0.1)"
Formula: "if x > 0.1 and x < 0.2 then 0 else self fi"
```

We create the noise first and then modify the existing sound with a formula. In the else part the expression `self` essentially means “leave me alone”. The `self` after the `else` indicates that the `else` part applies no changes to the sound.

4.6.2. Create a stereo sound from a Formula

We can create stereo sounds in Praat in various ways. For example, we could read a stereo sound from a file with the "New → Read from file..." command. Or we could select two mono sound objects together and make a stereo sound from them by using the "Combine→ Combine to stereo" command. In this section, however, we use the "Create Sound from formula..." command to create a stereo sound because we want to show conditional expressions in formula's. We start by creating a stereo sound with identical sounds in the left and right channels. After this we will learn how to use a conditional expression to make different sounds in the left and the right channel. We will learn something about *beats* too.[12]

We start by creating a stereo sound in which each channel is the same combination of two tones that differ only slightly in frequency:

```
Create Sound from formula: "s", "Stereo", 0, 2, 44100,
  ... "1/2 * sin(2*pi*500*x) + 1/2 * sin(2*pi*505*x)"
Play
```

This command needs the `Stereo` option in the **Number of channels** field. Of course you could also have supplied the number 2 because a stereo sound has two channels by definition. The formula part now contains an expression that shows the sum of two tones. To check that the sound is stereo, click the "Edit" button in the dynamic menu. The two channels will appear in the sound window, one above the other (listening to a sound in order to decide whether it is mono or stereo sound generally does not give you any clue because many sound systems play a mono sound on both channels, i.e. as if it were a stereo sound with identical left and right channel).

Another way to check for stereo is to click the "Info" button at the bottom of the Object window if the sound is selected. A new Info window pops up, showing information about the selected sound. The info window starts with general information about the sound: its number in the list of objects, its type, name and the date and time the info was requested. If you repeat clicking the info button than this line will be the only line that changes. On line six the number of channels shows 2, which means that it is a stereo sound. The next lines show information on the time domain, the start time and end time values you supplied , the duration is a value calculated as the difference between the end and start time. The digital representation is given next. The number of samples is calculated as the product of duration and sampling frequency. The sampling period is the inverse of the sampling frequency, i.e. 1/44100 for the given example, and the first sample is located in the middle of the first sampling period.

When you listen to this sound, you will hear beats: the sound increases and decreases in intensity. The sounds in both channels are equal and we can hear the beats in each ear separately.

Next create a new stereo sound according to the following script:

```
Create Sound from formula: "s", "Stereo", 0, 2, 44100,
  ... "if row=1 then 1/2 * sin(2*pi*500*x) else 1/2 * sin(2*pi*505*x) endif"
```

or with the notational variant:

[12] A beat is an interference between two sounds of slightly different frequencies and you perceive beats as periodic variations in the intensity of the sound. The frequency of the beat is half the difference between the two frequencies.

```
Object id: 4
Object type: Sound
Object name: s
Date: Tue Apr 16 15:47:58 2013

Number of channels: 2 (stereo)
Time domain:
   Start time: 0 seconds
   End time: 2 seconds
   Total duration: 2 seconds
Time sampling:
   Number of samples: 88200
   Sampling period: 2.2675736961451248e-05 seconds
   Sampling frequency: 44100 Hz
   First sample centred at: 1.1337868480725639e-05 seconds
```

Figure 4.9.: The first part of the text in the Info window for a stereo sound.

```
Create Sound from formula: "s", "Stereo", 0, 1, 44100,
  ... "1/2 * sin(2*pi*(if row=1 then 500 else 505 endif)*x)"
```

These formulas make use of the internal representation of a stereo sound in Praat as two rows of numbers: the first row of numbers is for the first channel, the second row is for the second channel. The conditional expression in the formula part of the script above, directs the first row (channel 1) to contain a frequency of 500 Hz and the other row (channel 2) to contain a frequency of 505 Hz.[13]

Listen to this sound but *don't use* your headphones yet. Instead use the stereo speaker(s) from the computer. If everything works out alright, you will hear beats again.

Now use headphones, play the sound several times but listen to it only with the left ear and then only with the right ear. You will hear tones that differ slightly in frequency. Finally, listen with both ears and you will hear beats. In contrast to the beats in the previous examples which were present in the audio signals entering your ears, these beats are constructed only in your head. This is a nice demonstration that information from both ears is integrated in the brain.

In figure 4.10 the difference between the two stereo sounds we have created in this section is illustrated. In upper part (a) you see the separate channels of the first stereo sound. It contains the same frequencies and beats in both channels. The beats are visible in the amplitude envelope: in the upper part the envelope starts at an extreme at the start, then falls to zero at a time of 0.1 s, then rises again to the extreme value at time 0.2 s, then falls again to zero at time 0.3 s and finally rises again to the extreme at the end of the sound. This is a relative slow variation of the amplitude of the underlying higher frequency sound and you will hear this amplitude variation as a beat. Another way of describing this sound starts by realizing that if we trace the envelope from its maximum at the start, going through the zero at time 0.1 s, then going to the minimum at time 0.2 s, and going up again through time 0.3 s to the end, then this envelope curve traces exactly one period of a cosine. The duration of the sound in the figure is 0.4 s, the beat frequency will therefore be $1/0.4 = 2.5$ Hz. In contrast with this, the channels

[13]The part `if row=1 then` tests if the predefined variable row equals 1. The equal sign = after the `if` expression is an equality test and is *not* an assignment. For more predefined variables see section C.1.1.

of the last sound, as displayed in part (b), only show two slightly different frequencies in the two channels. No sign of beats show up here!

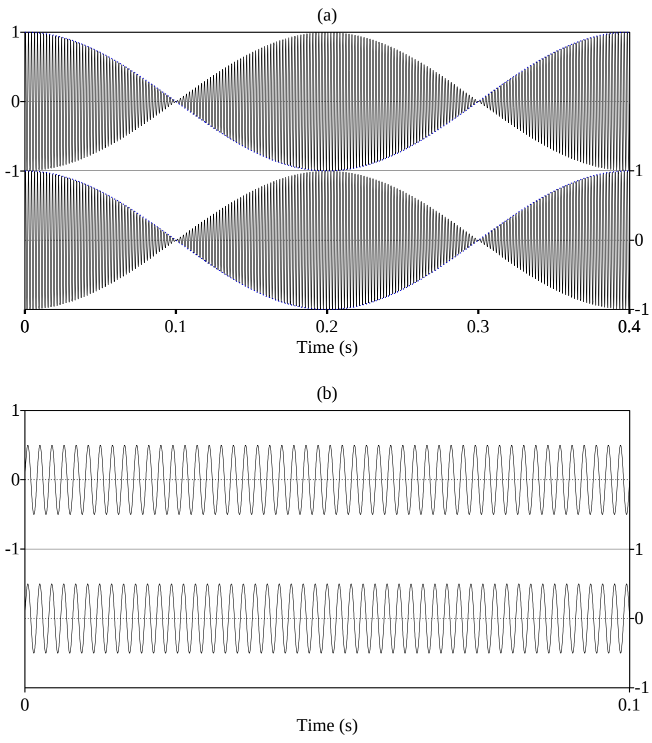


Figure 4.10.: The stereo channels for the sounds that (a) have beats in the signal and (b) generate beats in your brain.

4.7. Loops

With a conditional expression you can change the execution path in the script only once. Sometimes you need to repeat an action several times. In this section we will introduce a number of constructs that enable repetitive series of actions by reusing script lines.

4.7.1. For loops

Suppose you have to generate a lot of tones, all with frequencies that are related to each other. After generation of a tone we have to print the frequency value in the info window. Suppose the frequencies are the first five multiples of 100 Hz, i.e. 100, 200, 300, 400 and 500 Hz. To differentiate between the sounds in the object window we also name them with a text that shows their frequency. Given our knowledge so far, the first possibility that comes to mind will probably be:

```
Create Sound as pure tone: "100", 1, 0, 0.5, 44100, 100, 1, 0.01, 0.01
appendInfoLine: "The frequency was 100 Hz."
Create Sound as pure tone: "200", 1, 0, 0.5, 44100, 200, 1, 0.01, 0.01
appendInfoLine: "The frequency was 200 Hz."
Create Sound as pure tone: "300", 1, 0, 0.5, 44100, 300, 1, 0.01, 0.01
appendInfoLine: "The frequency was 300 Hz."
Create Sound as pure tone: "400", 1, 0, 0.5, 44100, 400, 1, 0.01, 0.01
appendInfoLine: "The frequency was 400 Hz."
Create Sound as pure tone: "500", 1, 0, 0.5, 44100, 500, 1, 0.01, 0.01
appendInfoLine: "The frequency was 500 Hz."
```

Note that the number in the name field is between double qoutes since thsi field needs text and not numbers. This seems like a perfect way to do it. However, we will show that it is possible to improve upon this little script. We will proceed in a number of small steps. First we note that each pair of lines differs *systematically* from the previous pair of lines at three places and it always involves the frequency number. Let first reduce two differences by introducing a new variable `f` for the frequency:

```
f=100
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01)
appendInfoLine: "The frequency was ", f, " Hz."
f=200
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01)
appendInfoLine: "The frequency was ", f, " Hz."
f=300
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01)
appendInfoLine: "The frequency was ", f, " Hz."
f=400
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01)
appendInfoLine: "The frequency was ", f, " Hz."
f=500
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01)
appendInfoLine: "The frequency was ", f, " Hz."
```

You might say: “This is not an improvement, the number of lines in the script has increased”. Yes, it has more lines, however, the *complexity* of the lines has been reduced. We note there is a lot of repetition in this script now. The lines that start with `do` and `appenInfoLine` are exactly repeated five times. Only the frequency assignment differs each time. In fact, the structure of the script can be viewed as repetition of *three* actions: (1) assign a value to `f`, (2) use `f` to create a sound and (3) print the `f` value. We now rewrite the frequency assignment a little bit and then we are ready for the shorthand notation: the **for-loop**. The for-loop repeats a number of statements a fixed number of times. Before explaining the syntax of the for-loop we make one extension to the script above by making the regularity in the frequency value `f` more explicit.

```
i = 1
f = i * 100
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
appendInfoLine: "The frequency was ", f, " Hz."
i = 2
f = i * 100
```

```
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
appendInfoLine: "The frequency was ", f, " Hz."
i = 3
f = i * 100
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
appendInfoLine: "The frequency was ", f, " Hz."
i = 4
f= i * 100
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
appendInfoLine: "The frequency was ", f, " Hz."
i = 5
f = i * 100
Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
appendInfoLine: "The frequency was ", f, " Hz."
```

The script states that each frequency `f` is a multiple of 100 Hz. However, the script has expanded now to twenty lines, four lines per tone. Let us first discuss this script in somewhat more detail and then get rid of it. Because of the five extra lines we now have a regular pattern of four lines of code that act in a similar way. The script starts with assigning a value 1 to the variable `i`; lets call this variable the *index* variable. In the following line the frequency variable `f` is calculated that depends on the value of the index variable `i`. The sound is created and its frequency value printed. In the next group of four lines the the only difference with the previous group is that the index variable i has increased by one. Consequently a new frequency value `f` is calculated, a new sound created and another frequency is printed. And so on for the next groups of four lines.

Now we are ready for the for-loop magic. The twenty-line script above can be simplified to the following five-liner:

```
for i from 1 to 5
    f = i * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

This final script is a substantial reduction in the number of lines even compared with the first script in this section. The groups of lines that were repeated in the previous script have been reduced to *one* instance only. The script expresses the repetition explicitly with a syntactical construct called the *for-loop*. The first and the last line in this script mark the *for-loop*: a for-loop starts with `for` and ends with `endfor`, both on separate lines. The code that has to be repeated is in the lines between the `for` and the `endfor`. The text, on the first line, that follows `for` specifies two things: (1) the *name* of the *index* variable and (2) the *successive* values that will be assigned to the index variable. What goes on in a for-loop?

- INITIALIZATION: The first item after the `for` expresses that the name of the index variable will be `i`;
- `i` will be assigned the value 1 because the `from 1` part says so;
  - CHECK: the interpreter checks if the value of `i` is smaller than or equal to 5 (this is the number specified by the `to 5` part). If `i` is larger than 5 the execution will proceed with the statement just *after* the `endfor`, in other words the execution *jumps out of the loop*. If $i \leq 5$, the next statement will be executed;

- ◦ `f` will get the value 100 (= 1*100); the sound will be created and the frequency value printed in the info window;
- ◦ at the `endfor` statement, the value of the index will be increased by 1, i.e. `i` will be 2 now;
- ◦ the execution now jumps to the CHECK label and the new value of `i` will be checked;
- ◦ f will get the value 200 (= 2*100); the sound will be created and the frequency value printed in the info window;
- ◦ at the `endfor` statement, the value of the index will be increased by 1, i.e. `i` will be 3 now;
- ◦ the execution now jumps to the CHECK label and the new value of `i` will be checked;
- ◦ f will get the value 300 (= 3*100); the sound will be created and the frequency value printed in the info window;
- ◦ `i = 4`; jump to CHECK; etc.
- ◦ `i = 5`; jump to CHECK; etc.
- ◦ `i = 6`; jump to CHECK;
- ◦ jump out of the loop because `i` is larger than 5 now;

- Continue execution after the `endfor`.

By now, the semantics of a for-loop should be fairly clear. We note that the name of the index variable in a for-loop can be freely chosen, as long as it confirms to a variable's name standard. A mere change in the name of index variable will not change the functionality of a script as the following script shows.

```
for k from 1 to 5
    f = k * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

Note that changing the index variable from `i` to `k` has to be done at two places here: (1) in the line that starts with `for` and (2) inside the loop where we use the index variable. The index variable's name doesn't have to be as short as the following equivalent script shows.

```
for a_very_long_index_variable_name from 1 to 5
    f = a_very_long_index_variable_name * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

A note on the extendability. As the script shows it is now very easy to increase the number of sounds by simply increasing the number after the `to`. The following script will generate 10 sounds, all harmonically related.

```
for ifreq from 1 to 10
    f = ifreq * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

And, may be unnecessary to add, the from and to numbers don't have to be fixed, they can be variables too.

```
ifrom = 1
ito = 10
for ifreq from ifrom to ito
    f = ifreq * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

By the way, a for-loop doesn't have to start at 1 as in the following example where 8 different sounds will be generated.

```
ifrom = 3
ito = 10
for ifreq from ifrom to ito
    f = ifreq * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

If the index starts at value 1, you can leave out the `from` part.

```
ito = 5
for ifreq to ito
    f = ifreq * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

The script above generates five sounds where the loop index `ifreq` starts at 1. You may guess by now why the following loop will never be executed.

```
ifrom = 6
ito = 5
for ifreq from ifrom to ito
    f = ifreq * 100
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

4.7.1.1. More variables: an array of variables

Suppose that just like in the previous section you have to generate a lot of sounds, and print out their frequencies. However, this time there appears to be no exploitable regularity in the frequencies. Say we have frequencies 111 Hz, 601 Hz, 277 Hz, 512 Hz and 213 Hz. Should we stick to the following script or is there a better way?

```
Create Sound as pure tone: "111", 1, 0, 0.5, 44100, 111, 1, 0.01, 0.01
appendInfoLine: "The frequency was 111 Hz."
Create Sound as pure tone: "601", 1, 0, 0.5, 44100, 601, 1, 0.01, 0.01
appendInfoLine: "The frequency was 601 Hz."
Create Sound as pure tone: "277", 1, 0, 0.5, 44100, 277, 1, 0.01, 0.01
appendInfoLine: "The frequency was 277 Hz."
Create Sound as pure tone: "512", 1, 0, 0.5, 44100, 512, 1, 0.01, 0.01
appendInfoLine: "The frequency was 512 Hz."
Create Sound as pure tone: "213", 1, 0, 0.5, 44100, 213, 1, 0.01, 0.01
appendInfoLine: "The frequency was 213 Hz."
```

As before, the extendability of this script is poor but since there is definitely a repetition going on, it would be nice to have something like the following.

```
for i from 1 to 5
    f = <a number that depends on the index i>
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

This script repeats 5 times the sequence of (1) an assignment, (2) a sound creation and (3) printing a line. The only thing not specified yet is how to get the right-hand side of the assignment for `f` as the loop index variable `i` varies from 1 to 5. A not very elegant first try might be:

```
for i to 5
    if i = 1
        f = 111
    elsif i = 2
        f = 601
    elsif i = 3
        f = 277
    elsif i = 4
        f = 512
    elsif i = 5
        f = 213
    endif
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

This is not elegant because the code is not shorter and, most importantly, not easily extendable. For example if we would have 6 frequencies we have to change the script at two places: the number 5 at line 1 has to become a 6 and we have to include 2 extra lines, one for the test `elsif i=6`, and one for the assignment. There is a better way. We can achieve the functionality of the above conditional expression with only one line of scripting by using an *array variable*. Lets first present the complete script and later explain what is going on.

```
freq[1] = 111
freq[2] = 601
freq[3] = 277
freq[4] = 512
freq[5] = 213
for i to 5
    f = freq[i]
    Create Sound as pure tone: string$ (f), 1, 0, 0.5, 44100, f, 1, 0.01, 0.01
    appendInfoLine: "The frequency was ", f, " Hz."
endfor
```

The first five lines in the script assign values to consecutive elements of the array variable `freq`. These lines are inevitable because no regularity can be exploited and nothing else remains than assigning each one of these values.[14] The interpretation of the assignment in the loop `f = freq[i]` is hopefully clear: at the start of the loop `i=1` and the first line in the loop translates to `f = freq[1]`, i.e. `f` will get the value of the first element in the array `freq`, which is 111. The corresponding sound will be generated in the next line. The next time in the loop, `i=2` and `f = freq[2]` will result in `f=601`, another sound will be created, and so on. Although the number of lines in the script has not decreased as compared to the first script in this section, the total number of characters has. Besides this, the structure of the script has improved. All frequencies are displayed clearly at consecutive lines and the sound creation command occurs only once, inside the loop. The extendability has improved too: for example, if we want to generate 10 sounds instead of 5 we only have to specify `freq[6]` to `freq[10]` and change the number 5 in the for-loop to a 10. Note that it would be o.k. to write the loop as:

```
for i to 5
    Create Sound as pure tone: string$ (freq[i]), 1, 0, 0.5, 44100, freq[i], 1, 0.01, 0.
    appendInfoLine: "The frequency was ", freq[i], " Hz."
endfor
```

where the variable `f` is not needed anymore and `freq[i]` has been substituted everywhere.

An array variable is not limited to numeric values. String arrays can also be defined like the following examples shows:

```
word$[1] = "This"
word$[2] = "is"
word$[3] = "a"
word$[4] = "sentence"
appendInfo: word$[1], " ", word$[2], " ", word$[3], " ", word$[4], "."
```

It would print in the info window: `This is a sentence.`

4.7.1.2. What goes on in a Formula...

We now try to make explicit what goes on in the formula part of the `Create sound from formula...` command. This one command performs two actions: (1) it allocates enough computer memory for the sound and (2) modifies this memory according to a formula. In the following script these two actions are split up into separate lines.

```
Create Sound from formula: "s", "Mono", 0, 0.5, 44100, "0"
Formula: <some double-quoted formula>
```

This script is completely equivalent to the following one-liner:

```
Create Sound from formula: "s", "Mono", 0, 0.5, 44100, <some double-quoted formula>
```

[14]Later on we will learn that these values don't have to be assigned in this particular way but could also be extracted from, for example, a supplied table of frequency values.

Before we go on, we repeat that a sound is represented in Praat as a matrix which means that sounds are stored as rows of numbers. A mono sound is a matrix with only one row and many columns. A stereo sound is a sound with two channels, each channel is represented in one row of the matrix. A stereo sound is therefore a matrix with two rows and both rows have the same number of columns. Each matrix cell contains one sample value. Whenever we want to use a formula on a sound we can think about a sound as a matrix. The position of any sample value in a sound can be indexed with a pair of numbers indicated as `[row,col]`, where `row` is the row number (i.e. channel number) and `col` is the column number (i.e. sample number). *Row numbers and column numbers always start at 1.* For example the element indexed by [1,2] is the second number from the first row. Each sample value in a row is indexed with a column number and represents the (average) value of the amplitude of the analog sound in a very small time interval, the *sampling period*. In general, the total *duration* of a sound is the *sampling period multiplied by the number of samples* in a row. To be able to calculate this duration for a sound, Praat keeps the necessary extra information, together with the rows with the sample values, in the Sound object itself. In a script you have access to this extra information because some predefined variables exist to access this information. The predefined variables for a sound have already been discussed in section 4.5.2.2. To refer to the interpretation of the column values of a matrix we have `xmin`, `xmax`, `x1`, `nx`, and `dx`. For the rows we have `ymin`, `ymax`, `y1`, `ny`, and `dy`, while the predefined variables `row` and `col` refer to the current row and column. Finally the variable `self` refers to the current element. Figure 4.11 gives an overview.

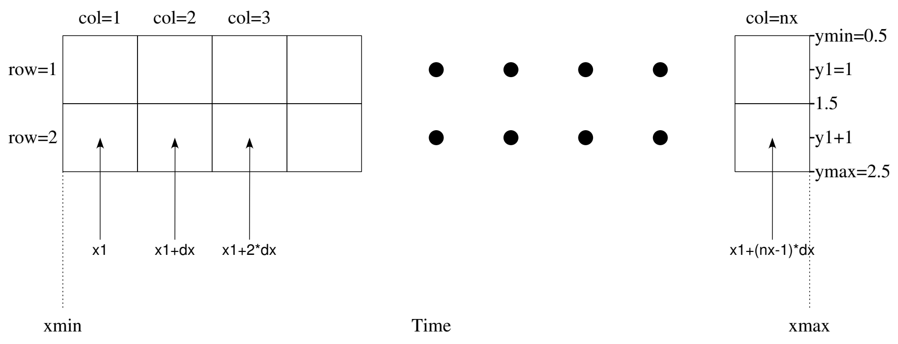


Figure 4.11.: A stereo sound as a matrix with two rows.

The first sample in a sound is at time `x1`. The second sample will be at a time that lies `dx` from the first sample, i.e. at `x1+dx`, the third sample will be another `dx` further away at `x1+2*dx`, et cetera. The last sample of the sound, this is also the last sample in a row, will be at time `x1+(nx-1)*dx`. The general equation to calculate the time that corresponds to the sample in column number `col` is therefore `x1+(col-1)*dx`.

The big picture now is that, for example for a mono sound, the `Formula... sin(2*pi*100*x)` command is expanded by Praat like this:

```
for col to nx
```

```
  xt = x1 + (col - 1) * dx
  self[1, col] = sin (2 * pi * 100 * xt)
endfor
```

the `self[1,col]` is the element at position `col` in row 1 of the sound.

For a stereo sound there is one extra loop for the number of rows `ny` and the `Formula... sin(2*pi*100*x)` will be executed as:

```
for row to ny
  for col to nx
    xt = x1 + (col - 1) * dx
    self[row, col] = sin (2 * pi * 100 * xt)
  endfor
endfor
```

Here the inner loop will be executed twice, first for `row` equal to 1 and then for `row` equal to 2. Therefore, for a stereo sound with 44100 Hz sampling frequency a formula will be evaluated 88200 = 2*44100 times.

The `Formula...` command is very powerful and can do a lot more things than we have shown here. The next sections will show you some more.

4.7.1.3. Modify a Matrix with a formula

With `Formula...` you can modify all data types that have several rows of numbers (matrices). The most often used data types in Praat that can be represented as a matrix are probably Sound, Spectrum and Spectrogram. As we saw in the previous section, things make more sense if you are aware of the implicit loops around the formula text. You can do very powerful things with `self` in a formula.

- Multiply the sound amplitudes with two.

  ```
  Formula: "self*2"
  ```

  If we multiply a sound by a number larger than 1 we always have to be careful that the amplitudes stay within the range from -1 to $+1$.

- Rectify a sound, i.e. make negative amplitudes positive.

  ```
  Modify: "if self < 0 then -self else self fi"
  ```

- Square the amplitude of the sound

  ```
  Formula: "self^2"
  ```

- Chop off peaks and valleys, i.e. simulate the effect of oversteering (see figure 3.4). The following script simulates clipping of amplitudes whose absolute value exceeds 0.5.

  ```
  Formula: "if self < -0.5 then -0.5 else self fi"
  Formula: "if self > 0.5  then  0.5 else self fi"
  ```

  Here two formulas are applied in succession. First all amplitudes smaller than -0.5 will be set to -0.5 by the first formula. Next the already modified sound will be modified again by the second formula and all amplitudes larger than 0.5 will be set equal to 0.5.

- Create a white noise sound.

```
Create Sound from formula: "white_noise", "Mono", 0, 1, 44100, "0"
Formula: "randomGauss(0,1)"
```

  White noise has equal spectral power per frequency bin on a linear frequency scale.

- Create a pink noise sound.

```
Create Sound from formula: "white_noise", "Mono", 0, 1, 44100, "0"
Formula: "randomGauss(0,1)"
To Spectrum: "no"
Formula: "if x > 100 then self*sqrt(100/x) else 0 fi"
To Sound
```

  Pink noise has equal spectral power per frequency bin on a logarithmic frequency scale.

4.7.1.4. Reference to other Sounds in Formula...

Suppose you have two sounds in the list of objects named s1 and s2 and you want to create a third sound that is the sample by sample average of the two. There are two ways to accomplish this in Praat. Both ways involve new syntax. The following examples will show you the difference between calculating *with* interpretation and *without* interpretation.

```
1 Create Sound from formula: "s1", "Mono", 0, 1, 44100, "0"
2 Formula: "sin(2*pi*500*x)"
3 Create Sound from formula: "s2", "Mono", 2, 3, 44100, "0"
4 Formula: "sin(2*pi*505*x)"
5 Create Sound from formula: "s3", "Mono", 0, 3, 44100, "0"
6 Formula: "(Sound_s1[] + Sound_s2[])/2"
7 Create Sound from formula: "s4", "Mono", 0, 3, 44100, "0"
8 Formula: "(Sound_s1(x) + Sound_s2(x))/2"
```

The script first creates the two sound objects s1 and s2. Line 1 creates the empty sound named s1 which starts at time 0 and lasts for 1 second. Line 2 modifies the s1 sound by changing it into a tone of 500 Hz. In line 3 sound s2 with a duration of 1 s is created, but now the starting time is at 2 s. In lines 5 and 7 we create two silent sounds s3 and s4, both start at 0 s and end at 3 s. These latter two sounds will receive the results of the averaging operations. The two fundamentally different ways to do the averaging are in lines 6 and 8 and involve using either [] or () for indexing.

1. Let us first magnify what happens in line 6 where the formula works on the selected sound s3:

```
for col to 3*44100
  self[1, col] = (Sound_s1[1, col] + Sound_s2[1, col]) / 2
endfor
```

   The sound s3 lasts 3 seconds, therefore the last value for col is 3*44100. The assignment in the loop to `self[1,col]` refers to the element at position `col` in the first row of the *selected* sound s3. The value assigned is the sum of two terms divided by 2. Each term involves new syntax and shows how to refer to data that is *not* in the current selected object! The first term, `Sound_s1[1,col]`, refers to the element at position `col` in the first

row of an object of type Sound with name s1. The second term refers to an element at the same position but now in an object of type Sound with name s2. This is a very powerful extension within the formula context, because it gives us possibilities to use information from other matrix type objects besides the selected one.
Therefore, in a `Formula...`, the syntax `Sound_s1[1,col]` and `Sound_s2[1,col]` refer to the element at position `col` in row 1 from a sound named s1 and a sound named s2, respectively.
The loop in more detail now. The first time, when col=1, the value from column 1 from sound s1 is added to the value from column 1 from the sound s2, averaged and assigned to the first column from the selected sound s3. Then, for col=2, the second numbers in the rows are averaged and assigned to the second position in the row of s3. This can go on until col reaches `1*44100+1` because then the numbers in the s1 and the s2 sound are finished, (they were each just one second of duration). Praat then assigns to the Sounds s1 and s2 zero amplitude outside their domains. Thus indexes that are out of their domain for a sound, like index 44101 is for s1 and s2, will be valid indexes but Praat will assign a zero amplitude. In this way, the final second and third seconds of s3 are filled with zeros, i.e. silence. When you listen to the outcome of the formula, i.e. sound s3, you will hear frequency beats just like you did in section 4.6.2 (but they last only one second now).

2. Now we proceed with the other summation. In line 8 the sounds are also summed but now instead of the square brackets we use parentheses. Inside parentheses the expression will evaluate to *real* time. We magnify what happens.

```
for col to 3*44100
  x = x1 + (col-1)*dx
  self[1,col]=(Sound_s1(x) + Sound_s2(x))/2
endfor
```

In the third line of this script, the two sounds are queried for their values *at a certain time*. Now the time domains of the corresponding sounds are used in the calculation. The domains of s1 and s2 are not the same, the domains don't even overlap. Just like in the previous case Praat accepts the non-overlapping domains and assumes the sounds to be zero amplitude outside their domains. The resulting sound s4 is now very different from sound s3.

The difference between the indexing of the sounds with [] versus () is very important. In indexing with [] the sounds were treated as a row of amplitude values. Amplitude values *at the same index* were blindly added, irrespective of differences in domains or differences in sampling frequencies.[15] In indexing with () the sounds are treated as functions of time and amplitude values of the sounds *at the same time* were added and averaged.

Only if sampling frequencies are equal and sounds start at the same time, the two methods result in the same output.

[15]Create sound s2 with a sampling frequency of 22050 Hz instead of 44100 and investigate the difference between the behavior of [] and () .

4.7.1.5. Use matrix elements outside a Formula context

If we want to access sound sample values outside the **Formula...** context we may also use the construct of the previous section. The following script averages the first two sample values of a sound.

```
Create Sound form formula: "s", "Mono",0, 1, 44100, "randomUniform(-0.9,0.9)"
mean12 = (Sound_a[1,1] + Sound_a[1,2])/2
```

The first line is only cosmetic because we need a sound object to refer to by name. In the second line we use the values of the first two cells in the first row of the sound matrix, we add them and divide by two. Note that we cannot make an assignment this way, the construct Sound_a[i,j] may only occur on the right-hand side of an equal sign. For assignments we need to use the "Modify → Set value at sample number..." command. Probably you will never have to change individual samples of a sound in this way. We will learn better ways to refer to objects later on in advanced scripting (you may already have thought "What happens if two objects bear the same name?").

4.7.2. Repeat until loops

The following scripts simulates the number of times we have to "throw" a pair of dice to reach a sum of twelve eyes.

```
throws = 0
repeat
    eyes = randomInteger(1,6) + randomInteger(1,6)
    throws = throws + 1
until eyes = 12
writeInfoLine: "It took ", throws, " trials to reach ", eyes, " with two dice."
```

The `randomInteger` function generates another random integer value in the range from 1 to 6 each time it is called and can threfore simulate the throwing of a dice. A repeat loop is used when the number of times a loop should be executed is difficult to calculate or is not certain beforehand. This means that the stop condition has to be calculated during execution of the loop.

4.7.3. While loops

Given a number m find n, the nearest power of two, such that $m \leq n$. For example if $m = 7$ then $n = 8$ since the nearest power of 2 is 8 $(= 2^3)$. If $m = 9$ then $n = 16 = 2^4$. This little algorithm is used to find the size of the buffer we need to perform a fast Fourier transform on m sample values (we use the shorthand notation `n*=2` for `n = n * 2`).

```
n = 1
while n < m
  n *= 2
endwhile
```

Just like the repeat loop this construction is used if we don't know beforehand how often the loop has to be executed. The difference between a while and a repeat loop is that in the while loop the exit condition is tested *before* the execution of the statements in the loop and that the repeat loop always executes the statements in the loop at least once.

4.8. Functions

For scripting you can use specialized units of code that *process an input* and then *return a value*. In Praat you have a number of these so called builtin *functions* at your disposal. Functions can be categorized into two groups: functions that operate on numbers and functions that operate on a text or string value. The built-in functions that operate on numbers are called *mathematical* functions, while the others are call *string* functions. Functions may return a number or a string. Those that return a string value have a name that also ends with a $-sign. In some of the functions the input domain is limited, for example if we ask for the square root of a number the number must be positive otherwise the result is undefined. If we happen to call the square root function with a negative argument the outcome will be assigned a special undefined value which will print as `-- undefined --`.

4.8.1. Mathematical functions in Praat

abs (x) absolute value. After running the following script

```
a = abs(-2)
b = 3 + abs(3)
writeInfoLine: "*abs: ", a, " ", b, "*"
```

the info window will show: `*abs:  2 6*`

round (x) returns the nearest integer.

```
a = round (1.5)
b = round (-1.5)
c = round(1.1)
writeInfoLine: "*round: ", a, " ", b, " ", c, "*"
```

The info window will show: `*round:  2 -1 1*`

floor (x) returns the highest integer value not greater than `x`.

```
a = floor (1.5)
b = floor (-1.5)
c = floor(1.1)
writeInfoLine: "*floor: ", a, " ", b, " ", c, "*"
```

The info window will show: `*floor:  1 -2 1*`

ceiling (x) returns the lowest integer value not less than `x`.

```
a = ceiling (1.5)
b = ceiling (-1.5)
c = ceiling(1.1)
writeInfoLine: "*ceiling: ", a, " ", b, " ", c, "*"
```

The info window will show: *ceiling: `2 -1 2*` sqrt (x) returns the number y such that $y^2 = x$

```
a = sqrt(4)
b = 0.5 * sqrt(2)
c = sqrt (-2)
writeInfoLine: "*sqrt: ", a, " ", b, " ", c, "*"
```

The info window will show: `*sqrt:  2 0.7071067811865476 --undefined--*`

min (x, ...) returns the minimum of a series of numbers.

```
a = min(1, -2, 4, 6, 7.6)
writeInfoLine: "*min: ", a, "*"
```

The info window will show: `*min:  -2*`

max (x, ...) returns the maximum of a series of numbers.

```
a = max(1, -2, 4, 6, 7.6)
writeInfoLine: "*max: ", a, "*"
```

The info window will show: `*max:  7.6*`

imin (x, ...) returns the location of the minimum in a series of numbers.

```
a = imin(1, -2, 4, 6, 7.6)
writeInfoLine: "*imin: ", a, "*"
```

The info window will show: `*imin:  2*`

imax (x, ...) returns the location of the maximum in a series of numbers.

```
a = imax(1, -2, 4, 6, 7.6)
writeInfoLine: "*imax: ", a, "*"
```

The info window will show: `*imax:  5*`

sin (x) returns the sine of x.

```
a = sin(0)
b = sin(pi)
writeInfoLine: "*sin: ", a, " ", b, "*"
```

The info window will show: `*sin:  0 1.2246063538223773e-16*`
Due to finite precision, sin(pi) will not show a value of zero.

cos (x) returns the cosine of x.

```
a = cos(0)
b = cos(pi)
writeInfoLine: "*cos: ", a, " ", b, "*"
```

The info window will show: `*cos:  1 -1*`

tan (x) returns the tangent of x

```
a = tan(0)
b = tan(pi/4)
writeInfoLine: "*tan: ", a, " ", b, "*"
```

The info window will show: `*tan:  0 0.9999999999999999*`
Because of finite precision the last value is not exactly 1.

arcsin (x) returns the inverse of the sine; `arcsin(x)=y` means `x=sin(y)` or more directly `arcsin(sin(y))=y`.
The last form makes clear that the input x must be in the the domain [-1,1]. The output will always be in the range $[-\pi/2, \pi/2]$.

```
a = arcsin(1)
b = arcsin(-1)
writeInfoLine: "*arcsin: ", a, " ", b, "*"
```

The info window will show: `*arcsin:  1.5707963267948966 -1.5707963267948966*`
(these numbers are $\pi/2$ and $-\pi/2$).

arccos (x) returns the inverse of the cosine.
The input x must be in the the domain [-1,1] while the output will be in the range $[0, \pi]$.

```
a = arccos(1)
b = arccos(-1)
writeInfoLine: "*arccos: ", a, " ", b, "*"
```

The info window will show: `*arccos:  0 3.141592653589793*`

arctan (x) returns the inverse of the tangent.
The input x can be any real number while the output range will be limited to $[-\pi/2, \pi/2]$
.

```
a = arctan(0)
b = 4 * arctan(1)
writeInfoLine: "*arctan: ", a, " ", b, "*"
```

The info window will show: `*arctan:  0 3.141592653589793*`

arctan2 (y, x) returns the angle (in radians) between the positive x-axis and the point given by the coordinates (x, y) on it. The angle is positive for counter-clockwise angles (upper half-plane, y > 0), and negative for clockwise angles (lower half-plane, y < 0). The output will be in the range $[-\pi, \pi]$.

```
a = arctan2(1,1)
b = arctan2(1,-1)
c = arctan2(-1,-1)
d = arctan2(-1,1)
writeInfoLine: "*arctan2: ", a, " ", b, " ", c, " ", d, "*"
```

The info window will show: `*arctan2:  0.7853981633974483 2.356194490192345 -2.356194490192345 -2.356194490192345*`
(these numbers are $\pi/4$, $3\pi/4$, $-3\pi/4$, $-\pi/4$).

sinc (x) returns the sinus cardinalis `sin(x)/x`.

```
a = sinc(0)
b = sinc(pi/2)
writeInfoLine: "*sinc: ", a, " ", b, "*"
```

The info window will show: `*sinc: 1 0.6366197723675814*`

sincpi (x) returns sinus cardinalis sin(πx)/(πx).

```
a = sincpi(0)
b = sincpi(1/2)
writeInfoLine: "*sincpi: ", a, " ", b, "*"
```

The info window will show: `*sincpi: 1 0.6366197723675814*`

exp(x) returns the exponent e^x.

```
a = exp(0)
b = exp(-1)
c = exp(1)
writeInfoLine: "*exp: ", a, " ", b, " ", c, "*"
```

The info window will show: `*exp: 1 0.36787944117144233 2.718281828459045*`

ln (x) returns the natural logarithm of `x` for x $\geq$ 0.

```
a = ln(e)
b = ln (2)
c = ln(10)
writeInfoLine: "*ln: ", a, " ", b, " ", c, "*"
```

The info window will show: `*ln: 1 0.6931471805599453 2.302585092994046*`

log10 (x) returns the logarithm of `x` base 10.

```
a = log10(e)
b = log10(2)
c = log10(10)
writeInfoLine: "*log10: ", a, " ", b, " ", c, "*"
```

The info window will show: `*log10: 0.4342944819032518 0.3010299956639812 1*`

log2 (x) returns the logarithm of `x` base 2.

```
a = log2(e)
b = log2(2)
c = log2(10)
writeInfoLine: "*log2: ", a, " ", b, " ", c, "*"
```

The info window will show: `*log2: 1.4426950408889634 1 3.3219280948873626*`

sinh (x) returns the hyperbolic sine $(e^x - e^{-x})/2$.

```
a = sinh(0)
b = sinh(1)
writeInfoLine: "*sinh: ", a, " ", b, "*"
```

The info window will show: `*sinh:  0 1.1752011936438014*`

cosh (x) returns the hyperbolic cosine $(e^x + e^{-x})/2$.

```
a = cosh(0)
b = cosh(1)
writeInfoLine: "*cosh: ", a, " ", b, "*"
```

The info window will show: `*cosh:  1 1.5430806348152437*`

tanh (x) returns the hyperbolic tangent sinh(x)/cosh(x).

```
a = tanh(0)
b = tanh(1)
writeInfoLine: "*tanh: ", a, " ", b, "*"
```

The info window will show: `*tanh:  0 0.7615941559557649*`

arcsinh (x) returns the inverse hyperbolic sine $\ln(1 + \sqrt{x^2 + 1})$.

```
a = arcsinh(0)
b = arcsinh(1)
c = arcsinh(-10)
writeInfoLine: "*arcsinh: ", a, " ", b, " ", c, "*"
```

The info window will show: `*arcsinh:  0 0.8813735870195429 -2.998222950297976*`

arccosh (x) returns the inverse hyperbolic cosine $\ln(1 + \sqrt{x^2 - 1})$, where the input range is limited to x $\geq$ 1.

```
a = arccosh(1)
b = arccosh(10)
writeInfoLine: "*arccosh: ", a, " ", b, "*"
```

The info window will show: `*arccosh:  0 2.993222846126381*`

arctan (x) returns the inverse hyperbolic tangent: $\frac{1}{2} \ln \frac{1+x}{1-x}$, where the input range is limited to $-1 \leq x \leq 1$.

```
a = arctanh(0)
b = arctanh(1/2)
writeInfoLine: "*arctanh: ", a, " ", b, "*"
```

The info window will show: `*arctanh:  0 0.5493061443340549*`

sigmoid (x) returns the value of `1/(1+exp(-x))`, which will be a number between 0 and 1.

```
a = sigmoid(0)
b = sigmoid(1)
c = sigmoid(-1)
writeInfoLine: "*sigmoid: ", a, " ", b, " ", c, "*"
```

The info window will show: `*sigmoid: 0.5 0.7310585786300049 0.2689414213699951*`

invSigmoid (x) returns a value y such that `sigmoid(y)=x` or `invSigmoid(sigmoid(x))=x`. The input range is $0 < x < 1$.

```
a = invSigmoid(0.5)
b = invSigmoid(0.9)
c = invSigmoid(0.1)
writeInfoLine: "*invSigmoid: ", a, " ", b, " ", c, "*"
```

The info window will show: `*invSigmoid: 0 2.1972245773362196 -2.197224577336219*`

erf (x) returns the error function $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. The total area under the error function equals 1.

```
a = erf(1)
b = erf(2)
c = erf(3)
writeInfoLine: "*erf: ", a, " ", b, " ", c, "*"
```

The info window will show: `*erf: 0.8427007929497149 0.9953222650189527 0.9999779095030014*`

erfc (x) return the complement of error function $1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$.

```
a = erfc(1)
b = erfc(2)
c = erfc(3)
writeInfoLine: "*erfc: ", a, " ", b, " ", c, "*"
```

The info window will show: `*erfc: 0.1572992070502851 0.004677734981047265 2.209049699858544e-05*`

hertzToBark (x) transforms acoustic frequency to Bark-rate according to $7\ln(x/650+\sqrt{1+(x/650)^2})$, where `x` is frequency in hertz. The solid line in figure 4.12 displays this function on the frequency interval from 0 to 10 kHz.

barkToHertz (x) returns the inverse of the previous function $650\sinh(x/7)$.

hertzToMel (x) transforms acoustic frequency to perceptual pitch as $550\ln(1+x/550)$, where `x` is frequency in hertz. The dotted line in figure 4.12 displays this function on the frequency interval from 0 to 10 kHz.

melToHertz (x) inverse of previous function. Transforms mel to acoustic frequency as $550(e^{x/550}-1)$, where `x` is frequency in mels.

hertzToSemitones (x) from acoustic frequency to a logarithmic musical scale, relative to 100 Hz: $12\ln(x/100)/\ln 2$.
Examples: hertzToSemitones(100)=0, hertzToSemitones(200)=12, hertzToSemitones(400)=24.

semitonesToHertz (x) The inverse of the above: $100e^{x\ln 2/12}$.

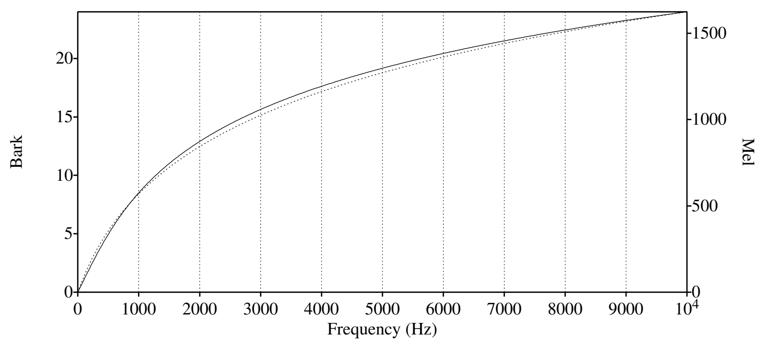


Figure 4.12.: The solid line displays the function `hertzToBark` while the dotted line displays the function `hertzToMel`.

erb (x) the perceptual equivalent rectangular bandwidth (ERB) in hertz, for a specified acoustic frequency in hertz as $6.23 \cdot 10^{-6}x^2 + 0.09339x + 28.52$.

hertzToErb (x) from acoustic frequency to ERB-rate as $11.17 \ln(\frac{x+312}{x+14680} + 43)$.

erbToHertz (x) from ERB-rate to acoustic frequency in hertz as $\frac{14680d-312}{1-d}$, where $d = e^{(x-43)/11.17}$.

4.8.2. String functions in Praat

In the previous section a large number of mathematical functions of Praat were enumerated. In this section we will concentrate on Praat's string functions which accept strings as arguments. We start counting characters in a string at 1, i.e. the first character has index 1.

length (a$) returns the length of the string `a$` as a number.

```
length = length ("This is a string with spaces!")
writeInfoLine: "*lenght: ", length, "*"
```

The info window will show: `*lenght:  29*`
As you see, variables can have the same name as functions without confusing the interpreter.

left$ (a$, n) return a string with the first n characters of the string `a$`.

```
b$ = left$ ("This is a string with spaces!", 4)
c$ = left$ (b$, 10)
writeInfoLine: "*left$: ", b$, "*", c$, "*"
```

The info window will show: `*left$:  This*This*`

right$ (a$, n) returns a string with the rightmost n characters of the string `a$`.

```
b$ = right$ ("This is a string with spaces!", 7)
c$ = left$ (b$, 10)
d$ = right$ (b$, 10)
writeInfoLine: "*left$: ", b$, "*", c$, "*", d$, "*"
```

The info window will show: `*left$:  spaces!*spaces!*spaces!*`

mid$ (a$, start, number) returns the substring starting at position `start` counting `number` characters.

```
b$ = mid$ ("This is a string with spaces!", 6, 2)
c$ = mid$ ("This is a string with spaces!", 1, 4)
d$ = mid$ (c$, 10, 2)
writeInfoLine: "*mid$: ", b$, "*", c$, "*", d$, "*"
```

The info window will show: `*mid$:  is*This**`

index (a$, b$) returns the index of the first occurrence of string b$ in a$.

```
b = index ("This is a string with spaces!", "is")
c = index ("This is a string with spaces!", "piet")
writeInfoLine ("*index: ", b, " ", c, "*")
```

The info window will show: `*index:  3 0*`

rindex (a$, b$) returns the index of the last occurrence of string b$ in a$.

```
b = rindex ("This is a string with spaces!", "is")
c = rindex ("This is a string with spaces!", "s")
writeInfoLine: "*rindex: ", b, " ", c, "*"
```

The info window will show: `*rindex:  6 28*`

replace$ (a$, fs$, rs$, n) returns a modified version of the string `a$` where at most `n` occurrences of the string `fs$` are replaced with the string `rs$`. If n equals zero all occurences are replaced.

```
b$ = replace$ ("This is a string with spaces!", " ", "", 0)
c$ = replace$ (b$, "with", "without", 1)
writeInfoLine: "*replace$: ", c$, "*"
```

The info window will show: `*replace$:  Thisisastringwithoutspaces!*`

index_regex (a$, re$) returns the index where the regular expression `re$` first matches the string `a$`. Regular expressions offer the possibility to search for patterns instead of literal occurences as in the previous functions. The **Help > Regular expressions** in Praat will show you the syntax and many possible uses of regular expressions. The first line in the folowing script searches for uppercase characters while the second line searches for whitespace.

```
b = index_regex ("This is a string with spaces!", "[A-Z]")
c = index_regex ("This is a string with spaces!", "\s")
writeInfoLine: "*index_regex: ", b, " ", c, "*"
```

The info window will show: `*index_regex:  1 5*` rindex_regex (a$, re$) return the last match of the regular expression `re$` within the string `a$`.

replace_regex$ (a$, fs$, rs$, n) returns a modified version of the string `a$` where at most `n` matches of the regular expression `fs$` are replaced by the pattern `rs$`. If n equals zero all occurrences are replaced. The first line doubles each character from the `a$` string while the second line replaces double characters in the b$ string by single ones.

```
b$ = replace_regex$ ("hello", ".", "&&", 0)
c$ = replace_regex$ (b$, "(.)\1","\1", 0)
writeInfoLine: "*replace_regex$: ", b$, "*", c$, "*"
```

The info window will show: `**replace_regex$:  hheellllloo*hello*`

fixed$ (number, precision) return a string with `number` formatted with `precision` digits after the decimal point.

```
b$ = fixed$ (5.1236, 3)
c$ = fixed$ (0.0001234, 2)
writeInfoLine: "*fixed$: ", b$, "*", c$, "*"
```

The info window will show: `*fixed$:  5.124*0.0001*`

percent$ (number, precision) as fixed$ but with a percentage sign.

```
b$ = percent$ (0.123456, 3)
c$ = percent$ (0.0001234, 2)
writeInfoLine: "*percent$: ", b$, "*", c$, "*"
```

The info window will show: `*percent$:  12.346%*0.01%*`

date$ () gives the date and time in a fixed format.

```
date$ = date$ ()
writeInfoLine: "*date$: ", date$, "*"
```

The info window will show e.g.: `*date$:  Tue Mar 1 19:48:42 2011*`

extractNumber (sentence$, precursor$) returns the number in `sentence$` after the `precursor$` string.

```
b = extractNumber ("Lucky number 7", "number")
c = extractNumber ("Lucky number", "number")
writeInfoLine: "*extractNumber: ", b, " ", c, "*"
```

The info window will show: `*extractNumber:  7 --undefined--*`
As you may have noted the second line request a number where none is present in the input sentence.

extractWord (sentence$, precursor$) returns the word following the `precursor$` string from `sentence$`. If no word can be found an empty string will be returned.

```
b$ = extractWord$ ("Lucky number 7", "Lucky")
c$ = extractWord$ ("Lucky number", "number")
writeInfoLine: "*extractWord$: ", b$, "*", c$, "*"
```

The info window will show: `**extractWord$:  number**`

extraxtLine (sentence$, precursor$) returns the rest of `sentence$` after `precursor$`.

```
b$ = extractLine$ ("This is a string with spaces!", "is")
writeInfoLine: "*extractLine$: *", b$, "*"
```

The info window will show: `*extractLine$:  * is a string with spaces!*`

4.9. The layout of a script

The layout of a script is important for *you*. It enables *you* to see more easily the structure of the script. Layout is not important for Praat, it just doesn't care: as long as the script text is syntactically correct, Praat will run the script.

You are allowed, for example, to add additional comments in the script text that may describe in your terms what is going on. The more easily you can identify what is going on in a script the more easily you can check the semantic correctness of the script. The following elements may help you to structure the script so your intentions become clear.[16]

- White space, i.e. spaces and tabs. White space at the beginning of a script line is ignored in Praat.[17] You can use whitespace to help you see more easily the structure of your script. For example in conditional expression you should indent every line between the `if` and the `endif`. In all the examples we presented, white space was used to show the structure of the script. See for example the scripts in section 4.6. However, see also section 4.10 for possible pitfalls.

- Besides white space for laying out the structure you can use comments. Comments are lines that start with "#", "!" or ";". Make comments useful, they should not repeat what is already completely clear in the script. For example, the comment in the following script is useless.

```
# add 2 to a
a = a + 2
```

- Continuation lines start with three dots (...). Use continuation lines to split a long line into several shorter lines.

- Split off sections of code into procedures.

4.10. Mistakes to avoid in scripting

The text that you use in a script for Praat is not completely free but has to conform to a syntax and also has semantics. Three of the most occurring errors in Praat are listed below, the first

[16]For masters of the C programming language there is a yearly contest to accomplish exactly the *opposite*. Programmers try to write the most obscure code possible. For some really magnificent examples, see the website of The International Obfuscated C Code Contest at http://www.ioccc.org.

[17]There are other computer languages like Python in which white space is part of the syntax of the language.

two are syntax errors while the third one is a semantic error. We anticipate a little bit on things not yet explained. They will become more clear later on.

- Praat commands in scripts have to be spelled exactly right. Text on menu options and buttons are Praat commands and, as you can easily check, they always starts with an uppercase character. For example, if you want to play a sound from a script and type `play` instead of `Play` you will receive a message "Command "play" not available for the current selection" and the script will stop running.
- Sometimes during copy-paste actions or other edit actions, accidentally extra white space may creep in, mostly in the form of blank space. For example if you write `do ("Play ")`, with a blank space after the "y" instead of the correct "Play", you receive a message like "Command "Play " not available for current selection". Another often occurring error is that you have written a blank space before the final colon of a command. If you run the following script an error message is displayed that starts with "Command "Create Sound from formula :" not available for current selection."

  ```
  Create Sound from formula : "s", 1, 0, 1, 44100, "sin(2*pi*300*x)"
  ```

  This error is easily corrected by removing the blank space before the colon.
- A command may exist but is not valid for the selected object. For example, in your script you wanted to "Play" a sound but accidentally selected a Spectrum object. In the following script, Praat will show the error message "Command "Play" not available for current selection, Script line 3 not performed or completed: « Play »".

  ```
  Create Sound from formula: "s", 1, 0, 1, 44100, "sin(2*pi*300*x)"
  To Spectrum: "no"
  Play
  ```

  One way to get this script right is to move the `Play` command one line up to the second line.
- Be careful when copying scripts via programs like Word or Acrobat reader because characters that may seem visually correct in the script editor may use a different underlying representation that might confuse the script interpreter.

5. Pitch analysis

In Praat a Pitch object represents periodicity candidates as a function of time. The periodicity may refer to acoustics, perception or vocal fold vibrations. The standard pitch algorithm in Praat tries to detect and to measure this periodicity and the algorithm to do so is described in Boersma [1993]. In this chapter we will elaborate on this algorithm and give somewhat more background information on the steps involved.

The concept of pitch, however, is not as simple as we stated above, because, pitch is a subjective psycho-physical property of a sound. The ANSI[1] definition of pitch is as follows:

> Pitch is that auditory attribute of sound according to which sounds can be ordered on a scale from low to high.

Pitch is a sensation and the fact that pitch is formed in our brain already hints that it will not always be simple to calculate it. The definition implies that essentially the calculation of pitch has to boil down to one number; numbers can be ordered from low to high. Actually, the only simple case in measuring pitch is for the pitch associated with a pure tone: a pure tone always evokes the same pitch sensation within a normal-hearing listener. This was experimentally verified by letting subjects adjust the frequency of a tone to make its pitch sound equal to the pitch of a test tone. After many repetitions of the experiment and after averaging over many listeners, the distribution of all the subjects measured pitches shows only one peak, centered at the test frequency. For more complex sounds, distributions with more than one peak may occur. Various theories about pitch and pitch perception exist and a nice introduction is supplied for example by Terhardt's website.

The topic of this chapter, however, is not pitch *perception* but pitch *measurement*. A large number of pitch measurement algorithms exist and new ones are still being developed every year. We will describe the pitch detector implemented in Praat because it is one of the best available.

5.1. Praat's pitch algorithm

Praat's pitch analysis can be divided into two steps.

1. Finding the pitch candidates and their strengths in each analysis frame. For complex sounds, human pitch perception does not always result in a mono-valued distribution but in a distribution with many peaks. Any real pitch detection algorithm in principle should replicate this behavior and therefore always produce a list of possible pitch candidates. The algorithm for the pitch analysis of a sound follows the general analysis

[1] American National Standards Institute, see `http://www.ansi.org`.

scheme discussed in section 2.4 as shown in figure 2.9 by analyzing overlapping windowed segments from the sound. The analysis of each segment results in an analysis frame with a number of pitch candidates together with an indication of each candidate's strength. The pitch candidates are determined with a signal analysis technique called autocorrelation and this part of the algorithm will be explained in section 5.1.1.

2. Find the best pitch candidate for each analysis frame. At each time step we have several candidates, finding the best candidate at each time step is equivalent, as we will see later, to finding a best path through the candidate "space". Finding the best path is performed with a technique called *Viterbi* and will be explained in section 5.1.4.

Before we can explore the details of the pitch algorithm we first need to be familiar with a new signal analysis concept: the autocorrelation function. If you are not familiar with autocorrelation than check out section A.12, where we introduce functions that correlate one or more sounds.

5.1.1. Finding pitch candidates by autocorrelation

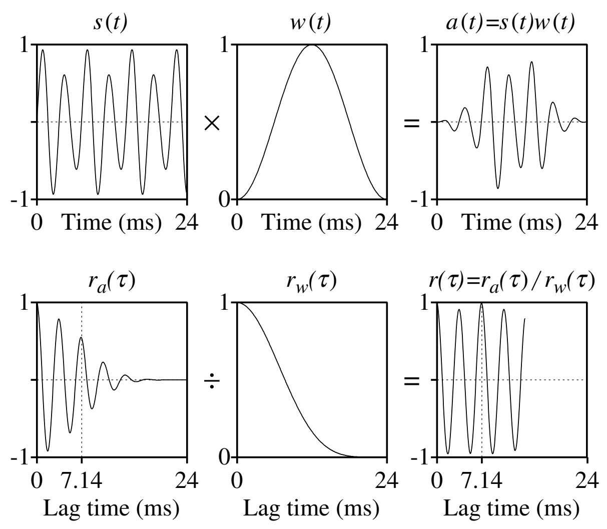


Figure 5.1.: The pitch determination using the autocorrelation of a windowed speech segment.

In this section we will explain how the autocorrelation is used for finding pitch candidates in each analysis frame. As we already know, pitch analysis conforms to the general analysis scheme as was laid out in section 5.1, and therefore the same analysis is applied to each consecutive analysis frames of the sound. If we explain how one (analysis) frame is analysed we know how the complete sound will be analysed.

By definition the best candidate for the pitch can be found at the maximum peak of the autocorrelation. However, *windowing* and *sampling* may cause accuracy problems during determination of the positions and the amplitudes of the maxima of the autocorrelation. The pitch determination algorithm is summarized in figure 5.1 for a speech-like sound of the form $s(t) = (1 + 0.3\sin(2\pi 140t))sin(2\pi 280t)$. This is a minimal periodic sound with a 140 Hz fundamental frequency and a "formant" at 280 Hz. This special sound was chosen for the demonstration because it is periodic and the 280 Hz component interferes with the 140 Hz component making correct pitch determination difficult by pitch algorithms that use the standard autocorrelation method to determine the correct pitch. The top left panel in the figure clearly shows that approximately three periods are visible during the 24 ms of the analysis window and that its fundamental frequency therefore is 140 Hz, which corresponds to a period of 7.14 ms. The steps involved in determining the pitch of this sound by autocorrelation are the following:

1. The first step in any type of analysis is the application of a window function $w(t)$ on the analysis frame $s(t)$. This operation is shown at the top row of the figure where the signal $s(t)$ on the left is multiplied by the window function $w(t)$ in the middle to result in the windowed signal $a(t) = s(t)w(t)$ at the right.

2. The normalized autocorrelation $r_a(\tau)$ of $a(t)$ is calculated (and shown at the bottom left panel).

3. The normalized autocorrelation $r_w(\tau)$ of the window function $w(t)$ is calculated (and shown at the middle panel of the bottom row).

4. The important new step follows now: we divide *correct* autocorrelation of the windowed signal by the autocorrelation of the window. This results in the signal $r(\tau) = \frac{r_a(\tau)}{r_w(\tau)}$ shown at the bottom right. If we compare the autocorrelation before and after the correction, as the left and right panels at the bottom of the figure show, it is immediately clear why this step is absolutely essential: the maximum in the corrected autocorrelation $r(\tau)$ is now at a lag time of 7.14 ms and therefore corresponds to the correct pitch of 140 Hz, while the maximum in the uncorrected autocorrelation $r_a(\tau)$ is at 3.07 ms, half the previous value and therefore correspond to an incorrect pitch of 280 Hz. By applying the correction for the windowing function, the amplitude at the correct lag was boosted enough to be the largest.

5. In the final part we select the positions τ_i and the amplitudes $r(\tau_i)$ of the first n local extrema in the corrected autocorrelation $r(\tau)$. Now the $1/\tau_i$ are the pitch candidates frequencies and the $r(\tau_i)$, always a number between 0 and 1, are the strengths of the candidates. The closer $r(\tau_i)$ is to one the stronger the candidate is.

The steps above result in an autocorrelation that is corrected for the disturbing effects of windowing. The second disturbing effect is due to the sampling of the sound. Sampling affects both the *precision* of the location of a local maximum as well as its *amplitude*. We have a local maximum r_m in the autocorrelation function at the m-th sample if $r_m > r_{m-1}$ and $r_m > r_{m+1}$. A simple estimate for the pitch would then be $F_0 \approx 1/(mT)$, where T is the sampling time. This would, however, be a crude approximation of the pitch because the precision is not so well as we will show now. For a sampling frequency of 10 kHz the samples in the sound are at distances of 0.0001 s apart. This will also be the distance between the lag values in the autocorrelation of this sound. The pitch value that corresponds to a lag at position $m = 31$ will be $1/(32 \cdot 0.0001) \approx 312.5$ Hz. For positions $m = 33$ and $m = 34$ the corresponding pitches will be 303.0 and 294.1 Hz, respectively. These three pitches differ approximately 9 Hz from each other. Therefore, near 300 Hz the only possible pitches are limited to the values 312.5, 303.0 and 294.1 Hz and this is not precise enough. In general we like to have a better estimate of the pitch that is not limited by the sampling. This is possible by using *interpolation*. Interpolation is the estimation of values in between sample points based on some assumption about how the amplitude varies. To get a sufficiently accurate interpolation for normal pitch ranges, parabolic interpolation[2] would suffice. However, with parabolic interpolation the autocorrelation *amplitudes* could still turn out to be very wrong. As the normalized autocorrelation amplitude correspond to the pitch strength of this candidate parabolic interpolation can not be used. Because the autocorrelation is a sampled signal, and the correct interpolation for a sampled signal is a "sinc" interpolation, the pitch algorithm will use sinc interpolation to find both the correct position as well as the correct amplitude of the local maxima of the autocorrelation.[3] The precision achieved with sinc interpolation is phenominal. With an analysis width of 40 ms, the frequency of a sine tone of 3777 Hz, sampled at 10 kHz sampling frequency, can be determined as accurately as 3777.0001 Hz. If you take a look at the sampled representation of this tone this will impress you even more.[4]

In this section we have explained how the pitch candidates in each analysis frame are calculated. In the following section we will explain what parameters are involved in the pitch algorithm

5.1.2. Parameters of the pitch algorithm

Figure 5.2 shows the form that appears if you select the "Sound: To Pitch (ac)..." button. With the parameters in the form you can fine tune the two steps involved in calculating pitch: finding the candidates in each frame and finding the best global pitch value in each frame. The parameters for finding the candidates are:

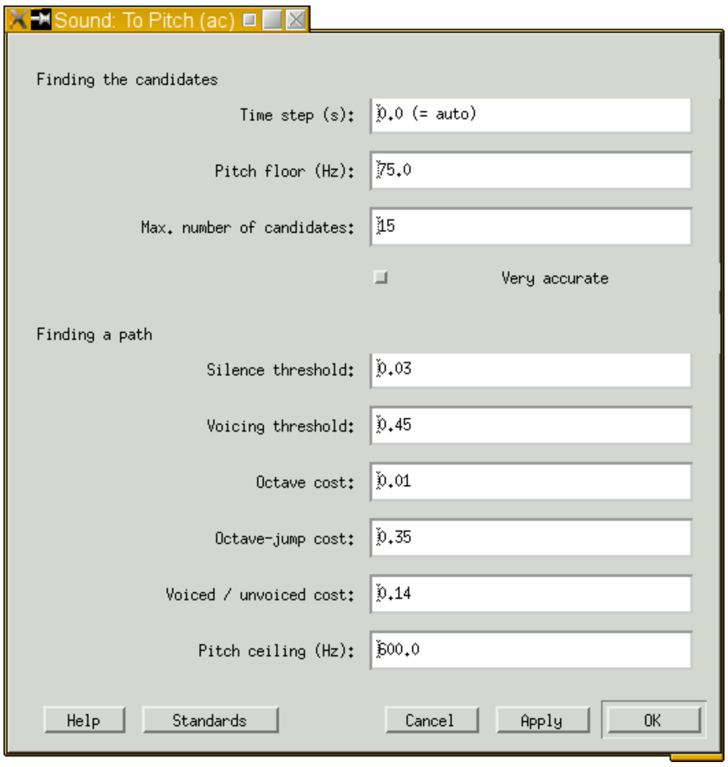
Time step *(standard value: 0.0 s)* The time between different measurements (see also section 2.4). If you supply the standard value of 0.0 Praat will choose an appropriate time step which will be 0.75 / (pitch floor). A pitch floor of 75 Hz then results in a time step of 0.01 s and for each second of the sound 100 pitch values will be calculated.

Pitch floor *(standard value: 75 Hz)* The lowest candidate frequency to consider. The pitch floor parameter directly determines the effective length of the analysis window. For periodicity detection we need a minimum of three periods in an analysis window. The lower the pitch, the longer a period will be and the longer the analysis window length needs to be. Three periods of a 75 Hz periodic signal last $3/75 = 0.04$ s. To resume,

[2]See section A.10 on interpolation.

[3]A local maximum can be found by a algorithm specialized for this purpose. In the successive approximation of this extremum a sinc interpolation is used to calculate amplitudes. See section A.3 for a mathematical description of the sinc function.

[4]You can create such a tone as "Create Sound from formula... s Mono 0 1 10000 sin(2*pi*3777*x)"



Finding the candidates

Time step (s):	0.0 (= auto)
Pitch floor (Hz):	75.0
Max. number of candidates:	15
☐	Very accurate

Finding a path

Silence threshold:	0.03
Voicing threshold:	0.45
Octave cost:	0.01
Octave-jump cost:	0.35
Voiced / unvoiced cost:	0.14
Pitch ceiling (Hz):	600.0

Figure 5.2.: The Sound: To Pitch (ac)... form with parameter defaults.

if the algorithm needs to detect pitches as low as 75 Hz then we need an analysis window that has at least 40 ms duration. If you want to go lower, for example to measure creaky voice you could lower the floor to say 50 or 60 Hz; for a 60 Hz parameter value, the length of the analysis window will be $3/60 = 50$ ms. For female voices you could probably increase this value to say 100 Hz.
If the time step parameter is 0.0, the pitch floor also determines the time step. At 75 Hz an analysis window of 40 ms and a time step of 10 ms amount to four times oversampling.

Max. number of candidates *(standard value: 15)* determines the number of local maxima in the autocorrelation that have to be remembered.

Very accurate determines the window function. If *off,* a Hanning window is used with the same duration as the analysis window. If *on,* a Gaussian window is selected of twice the effective window length duration. Although the analysis window length is doubled, the effective width of the Gaussian window is half this width.

After the candidates have been calculated a post-processing algorithm seeks the best candidates for the global pitch asssignment. The post-processing tries to find the cheapest path to connect the “best” pitch value in a frame with the “best” value in the next frame. The following parameters determine the cheapest path.

Silence threshold *(standard value: 0.03).* Sound frames in which the largest amplitudes do not exceed this value, relative to the global maximum peak, are considered as silent and therefore voiceless.

Voicing threshold *(standard value: 0.45)* determines whether a frame is considered voiceless or not. If the strengths of all candidates in a pitch frame do not exceed this value, the frame is marked as voiceless. If you increase the voicing threshold more frames will be marked voiceless.

Octave cost *(standard value: 0.01 per octave)* determines how much high frequencies are favored above low frequencies. This parameter is necessary to force a decision for perfectly periodic signals. A sine of frequency F will show maxima in the autocorrelation at lag times $1/F$, $2/F$, $3/F$, ..., that correspond to the pitches F, $F/2$, $F/3$, ... Besides the correct pitch F all the undertones are candidates too because all these maxima have the same autocorrelation amplitude. The octave cost parameter gives the highest candidate an advantage above the others.

Octave-jump cost *(standard value: 0.35)* determines the degree of disfavoring pitch changes. This parameter effects the choice going from one frame to the next. By giving pitch changes a penalty large frequency jumps are suppressed.

Voiced / unvoiced cost *(standard value: 0.14)* determines the degree of disfavoring voiced/unvoiced transitions. Increasing this value decreases the number of voiced/unvoiced transitions. This parameter is necessary to suppress an accidental strong local voiceless candidates within an otherwise voiced part or, the opposite: suppress an accidental strong voiced candidate within an otherwise voiceless part of the speech sound.

Pitch ceiling *(standard value: 600 Hz).* Candidates above this frequency will be ignored. For males voices you could lower the ceiling to, say, 300 Hz.

5.1.3. How are the pitch strengths calculated?

In this section we explain in somewhat more detail how some of the parameters above are used in the determination of the candidate strengths. The candidate that is always present is the voiceless one whose strength is detemined as

$$R = \mathit{voicingThreshold} - \max\left(0, 2 - \frac{(local\,peak)/(global\,peak)}{silenceThreshold/(1 + voicingThreshold)}\right).$$

The strength of the voiceless candidate depends on the voicing threshold, the silence threshold and the quotient of a frame’s local peak and the global peak. Mind you, the peaks we are talking about, now refer to the peaks in the oscillogram (and not to the peaks in the autocorrelation). By chosing the silence threshold as zero, the local peak will not influence the pitch strength anymore. For this special case the strength of the voiceless candididate will be $R = voicingThreshold$ and R is now only determined by the voicing threshold.

For the voiced candidates, as was noted above, one of the functions of the octave cost parameter was to give an advantage to higher frequencies. The strength of the i-th candidate $R(\tau_i)$ at lag time τ_i will be modified due to the octave cost parameter in the following way:

$$R(\tau_i) = r(\tau_i) - octaveCost \cdot {}^2\log(minimumPitch \cdot \tau_i).$$

Because a candidate at lag time τ_i has a frequency of $F_i = 1/\tau_i$ Hz, we can translate the formula above to frequencies:

$$R(F_i) = r(F_i) - octaveCost \cdot {}^2\log\left(\frac{minimumPitch}{F_i}\right).$$

Both formulas show that since $minimumPitch/F_i \leq 1$ for all candidates, the logarithm of this value turns out to be zero or negative and this always results in a positive (or zero) contribution to the autocorrelation value $r(\tau_i)$. Therefore, $R(F_i) \geq r(F_i)$, with equality only when a candidate frequency happens to be equal to the minimum pitch. If $\tau_i > \tau_j$, i.e. $F_i < F_j$, it holds that $R(\tau_i) < R(\tau_j)$. Because the highest frequency has the lowest lag and therefore receives the largest contribution due to the octave cost correction, this will result in the largest pitch strength for the highest frequency candidate.

Another important use of this parameter is to be able to modify the decision point between acoustic pitch versus perceived pitch. For example for modulated sounds $s(t) = (1 + d_{\text{mod}} \sin(2\pi Ft)) \sin(2\pi 2Ft)$ with a so called modulation depth d_{mod}, the acoustic pitch is F while the perceived pitch is $2F$ for values of d_{mod} smaller than 0.2 or 0.3.

You can judge your pitch assignment as a function of the modulation depth with the following script 5.1. If you want *Praat's* pitch algorithm to switch from pitch F to pitch $2F$ at a modulation depth d_{mod} then you have to set the octave cost parameter at a value of d^2_{mod}. For example, if you want a pitch assignment of $2F$ for a modulation depth of 0.2 set $octaveCost = 0.04$.

5.1.4. Using Viterbi to find the best candidates

After the autocorrelation step we have pitch frames that store one unvoiced pitch candidate and one or more voiced candidates and their strengths. For each pitch frame we now have to decide what the best candidate is. Simply picking the one with the largest pitch strength in each frame will not always lead to a correctly assigned global pitch because no considerations about continuity are involved since in this case all pitch decisions are local. This may lead to a very discontinuous global pitch. It is also not yet clear what to do if two candidate strengths turn out to be equal. To solve these problems and some more, the pitch algorithm associates a cost function to each pitch candidate. Assigning cost functions will pose the problem in the domain of path search for which excellent algorithms to find the optimal path exist. the optimal path is the path with minimum global costs. In the previous section we have introduced a *within-frame* "cost" function that modifies the candidates strengths, and only depends on the candidates within a frame. To guarantee a smooth curve we also have to associate costs that inhibit large frequency pitch changes between two successive frames. We call these costs *between-frame* costs. If F_1 is a pitch candidate in a frame and F_2 is a pitch candidate in the next frame than a transition cost can be defined as

Script 5.1 Script to test your perceived pitch as a function of modulation depth.

```
form Modulation depth
  positive Depth 0.1
  positive F 150
 endform
dp = 100*depth
f1 = Create Sound from formula: "sf1", "Mono", 0, 0.5, 44100,
  ... "0.5*sin(2*pi*f*x)"
@fade_in_out
s = Create Sound from formula: "s'dp'", "Mono", 0, 0.5, 44100,
  ... "0.5*(1+depth*sin(2*pi*f*x))*sin(2*pi*2*f*x)"
@fade_in_out:
f2 = Create Sound from formula: "sf2", "Mono", 0, 0.5, 44100,
   ... "0.5*sin(2*pi*2*f*x)"
@fade_in_out
select all
Play
Remove

procedure fade_in_out
  Fade in: "All", -1, 0.005, "no"
  Fade out: "All", 100, -0.005, "no"
endproc
```

$$
transitionCost(F_1, F_2) = \begin{cases} 0 & \text{if } F_1 = F_2 \\ voiced\,/\,unvoiced\,cost & \text{if } F_1 = 0 \text{ xor } F_2 = 0 \\ (octave\,jump\,cost) \cdot \left| {}^2\log\left(\frac{F_1}{F_2}\right) \right| & \text{if } F_1 \neq 0 \text{ and } F_2 \neq 0. \end{cases}
$$

As the formula shows no transition costs are involved for candidates whose frequencies are equal, it is only when the frequency changes or when voicing changes that costs are involved.

Now, how do the strengths R_{kp} of the p-th candidate in the k-th frame and these transition costs work together? Before we explain this we first start with a definition: a *path* is a chain of connections between pitch candidates in successive frames. A path can starts at any candidate in the first frame and stop at any candidate in the last frame. In figure 5.3 we show for six successive pitch analysis frames the candidates. A figure like this, where an analysis is unfolded in time slices is called a trellis.[5] Of all the possible paths, only two paths are drawn, one with a solid line and one with a dotted line. Each path corresponds to a possible global pitch assignment. In general with m possible candidates in a frame and n pitch frames, the number of possible paths is m^n, which will be a very very large number even for moderate values of m and n. To give you an indication, for a standard pitch analysis of a 1 s duration signal with fifteen candidates per frame and hundred frames per second there are approximately $15^{100} \approx 4 \cdot 10^{117}$ possible paths. This number of paths is too large to handle by any computer, now, and in the future. Even if each atom in the universe were a computer with a 1 GHz clock and would have started computing at the start of the universe they still would not have finished tracking all possible paths when in the far future the sun has died out and doesn't shine anymore. It is clear that we need an algorithm that can improve on the above

[5]We skipped the start and end node in the trellis.

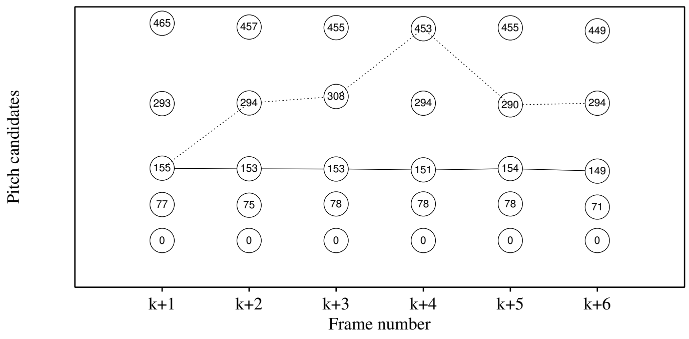


Figure 5.3.: Pitch trellis.

search time. Luckily the *Viterbi* algorithm can, it reduces the exponential time complexity from $O(m^n)$ to complexity $O(n \cdot m^2)$. For the example above, this amounts to a reduction in operations from the order of $4 \cdot 10^{117}$ to some 22500. This number is very manageable. Let us now show how Viterbi works.

The underlying model in the Viterbi algorithm is that the most likely path (which leads to a particular state) up to a certain point t must depend only on the observed pitches at point t and the most likely sequence of states which leads to that state at point $t - 1$. In other words we only use neighbouring frames in the calculation and there is no explicit depence on frames that are more than one time step in the past. A trivial, but necessary, condition for the pitch calculation is that the times associated with successive states in the path are strictly increasing, i.e. a path always goes from left to right. The following description of the Viterbi algorithm is rephrased from the wikipedia article on the Viterbi algorithm. The Viterbi algorithm operates on the state machine assumption. That is, at any time the pitch being modeled is in one of a finite number of states. Each state is characterized by a pitch candidate's frequency and strength. While multiple sequences of states (i.e. paths) can lead to a given state, at least one of them is the most likely path to that state, called the “winning path". This is a fundamental assumption of the algorithm because the algorithm will examine all possible paths leading to a state and only keep the one most likely. This way the algorithm does not have to keep track of all possible paths, but only one per state. If we unfold our pitch analysis as a trellis like we did in figure 5.3, the idea of *path* becomes clear: as a connection of states successive time points.

A second key assumption is that a transition from a state to the next state is accompanied by transition costs. The transition costs are computed from the candidates frequencies and strengths. The third key assumption is that we can add all the state to state transition costs to some cumulative cost. So the crux of the algorithm is to store the cumulative costs in each state. The algorithm examines moving forward to a new state by combining the cumulative costs of the all possible previous states with the local transition costs and chooses the transition which amounts in the smallest cumulative cost. The local transition cost, i.e. the costs involved in going from one state to the next, depends on the pitch candidate's frequencies and strengths in the old state and the new state. After computing the combinations of local costs and accumulated costs, only the best transition survives and all other paths are discarded. If we also store a pointer to the optimal previous state we can, after reaching the final state, make a trace back and so find the winning path, i.e. the path with the smalles accumulated cost.

5.2. Pulses in the SoundEditor

The pulses in the sound editor are a model of the excitation of the vocal tract. Each pulse models a new excitation and locally the distance between pulses correspond to the inverse of the local pitch. Therefore if the local pitch in an interval is 100 Hz, we can assign pulses in the interval that are 0.01 s apart. These pulses are *not* related to the moments of glottal opening or glottal closure. If it appears like that it is by mere coincidence because no algorithm is used for glottal closure or opening detection. Implementing a glottal closure and opening detection algorithm is something to be left for the future. In short: pulses only reflect the local pitch.

5.2.1. Challenges: Tone languages

5.2.1.1. Mandarin Chinese

Mandarin Chinese is a tone language, which means that tone is phonemic and used to distinguish words which would otherwise be homonyms. In Mandarin one considers five tones: the high tone, the rising tone, the low tone, the falling tone and the neutral tone.

6. Intensity analysis

In chapter 2 the propagation of sound waves was rudimentary treated. The most important conclusion was that sound is an wave phenomenon that results from air pressure variations. The sound pressure level is a measure of this air pressure variation. The air pressure is measured in pascal units (Pa), which are newtons per square metre (N/m²)[1]. The ambient presure is about 100,000 Pa. The amount by which the lungs can vary this air pressure is only some 200 to 1000 Pa. Outside your body, the air pressure caused by your speech is much smaller again, namely some 0.01 to 1 Pa at one metre from your lips. These values are comparable to the values that you see for a typical speech recording in Praat's sound editor. Although the amplitude of a sound in the sound editor is expressed in Pa these amplitudes can only be interpreted as *true* air pressures after applying a *sound pressure calibration*. Because this scaling effects all sound amplitudes in the same way we normally are not interested in this scaling and forget about it[2].

A normative human ear can detect a root-mean-square air pressure as small as 0.00002 Pa for a 1000 Hz pure tone. The *sound pressure level* (SPL) is generally expressed in decibel (dB) relative to this normative threshold:

$$\text{SPL} = 10 \log \left(\frac{1}{t_2 - t_1} \frac{\int_{t_1}^{t_2} x^2(t) dt}{\left(2 \cdot 10^{-5}\right)^2} \right),$$

where $x(t)$ is the sound pressure in Pa as a function of time and t_1 and t_2 are the times between which the energy is averaged. Essentially this formula says first to sum the squared amplitude values then divide this sum by the squared normative value and finally to take the logarithm of the outcome and multiply it by ten.

The Intensity object in Praat represents an intensity contour at linearly spaced time points with values in dB SPL, i.e. dB values relative to $2 \cdot 10^{-5}$ Pa. This intensity analysis is performed according to the general analysis scheme in figure 2.9, i.e. the sound is divided in overlapping segments and the intensity of each separate windowed segment is determined by the formula above.

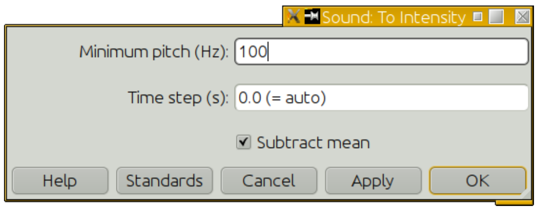
6.1. Sound: To Intensity...

In Praat the intensity analysis is performed by the command "Sound: To Intensity...". The form that appears is displayed in figure 6.1. The most inportant parameter in this form is the minimum pitch parameter as it determines the length of the windowed segment in the

[1]See Help > sound pressure level.

[2]You can read Help> sound pressure callibration if you like to know how to do the scaling.

calculation of the intensity values. If you set this parameter too high you will end up with pitch-synchronous intensity modulations.If you set it too low, the intensity contour will look smeared. If you want a sharp contour you should set it as high as possible. The effective window length that Praat calculates from this parameter is 3.2/*minimumPitch*. This guarantees that for a periodic signal with fundamental frequency equal to F_0 the intensity contour will hardly show any ripple if the minimum pitch is chosen equal to the fundamental frequency value F_0.



Sound: To Intensity

Minimum pitch (Hz): 100

Time step (s): 0.0 (= auto)

[✓] Subtract mean

Figure 6.1.: The Sound: To Intensity... form.

To show the influence of the minimum pitch smoothing parameter we have displayed in figure 6.2 the result of an intensity analysis on a short word segment for various values of the minimum pitch parameter.

In the top panel you see the waveform of the Dutch word /vrou/ as spoken by w male speaker. As can be seen, the initial voiced fricative /v/ is realised as unvoiced (/f/) and the vowel part is clearly voiced. In the bottom panel, from top to bottom, the result of the intensity analyses are displayed for minimum pitch values of 100, 200, 400 and 800 Hz. To be able to better visually compare these contours, each following contour was vertically shifted by an extra 5 dB. The intensity curve at the top, i.e. where the minimum pitch parameter was chosen as 100 Hz, is very smooth and seems to nicely follow the envelope of the sound. The 200 Hz contour that lies just below the previous one, shows almost the same features although it is a little bit more ragged. Increasing the minimum pitch value to 400 Hz makes the contour definitely ragged and in the voiced part of the word a ripple that varies with the individual pitch periods becomes clearly visible. Because of this large value of the minimum pitch the unvoiced parts in the contour also become rippled, although more irregularly because of lack of periodicity. When the minimum pitch is increased even further to 800 Hz, the ripples grow larger and also the number of ripples keeps increasing as the bottom curve shows. At the same time, while the ripples start increasing as the minimum pitch increases, the intensity better follows local amplitude changes. So there is a tradeoff between the smoothness of the curve and the posibility to follow local amplitude changes: the higher the minimum pitch the better we can follow the intensity contour details. From figure 6.2 it seems that a minimum pitch value between 100 and 200 Hz is adequate for this sound under normal circumstances. In this way the average pitch of the sounding interval, which is aroud 120 Hz, is in the 100 to 200 Hz inteval for the minimum pitch. We further note from the bottom pane that the lower the minimum pitch the further the curve starts to the right. This is because the lower the minimum pitch is, the longer the duration of each analysis window / segment is and the further away the midpoint of the first window is from the start of the signal. At the end we have the same effect, the larger window duration makes that the midpoint of the last analysed segment is now further away from the endpoint.

6.2. Intensity queries

Some queries of the intensity contour are the following:

Get maximum: returns the maximum value of the intensity contour in the chosen time range. In general default parabolic interpolation technique is more than sufficient. In general the maximum pitch parameter used in the determination of the intensity contour has a much larger influence on this value than the interpolation technique used.

Get minimum: returns the minimum value of the intensity contour in the chosen time range.

Get time of maximum: returns the time at which the maximum of the intensity contour occurs in the chosen time range.

6.3. Silence detection

Because the intensity object represents a weighted average of a sound's intensity it can be used to determine which parts of a sound have less intensity than other parts. Therefore, one of the uses of the intensity analysis is as the front end to a silence detector. Absolute silent parts will almost never occur in a sound: there will always be tiny background noises present that prevents the sound amplitude to attain large streches of zero amplitude values[3] . This means that if we want to detect stretches of silence in a sound we have to define the silence intensity level. The "Intensity: To TextGrid (silences)..." command in Praat is used for silence detection; its form is displayed in figure 6.3. The parameters of the form are the following:

Silence threshold (dB) is a relative threshold. It determines the maximum silence intensity with respect to the maximum intensity. Intervals that have intensities this number of dB's below the maximum intensity are considered voiceless. For example, if the maximum intensity value happens to be 78.2 dB and if the silence threshold is -25 dB then all intensity values that are below 53.2 dB ($= 78.2 - 25$) are considered as silent.

[3]The only way to achieve this would be to artificially make parts of a sound zero.

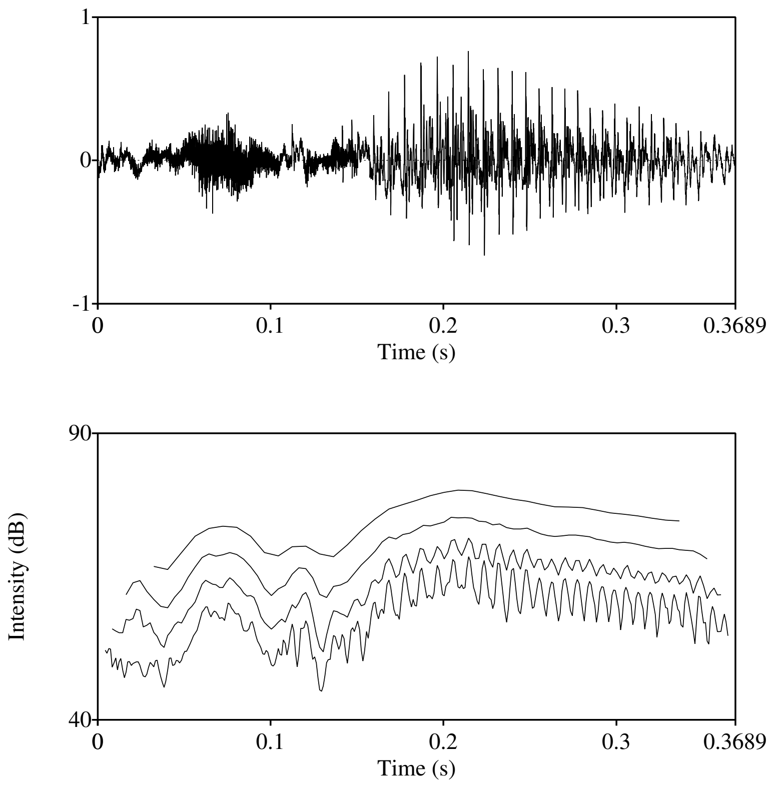
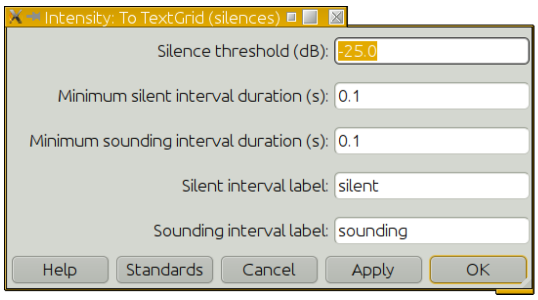


Figure 6.2.: The relation between the intensity curve and the minimum pitch parameter. In the top panel we show the oscillogram of the word /vrou/ as spoken by a male speaker. The bottom panel shows the intensity curves for different values of the minimum pitch parameter. From top to bottom the minimum pitch was 100, 200, 400 and 800 Hz, respectively. For displaying purposes only, intensity curves were vertically shifted by 5 dB.



Silence threshold (dB):	-25.0
Minimum silent interval duration (s):	0.1
Minimum sounding interval duration (s):	0.1
Silent interval label:	silent
Sounding interval label:	sounding

Figure 6.3.: The Intensity: To textGrid(silences)... form.

7. The Spectrum

One of the types of objects in Praat is the spectrum. The spectrum is an invaluable aid in studying differences between speech sounds. Almost all analyses that compare sounds, are based on spectra. The spectrum is a *frequency-domain* representation of a sound signal; the spectrum gives information about frequencies and their relative strengths. The other representation of a sound, the one we are already familiar with from the sound object in Praat, is the *time-domain* representation, i.e. the representation of sound amplitude versus time in an oscillogram.

A spectrum and a sound are different. A sound you can hear, a spectrum not. The spectrum is a (mathematical) construct to represent a sound for easier analysis. One makes calculations with a spectrum, one visualizes aspects of a spectrum but you can not hear it or touch it. Only after you have synthesized the sound from the spectrum, can you listen to the sound. The reason for the popularity of the spectrum is that it is often easier to work with than the sound. When the spectrum is calculated from a sound, a mathematical technique called *Fourier analysis* is used. A Fourier analysis finds all the frequencies in the sound and their amplitudes, i.e. their strengths. There is no information loss in the spectrum: we can get the original sound back from it by *Fourier synthesis*. These two transformations, analysis and synthesis, that are each others inverse, are visualized in figure 7.1. On the left we see a very small part of a sound as a function of *time* and on the right the sound as a function of *frequency*. The top arrow going from the sound to the spectrum, labeled "To Spectrum", visualizes the Fourier analysis. The bottom arrow, labeled "To Sound", visualizes Fourier synthesis. Although intuitively the spectrum is a simple object, i.e. a representation of the frequency content of a signal, the mathematics to calculate the spectrum from a sound is not simple. The main causes for mathematical complications are first of all the *finite duration* of the sound and secondly the fact that sounds are *sampled* in the time domain.

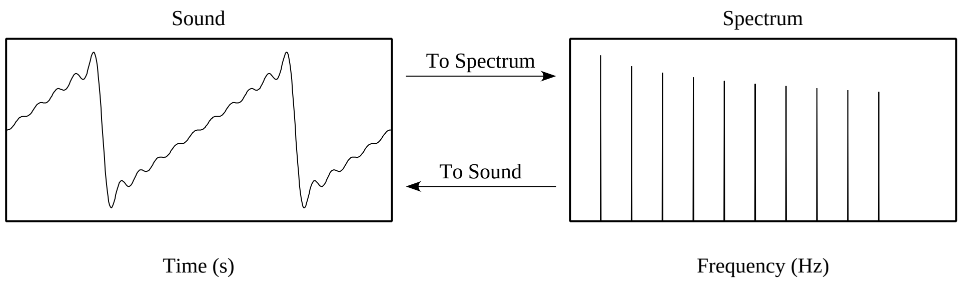


Figure 7.1.: The reversibility in Fourier analysis and Fourier synthesis.

Some terminology: instead of Fourier analysis one often talks about applying *a Fourier* *transform* and instead of Fourier synthesis one often says applying an *inverse Fourier transform*.

The spectrum is not a simple object like a mono sound but a complex one. Complex has a double meaning in this respect. The first meaning of complex is "composed of two or more parts". There are two parts in a spectrum: one part represents the *amplitudes* of all the frequencies and the other part the *phases* of the frequencies. The other meaning of complex is the mathematical one from "complex number".[1] This is about how the two aspects of a frequency, its amplitude and its phase, are represented. To visualize a complete spectrum we would need three dimensions: one for frequency, one for amplitude and one for phase. Three dimensional representations are difficult, we therefore limit ourselves to the most popular two dimensional representation: the amplitude spectrum, where vertically amplitude is displayed in decibel and horizontally frequency in hertz. Often the amplitude spectrum is *visualized* in text books in two different ways: as a *line spectrum* with vertical lines, or as an *amplitude spectrum* where instead of showing the vertical lines, the tips of the lines are connected. In the sequel we will show that what is visualized as a line spectrum only occurs for very special sound signals. In Praat the amplitude spectrum is always drawn, although for special combinations of tone frequencies and tone durations, the amplitude spectrum may have the appearance of a line spectrum. The most important reason for the popularity of the amplitude spectrum is that the human ear is not very sensitive to the relative phases of the components of a sound, the relative amplitudes of the component are of far more importance as an example in section 7.1.9 will show.

In the following sections we will first try to explain qualitatively the relation between a sound and its spectrum. We start to vary elementary signals and notice the effects in the spectrum. Then of course complexer signals will follow...

7.1. The spectrum of elementary signals

In this section we will explore the amplitude spectrum of pure tones. A pure tone can be described mathematically as a function of time as

$$s(t) = a\sin(2\pi f t), \tag{7.1}$$

where a is the tone's amplitude, f is its frequency and t is the time. Section A.1 gives a mathematical introduction to sine and cosine functions and in section (2.2.1) we showed by we can write a pure tone in the above form. Here we will not be concerned about how the spectrum is actually calculated from the sound, this is saved for a later section. Neither will we go into the details about how a spectrum is represented in Praat. We will start by just studying plots of the amplitude spectrum. The amplitude spectrum gives a graphical display of the most important part of the spectrum: the (relative) strengths of the frequency components in the spectrum. The amplitude spectrum shows on the horizontal axis frequency and on the vertical axis a measure of the amplitude of that frequency. We talk about two different meanings of amplitude here: the amplitude of a pure tone and the strength of a spectral component. Note that the *sound amplitude* and the *amplitudes in the spectrum* in general bear *no* relation. In

[1] See the appendix A.16 on complex numbers.

what follows we will see that *only for pure tones* there is a direct relation between the sound amplitude and the spectrum amplitude.

7.1.1. The spectrum of pure tones of varying frequency

In this section we investigate what the spectrum of pure tones looks like when we only vary the frequency and leave their amplitude constant. The following script (7.1) creates a pure tone, calculates the tone's spectrum and plots the tone and the spectrum next to each other in the same row.[2]

Script 7.1 Create a pure tone and spectrum and draw both next to each other.

```
a = 1
f = 100
Create Sound from formula: "100", "Mono", 0, 1, 44100, "a*sin(2*pi*f*x)"
Select outer viewport: 0, 3, 0, 3
Draw.: 0, 0.01, -1, 1, "yes", "Curve"
To Spectrum: "no"
Select outer viewport: 3, 6, 0, 3
Draw: 0, 500, 0, 100, "yes"
# Draw the marker on the right...
```

The figures in the three rows of figure 7.2 were all made with amplitude $a = 1$ while we chose three different values for the frequency f. Note that we use amplitude for two things The first plot in the top row on the left shows the first ten milliseconds of a pure tone with a frequency f of 100 Hz. The figure shows exactly one period of this tone within this interval because period and frequency are inversely related: for a frequency of $f = 100$ Hz, one period of the tone lasts $T = 1/f = 0.01$ seconds. The plot on the right shows the tone's *amplitude spectrum*. On the horizontal axis, the frequency range has been limited from 0 to 500 Hz for a better overview. The vertical scale is in dB/Hz (see section A.4.2 on decibel). There is only one vertical "line" in the amplitude spectrum. The line starts at the horizontal axis at position 100 Hz and rises to a value of 91 dB/Hz.[3] This line signals that in the amplitude spectrum there is only one frequency component present at a frequency of 100 Hz with an amplitude of 91 dB/Hz. [4]

The next row in the figure shows on the left the first ten milliseconds of a 200 Hz pure tone, also with an amplitude of one. We can now distinguish two periods because a tone of frequency $f = 200$ Hz has a period of $T = 1/200 = 0.005$ seconds and two of these periods fit in the plot interval of 0.01 seconds. The amplitude spectrum of this tone, again on the right, shows only one vertical line. This line signals that in the amplitude spectrum there is only one frequency component present at a frequency of 200 Hz with an amplitude of 91 dB/Hz.

[2]In section 7.1.7 we will explain why we calculate the spectrum from a signal with a duration of one second while we only draw 10 milliseconds.

[3]The spectral amplitude of 91 dB/Hz occurs because of the sound amplitude being 1 and the duration of the tone being 1 second. Had we chosen another fixed sound amplitude and/or duration then the spectral amplitude would have been a different number.

[4]Although, on the frequency scale presented, the spectrum looks like a line spectrum, zooming in will reveal that it actually is a very thin triangle. Nevertheless this we will still call it a line.

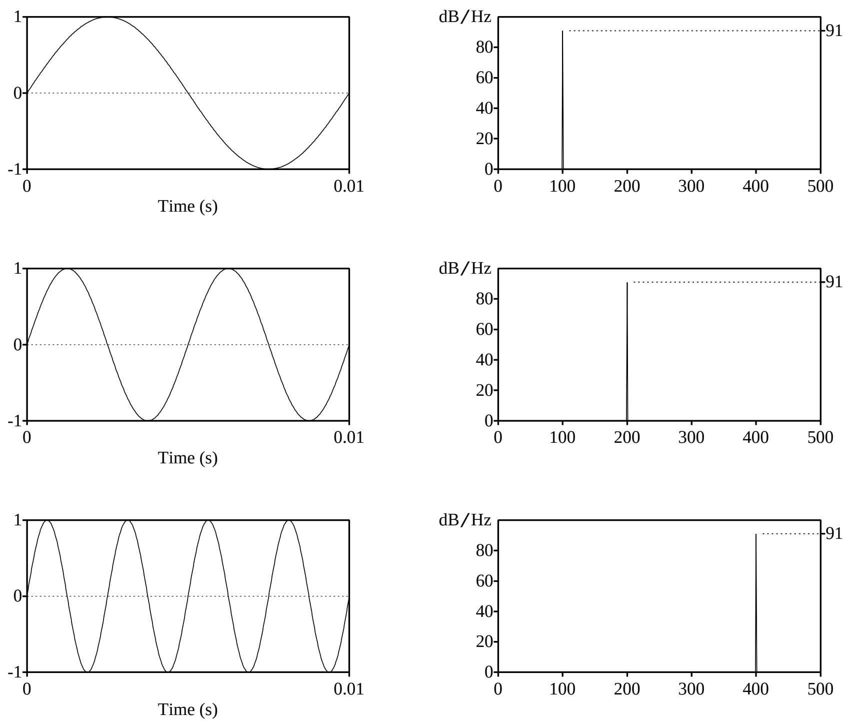


Figure 7.2.: In the left column from top to bottom the first 10 ms of 1 s duration pure tones with frequencies 100, 200 and 400 Hz. The right column shows the amplitude spectrum of each tone.

Because the frequency scale of the amplitude spectrum is a linear scale, a frequency of 200 Hz is twice as far from the origin at 0 Hz as a frequency of 100 Hz.

The last row shows, on the left, the first 0.01 s interval of the pure tone with frequency 400 Hz, and like the previous tones, with an amplitude of one. The period now equals $T = 1/400 = 0.0025$ s, hence four period fit into the plot interval of 0.01 seconds. The line in the amplitude spectrum on the right shows there is only one frequency component at a frequency of 400 Hz and again with an amplitude of 91 dB/Hz.

We could have continued figure 7.2 with more rows, showing periods of other pure tones of amplitude one and their spectra. This would always have resulted, for a tone with frequency f, in a left plot that shows $0.01 \times f$ periods and in a right amplitude spectrum with a line at frequency f with the same amplitude as before. Our conclusion is that *the amplitude spectra of pure tones with equal amplitudes show peaks of equal heights*.

7.1.2. The spectrum of pure tones of varying amplitude and decibels

Now that we know that different tones have different positions in the amplitude spectrum, we want to investigate how differences in the tone's amplitude translate to the amplitude spectrum. In the left column of figure 7.3 the first 10 milliseconds of tones with the same 200 Hz frequency but different amplitudes are plotted. The amplitude varies in steps of 10. The top figure has amplitude $a = 1$, the middle one has $a = 0.1$ and the bottom one is barely noticeable because of its small amplitude of $a = 0.01$. The figures in the first row are equal to the

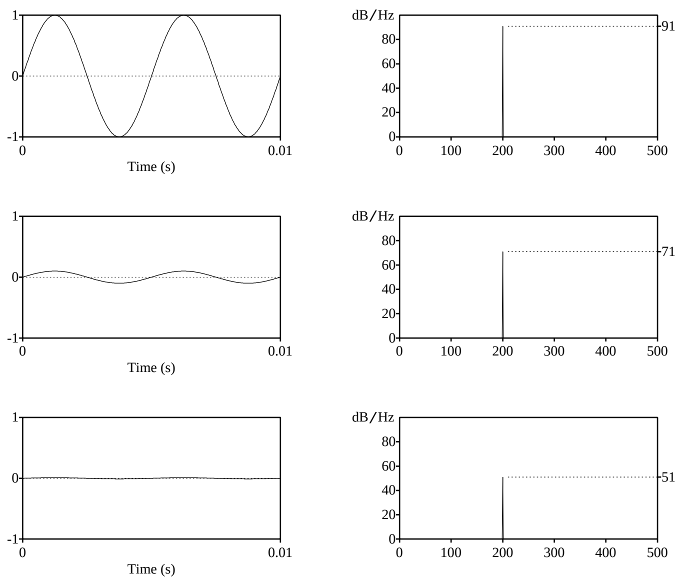


Figure 7.3.: In the left column from top to bottom the first 10 ms of 1 s duration pure tones with frequency 200 Hz and amplitudes of 1, 0.1 and 0.01. The right column shows the amplitude spectrum of each tone.

figures in the second row of the previous figure since these tones are equal. As the figures in the right column make clear, going from the first row to the second, an amplitude reduction by a factor of 10, results in a spectral amplitude reduction of 20 dB. We further note that as we reduce the amplitude this has no effect on the position of the peak in the amplitude spectrum, only on its heigth. As was shown in section A.4.2, the difference between amplitudes a_1 and a_2 in dB's can be calculated as $20\log(a_1/a_2)$. If we want to compare the tone from the top row with the one in the middle row, then with the values $a_1 = 1.0$ and $a_2 = 0.1$ we obtain $20\log(1.0/0.1) = 20\log(10) = 20\,\text{dB}$, i.e. the first tone is 20 dB louder than the second. Had we performed the calculation the other way around and taken $20\log(a_2/a_1)$, the result would have been $20\log(0.1/1.0) = -20\,\text{dB}$, i.e. the second tone is 20 dB weaker than the first tone. Both calculations result in the same 20 dB difference in spectral amplitude between the two tones. Only the signs of the numbers differ: a negative sign indicates that the denomi-

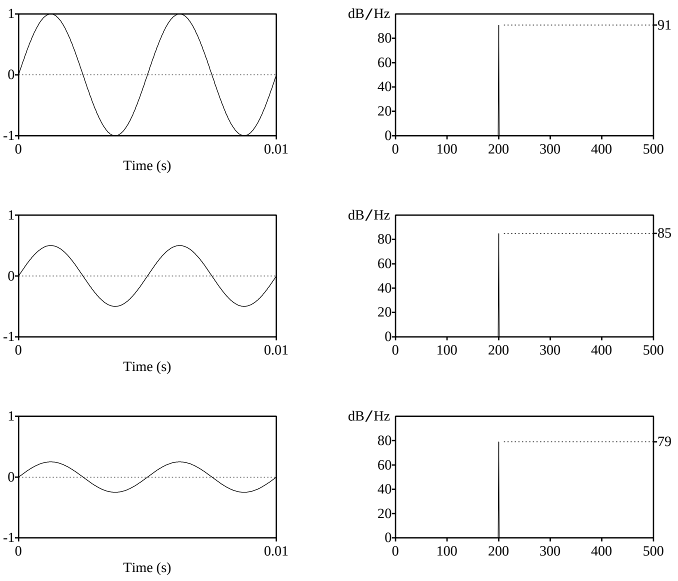


Figure 7.4.: In the left column from top to bottom the first 10 ms of 1 s duration pure tones with frequency 200 Hz and amplitudes of 1, 0.5 and 0.25. The right column shows the amplitude spectrum of each tone.

nator value is larger than the numerator. The value at the horizontal line in the middle row's spectrum confirms our calculation because it reads 71 dB/Hz which is 20 dB/Hz less than the 91.

When we compare the second row with the third, then again we have an amplitude reduction by the same factor 10. This results, again, in a 20 dB spectral amplitude reduction. The amplitude of the tone in the third row and the first row differ by a factor of 100. The calculation gives a difference of $20\log(100/1) = 20\times 2 = 40$ dB. This is confirmed by the 51 dB/Hz value in the amplitude spectrum of the third row.

As a confirmation we show in figure 7.4 the effect of reducing the amplitude of pure tones of 200 Hz by factors of 2. The first row is identical to the first row of the previous figure. The amplitudes of the tones in the left column, from top to bottom, are 1.0, 0.5 and 0.25. For the expected differences of the spectral amplitudes in decibel, we expect $20\log(1.0/0.5) = 20\log 2 \approx 20 \times 0.3 = 6$ dB. Because $a_1/a_2 = a_2/a_3$, the difference between the first and the second and the difference between the second and the third spectral amplitudes should be equal to 6 dB. The numbers at the horizontal lines in the amplitude spectrum confirm our calculations.

This shows that the amplitude and frequency of pure tones are displayed independently of each other in the amplitude spectrum. The frequency determines the position of the line on the frequency axis and the amplitude the height of the line, i.e. its spectral amplitude. *Amplitude and frequency are two independent aspects of a pure tone.*

7.1.3. The spectrum of pure tones of varying phase

In figures 7.2, 7.3 and 7.4 the sounds all start with a zero amplitude value. What would happen to the amplitude spectrum if the pure tones didn't start at a time where the amplitude is zero? To model this, the sine function of equation (7.1) is not sufficient because this one always starts with an amplitude of zero at time $t = 0$. However, an extra parameter in the argument of the sine can change this behaviour. This parameter is called the *phase*. Section A.1.3 contains more information on phase. We write the new pure tone function as

$$s(t) = a\sin(2\pi f t + \phi), \tag{7.2}$$

where ϕ denotes the phase. At time $t = 0$ the amplitude of the tone will be $s(0) = a\sin(\phi)$. By choosing the right value for ϕ, we can make $s(0)$ equal to any value in the interval from $-a$ to $+a$. In the left column of figure 7.5 we show, from top to bottom, the pure tones with constant

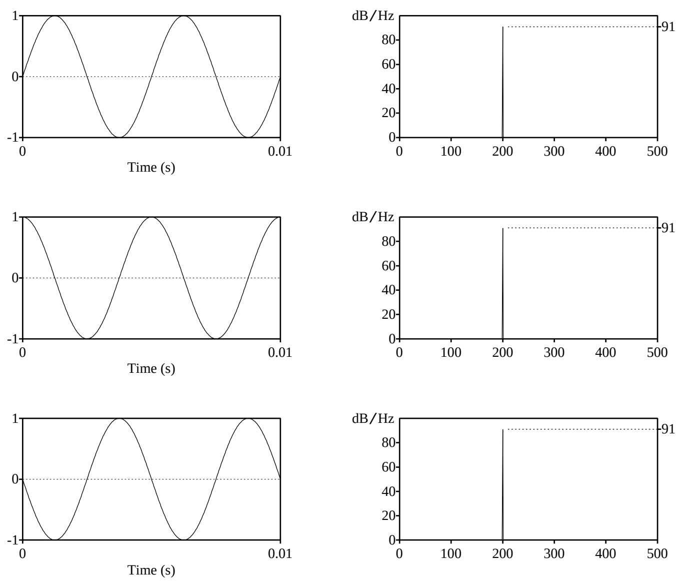


Figure 7.5.: In the left column from top to bottom the first 10 ms of 1 s duration pure tones with frequency 200 Hz and phases of 0, $\pi/2$, and π. The right column shows the amplitude spectrum of each tone.

frequency 200 Hz and amplitude 1, for three different phases ϕ of 0, $\pi/2$ and π. In the right column, the corresponding amplitude spectra are plotted. The amplitude spectra all show only one line with the same frequency and spectral amplitude in the three spectra. We conclude from this figure that *the phase of the tone has no influence on the amplitude spectrum*, only sound amplitude and sound frequency matter.

Warning: although the frequencies and amplitudes of the three sounds in the figure are the same, only the ones with phase 0 and phase π will *sound the same*. In the sound with phase

$\pi/2$ you will hear two clicks, one near the start and the other near the end of this sound. These clicks are caused by the abrupt change in amplitude at the start and end of this sound. Imagine the loudspeaker cone which is at rest, it has to move to full amplitude immediately at the start of the sound. It has to move in no time to reach its maximum, this fast movement causes the click because the loudspeaker will overshoot. The opposite happens at the end, the cone is at its maximum and immediately has to return to its rest position. The clicks at start and end are caused by the *discontinuities* in the signal. Abrupt changes in the amplitude of the sound are called discontinuities and cause click effects when you listen to them.

7.1.4. The spectrum of a simple mixture of tones

The pure tones we have worked with in the previous sections are elementary sounds but in no way the sounds of daily life. We now investigate complexer sounds and show what happens in the amplitude spectrum when we combine a number of these elementary sounds.

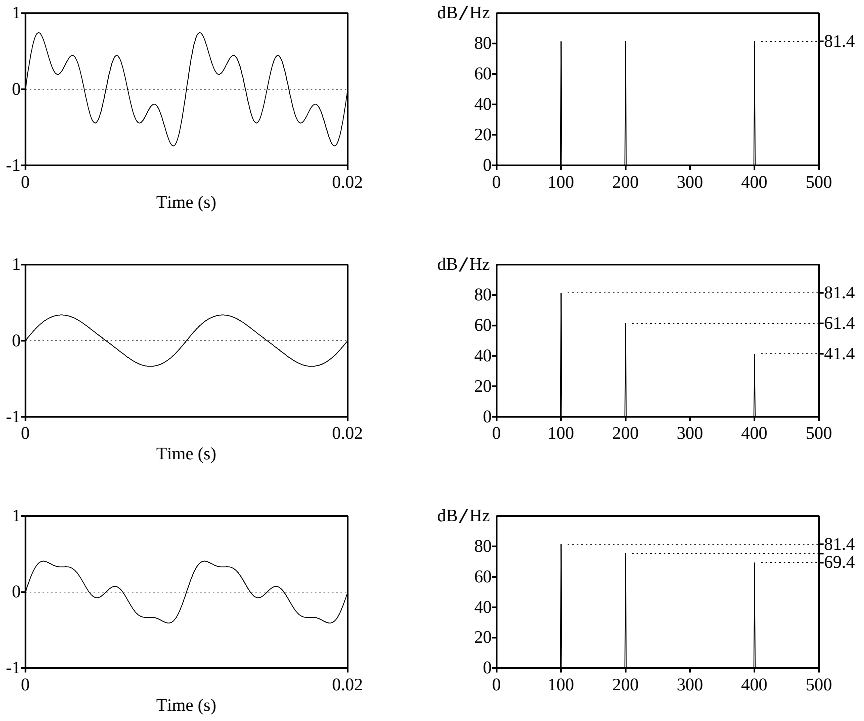


Figure 7.6.: In the left column from top to bottom the first 20 ms of 1 s duration mixtures of three pure tones of frequencies 100, 200 and 400 Hz. The right column shows the amplitude spectrum of each mixture.

In figure 7.6 we show on the left, from top to bottom, three different mixtures of three sines. The mixtures were created with:

```
Create Sound from formula: "mixture", "Mono", 0, 1, 44100,
    ... "1/3*a1*sin(2*pi*100*x)+1/3*a2*sin(2*pi*200*x)+1/3*a3*sin(2*pi*400*x)"
```

The formula shows that we add three tones with frequencies 100, 200 and 400 Hz. By choosing values for the coefficients a_1, a_2 and a_3, we can mix these tones in any way we like. The factor 1/3 guarantees that the sum of these three sines never exceeds one.[5] The mixture in the top row has all three coefficients equal to one, i.e. $a_1 = a_2 = a_3 = 1$. The amplitude spectrum on the right shows that if we add tones of equal sound amplitudes, they show an equal spectral amplitude. This spectral amplitude is 81.4 dB. If we had left out the scale factor of 1/3, the spectral amplitudes would all have been equal to 91 dB, just like they were in figure 7.2. We now know how to do the math to account for the 1/3: $20\log(1/3) = -20\log 3 \approx -20 \times 0.477 = -9.54$ dB. The scale factor of 1/3 lowers the spectral amplitude with 9.54 dB, from 91.0 to 81.46 which was rounded down to 81.4 dB.

The mixture in the middle row has $a_1 = 1$, $a_2 = 0.1$ and $a_3 = 0.01$, just like the single tones in figure 7.3 had. Here they show the same 20 dB spectral amplitude difference between successive values but now in the same plot.

The last row shows the mixture with $a_1 = 1$, $a_2 = 0.5$ and $a_3 = 0.25$. In the amplitude spectrum the peaks are 6 dB apart.

All the relations between sound amplitudes and spectral amplitudes that were established for pure tones also seem to work for mixtures or combinations of pure tones.

7.1.5. The spectrum of a tone complex

From the previous figure 7.6 it seems that by only varying the values of the three amplitude coefficients, a great variety of sounds can be made. What do the sounds look like when we would allow for more frequencies? Because we have many frequencies and many ways to mix them, we start with some popular examples with prescribed amplitudes and frequencies. The following script generates the sounds in the left column of figure 7.7.

Script 7.2 The synthesis of a sawtooth function

```
f0 = 200
n = 20
Create Sound from formula: "s", "Mono", 0, 1, 44100, "0"
for k from 1 to n
    Formula: "self + (-1)^(k-1) / k * sin(2*pi*(k*f0)*x)"
endfor
```

The script implements the formula $s(t) = \sum_{k=1}^{n} (-1)^{k-1}/k \sin(2\pi k f_0 t)$, the approximation of a so-called sawtooth function for $n = 20$ and $f_0 = 200$. The formula sums tones whose frequencies are multiples of a fundamental frequency f_0. We have a special name for frequencies that are integer multiples of a fundamental frequency: they are called *harmonic* frequencies. The fundamental frequency itself is called the *first* harmonic, the frequency $f = 2f_0$ is called the *second* harmonic, etc.[6]

[5]In the multiplication by 1/3 we implicitly assume that the three coefficients are not larger than one.

[6]In music terminology one refers to the term *overtone*. An overtone is harmonically related to the fundamental

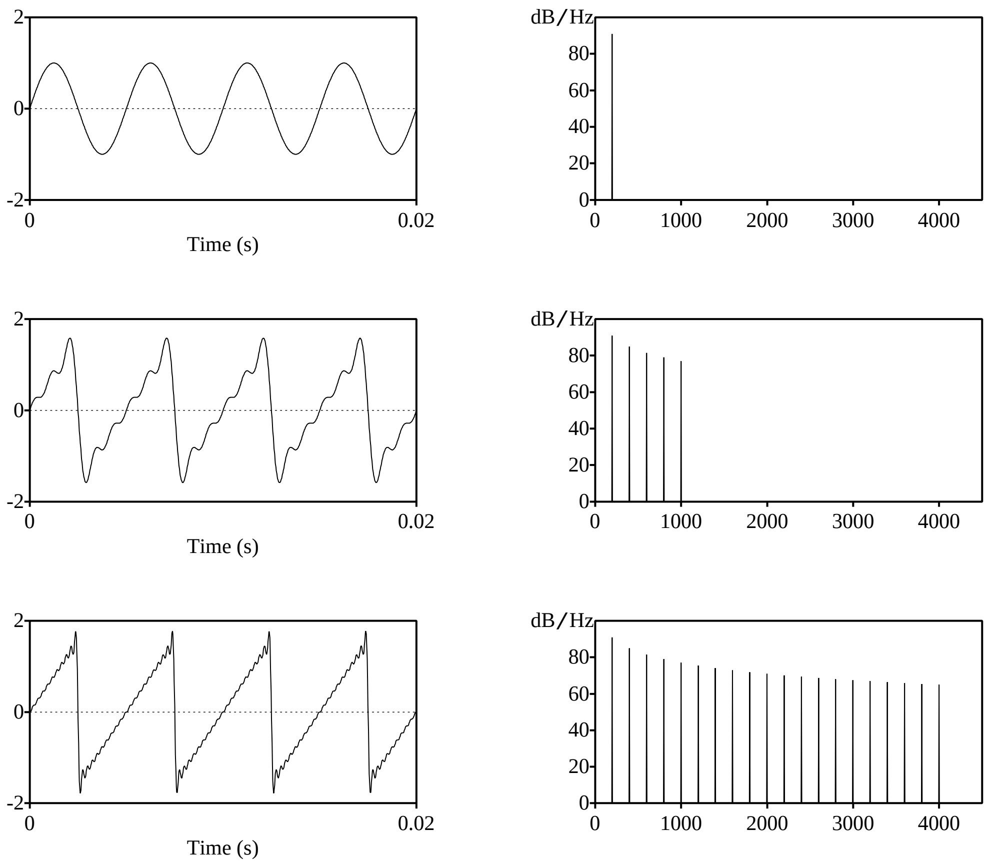


Figure 7.7.: The left column shows the sawtooth function synthesis with 1, 5 and 20 terms. The right column shows the spectrum of each synthesis.

The amplitude of each sine is $(-1)^{k-1}/k$. There are two parts in this term: because of the $1/k$, the amplitude of each sine is inversely proportional to its index number. The $(-1)^{k-1}$ part equals -1 for all even k, and equals $+1$ for all odd k. This results in a sum with alternating positive and negative terms: sines with odd k are added, while sines with even k are subtracted. The first three terms this sum are $s(t) = \sin(2\pi f_0 t) - 1/2\sin(2\pi 2 f_0 t) + 1/3\sin(2\pi 3 f_0 t) + \cdots$.

The script goes as follows: in script line 3 a silent sound object is created. For each new value of the loop variable k, a new tone is added to the existing sound. After 20 additions the script terminates. In the left column of figure 7.7, we show from top to bottom, the first

frequency but the numbering is shifted by one, i.e. the *first overtone* equals the *second harmonic*, the second overtone equals the third harmonic, etc. A special technique called overtone singing is used by many peoples around the globe. By careful articulation they amplify a specific overtone and suppress others. If you want to learn these techniques, see the book by Rachelle [1995].

0.02 seconds of the sawtooth approximation with 1, 5 and 20 terms, respectively. This can be accomplished by the substitution n=1, n=5 and n=20 in script line 2, respectively. The right column in the figure shows the corresponding spectra with 1, 5 and 20 components. The sound at the top shows the approximation for n=1, only one term. This term is, as you can see from the formula, just a $sin(2\pi f_0 t)$. Its spectrum contains consequently only one value at frequency f_0. In the middle row four extra terms show up and the sawtooth pattern is already recognizable. Note that the minus signs from the second and the fourth terms are not visible in the amplitude spectrum because this information is in the phase part of the spectrum only. In the bottom row the 20-term approximation clearly shows the sawtooth function shape.

When the number of terms in the synthesis increases, the sound $s(t)$ will synthesize the sawtooth increasingly better. How many terms do we need? In theory the number n in the formula has no upper bound. In practice we have a sampled sound and the sampling frequency limits the highest frequency that we can represent. In the sawtooth synthesis that we used, the sound had a sampling frequency of 44100 Hz which gives a highest possible frequency of 22050 Hz, the Nyquist frequency. Therefore the frequency of the components should not exceed the Nyquist value. The component with index $k = 110$ has frequency $f_{110} = 110 \times 200 = 22000$ Hz, the next one with $k = 111$ has frequency $f_{111} = 111 \times 200 = 22200$ Hz and it is already too high. This term would alias to a lower frequency. Therefore, for a sound with sampling frequency 44100 Hz we can use maximally 110 terms in the synthesis of a sawtooth if the frequencies of the synthesis components are multiples of 200 Hz.

From the first 0.02 seconds of the sounds on the left in the figure, we note that they are all periodic. There seem to be four periods of the sawtooth function in this time interval. Although, what is exactly one period, i.e. the repeating part, of the sawtooth function? Is it the part that rises from zero to its maximum, falls to the minimum and rises to zero again, or, is it the part that rises from its minimum value to its maximum? Perhaps the most natural choice here is dictated by how the sawtooth starts and we could opt for the first possibility: four successive periods are present in the time interval of 0.02 second duration.

In a second synthesis example in this section, we present the following script to synthesize another elementary signal: the block function.

Script 7.3 The synthesis of a block function.

```
f0 = 200
n = 20
Create Sound from formula: "b", "Mono", 0, 1, 44100, "0"
for k from 1 to n
    Formula: "self + sin(2*pi*((2*k-1)*f0)*x) / (2*k-1)"
endfor
```

The mathematical formula for a block is $b(t) = \sum_{k=1}^{n} \sin(2\pi(2k-1)f_0 t)/(2k-1)$. In the synthesis, because of the $2k - 1$ term, only odd multiples of f_0 are used. The first three terms are $b(t) = sin(2\pi f_0 t) + \sin(2\pi 3 f_0 t)/3 + \sin(2\pi 5 f_0 t)/5 + \cdots$. In figure 7.8, we show the results for three different values of the parameter n.

Isn't it amazing that if we combine sines, whose frequencies are harmonics of some frequency f_0, these block and sawtooth functions appear, with a period T that equals $1/f_0$?

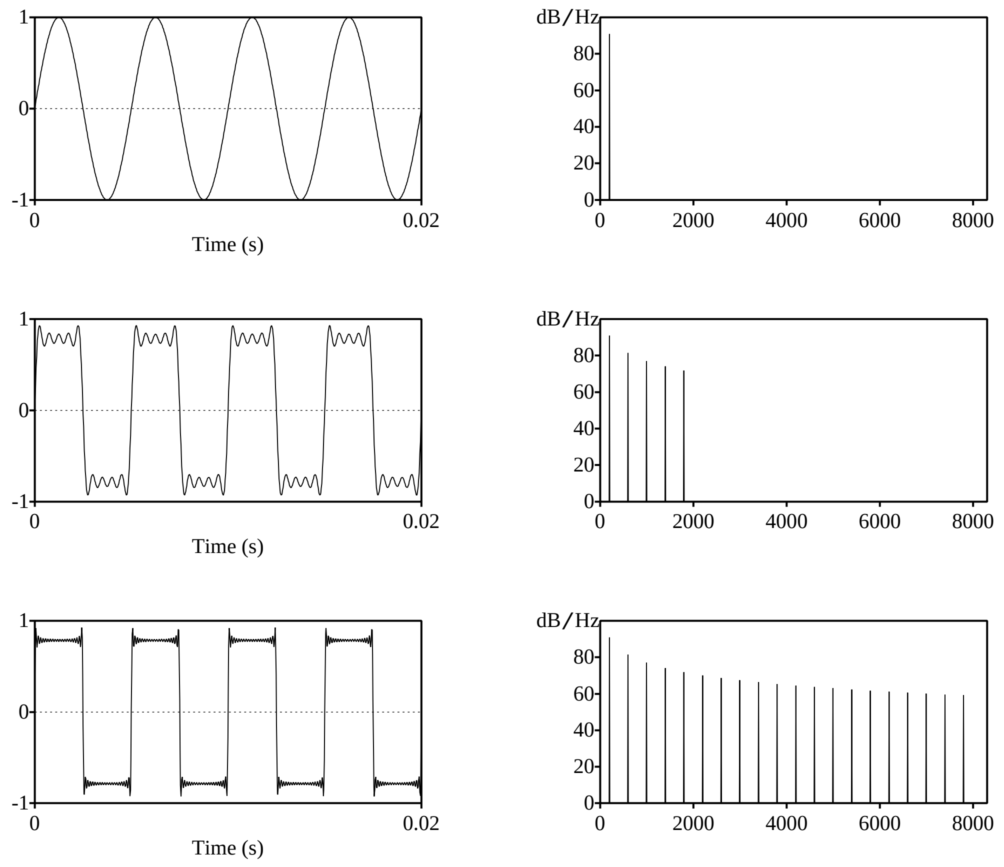


Figure 7.8.: The left column shows the block function synthesis with 1, 5 and 20 terms. The right column shows the corresponding amplitude spectrum.

Does this mean that any combination of harmonics leads to a periodic function? With the following script you can try it.

```
f0 = 200
n = 20
Create Sound from formula: "random", "Mono", 0, 1, 44100, "0"
for k from 1 to n
    ak = randomUniform (-0.9, 0.9) / k
    Formula: "self + ak * sin(2*pi*k*f0*x)"
endfor
```

The script synthesizes a sound object with 20 harmonics. The function `randomUniform(-0.9, 0.9)` will generate a new random uniform number with a value between −0.9 and 0.9 every time this function will be used. Each number will be completely independent from the previous one(s). if we run the script a number of times in succession, then each time a different series

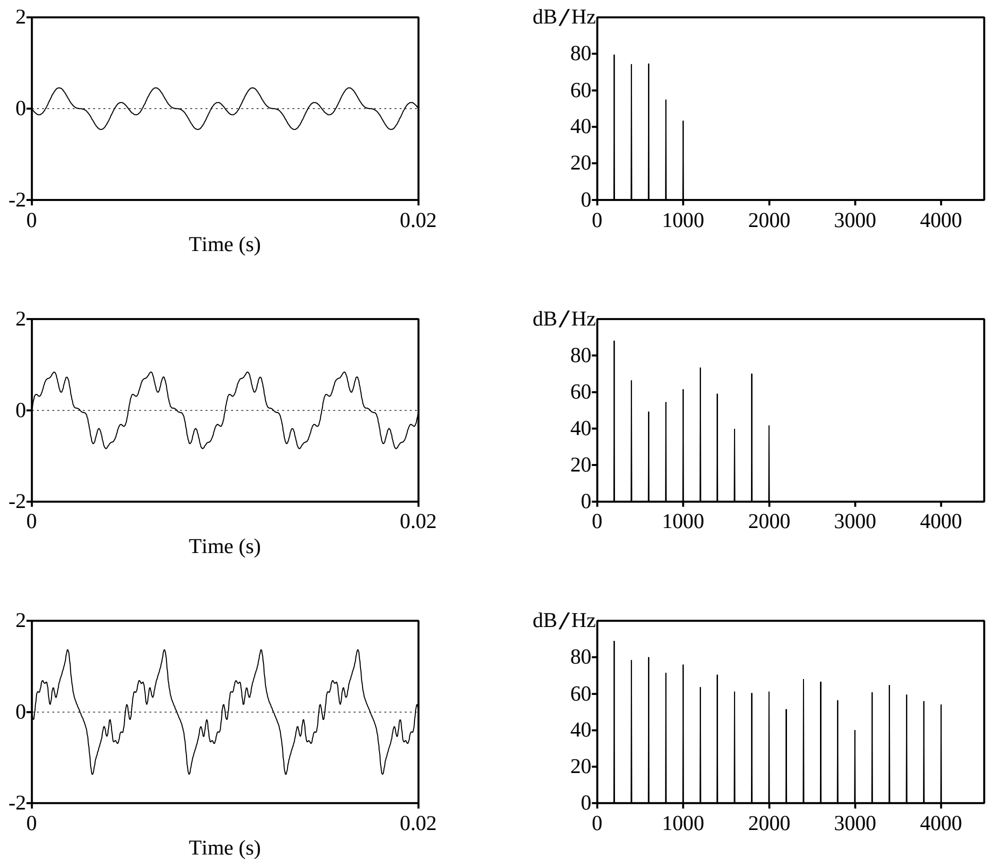


Figure 7.9.: The left column shows the synthesis with 5, 10 and 20 sine terms with random uniform amplitudes. The right column shows the amplitude spectrum of each synthesis.

of random uniform numbers will result.[7] The amplitude of each harmonic will be the product of this uniform random number and the scale factor $1/k$. In figure 7.9 the left column shows the synthesis of three sounds from the script, where the number of components n from top to bottom was chosen as 5, 10 and 20. The scale factor $1/k$ for each amplitude is for displaying reasons only: it makes the periodicity in the synthesized signal easier to see. All three signals above are *periodic* with period $1/f_0$.

If we would run the script over and over again, each time with randomly assigned frequency amplitudes, the resulting signals would all share the same period $1/f_0$ but the sound ampli-

[7]In a uniform distribution of numbers, all numbers in the distribution have equal probability of being chosen. When we write about a *random uniform number*, we use it as a shorthand for *a number randomly drawn from a uniform distribution of numbers*.

tudes as a function of time vary.[8] It is however very unlikely that we would ever reproduce exactly the sound as shown in figure 7.9.

If we would change the scale factor in the script to any function of the index we liked, the resulting signals still share the same period $1/f_0$.

We conclude: The sum of harmonically related sines synthesizes a periodic sound.

7.1.6. The spectrum of pure tones that don't "fit"

Now we are going to complicate things a little bit. In the examples we showed before, the durations of the sound and the frequencies were not picked at random. For all the frequencies we have used, an integral number of periods fitted in the sound's duration. We did this on purpose to show you that a sine with frequency f corresponds to one line in the amplitude spectrum. For example, for a sine with a frequency of 200 Hz, exactly 200 periods fit in a sound of 1 s duration. The corresponding amplitude spectrum shows a line at 200 Hz as is

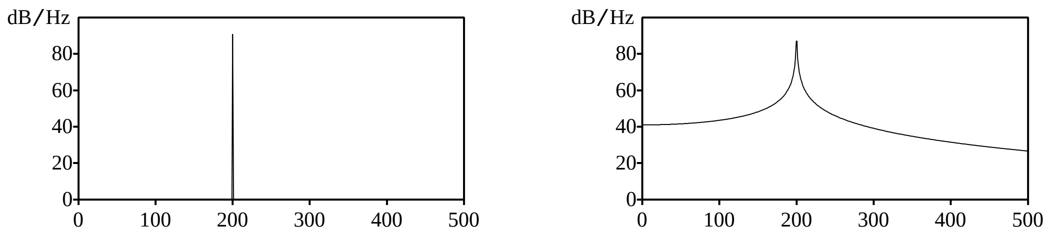


Figure 7.10.: The amplitude spectrum of a 200 Hz tone. On the left the tone was of 1 s duration and on the right of 0.9975 s duration.

shown in the left part of figure 7.10. If instead we create the tone with a duration of 0.9975 s, the amplitude spectrum looks like the right part of the figure. This does not look like a line at all, more like a very peaky mountain. To explain the difference we have to delve into the way the spectrum is calculated from the sound and how the duration of the sound comes into play. Lets us call the duration of the sound that is being analyzed T. The spectrum is calculated from the sound by a technique called Fourier analysis. A Fourier analysis tries to find amplitudes (and phases) of *all harmonics of the frequency* $1/T$ that do not exceed the Nyquist frequency. If you sum these harmonics again with the calculated amplitudes and phases you will get the signal back. Besides these harmonics the strength at a frequency of zero hertz is also calculated, this value equals the average value of the sound.[9] Therefore, a Fourier analysis *decomposes* a sound into separate frequency components.

These component frequencies are all multiples of a fundamental *analysis* frequency $1/T$. The component frequencies are then given by k/T, where $k = 0, 1, \ldots$ are on the horizontal

[8]Of course within certain limits: they are all synthesized with sine functions only, which dictates that they all start with amplitude zero.

[9]This is often called the DC component. The analogy is from the electronics domain where alternating current (AC) has the static pendant direct current (DC).

axis of the amplitude spectrum. The fundamental analysis frequency $1/T$ is not related to any inherent periodicity in the sound signal itself. Lets do some numbers to clarify these things. For a sound of 1 s duration the fundamental analysis frequency $1/T$ is $1/1 = 1$ Hz. If the sound has a sampling frequency of 44100 Hz there will be 22050 harmonics. For the 1 s duration tone of 200 Hz in figure 7.10, the Fourier analysis calculates the 22051 amplitudes and phases that describe the tone optimally.[10] Because harmonic number 200 has a frequency that exactly matches the frequency of the tone, the spectrum has only one component.

For the 0.9975 s duration tone of 200 Hz, the fundamental analysis frequency is $1/0.9975 \approx 1.0025$ Hz. For a 200 Hz tone there is no single component that matches exactly: component number 199 has a frequency of 199.5 Hz while component number 200 has a frequency of 200.5 Hz. The figure therefore shows that to represent the 200 Hz frequency all the harmonics of the 1.0025 Hz frequency are needed.

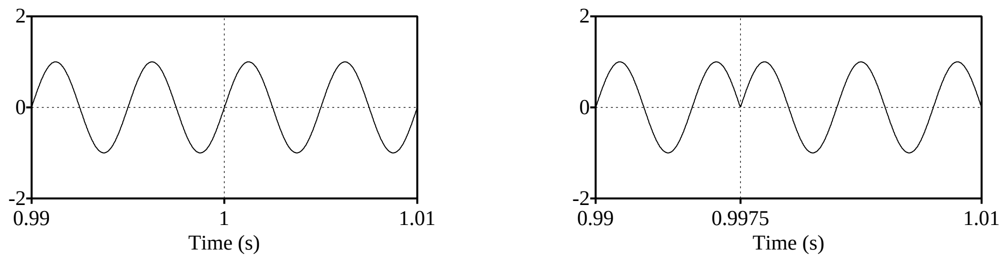


Figure 7.11.: Periodic extensions of a 200 Hz tone in a sound of duration 1 s (left) and 0.9975 s (right).

Another way to describe the difference between these two spectra is shown in figure 7.11. The Fourier analysis works with the underlying assumption that a sound that is analysed is periodic with period T equal to its duration, i.e. as if its duration were infinite. The analysis then occurs on this infinitely long signal that can be constructed from the original one by concatenating copies of itself. It is therefore important what happens at the borders where two copies meet. For the 1 s duration tone, the 200 Hz sine has an integral number of periods within its 1 s duration and therefore joins smoothly at the end of each interval with the sine at the start of the next interval. The left plot in the figure shows this. The infinitely extended signal just looks like a sine of 200 Hz of infinite duration. For the 0.9975 s duration sound, the 200 Hz sine does not join smoothly. It ends halfway one period as the right plot in the figure shows. This does not look like a sine anymore, because there is a discontinuity in the sound. It is clear that no single analysis frequency can match this discontinuity. [11]

Conclusion: In *Fourier analysis* a sound of duration T is decomposed in frequencies that are harmonics of the fundamental analysis frequency $1/T$. This decomposition is unique: from the decomposition we can get our original sound back with a technique called *Fourier synthesis*.

[10]Frequency points at 0, 1, 2, . . ., 22050 Hz.

[11]This explains also why the analyzing frequencies have to be harmonics of the frequency $1/T$: they are the only frequencies that are "continuous" at the borders.

7.1.7. Spectral resolution

We repeat the finding of the previous section again: a Fourier analysis decomposes a sound, of duration T seconds, in harmonic frequencies of the fundamental analysis frequency $1/T$. We will now investigate the consequences of this. For a 1 s duration sound the fundamental analysis frequency is 1 Hz and this results in frequency components that are 1 Hz apart in the spectrum. This means that for a tone complex with two tones whose frequencies f_1 and f_2 differ by 1 Hz, like for example $f_1 = 500$ Hz and $f_2 = 501$ Hz, each tone can be represented in the spectrum as a separate value. If the tone frequencies differ by less, like for example the frequencies $f_1 = 501.2$ Hz and $f_2 = 501.9$ Hz then these two frequencies have to merge into one line in the spectrum and will not be separately detectable anymore. For a sound composed of two tones with duration 1 s, the frequencies of the two tones have to be at least 1 Hz apart to be separately detectable. The term associated with this is “spectral resolution”.[12] We say that the spectral resolution is equal to the frequency $1/T$. The better the spectral resolution the lower this frequency. A longer duration results in a better resolution, a shorter duration results in worse resolution.

Figure 7.12 shows the effect of signal duration on the amplitude spectrum. In the left column we show sounds with an ever increasing number of periods of a 1000 Hz tone. The right column shows the amplitude spectrum. The sound in the first row was created with a duration of 0.001 s and shows one period of a 1000 Hz tone. Because of its very short duration, in the spectrum the analysis frequency components are multiples of $1/0.001 = 1000$ Hz. In the next row the duration is doubled to 0.002 s and the spectral resolution therefore halves to 500 Hz.[13]

[12] In a case where we know that these two frequencies are in the signal, we can use advanced techniques to measure their amplitudes and phases, because we have *extra* information. In general we have only the information in the spectrum.

[13] Because the spectrum shows power spectral *density*, the amplitude spectrum for a pure tone shows an increase of 3 dB for each time doubling. In the power spectral density spectrum that Praat shows, both the amplitude of the basis frequency components and the duration of the sound are intertwined: in the amplitude spectra of figure 7.12 doubling the sound’s duration increases the value at the 1000 Hz frequency component with 3 dB while the amplitude of the sine in the sounds did not change. This intertwining makes it impossible to directly estimate the strength of each frequency component. However, nothing is lost: the effect of duration is the same for all values in the amplitude spectrum. The whole amplitude spectrum is shifted up or downwards a number of decibels, depending whether the duration increases or decreases. All relations between the amplitudes within each spectrum therefore do not depend on duration. So you can compare the relative values of components between two amplitude spectra as well.

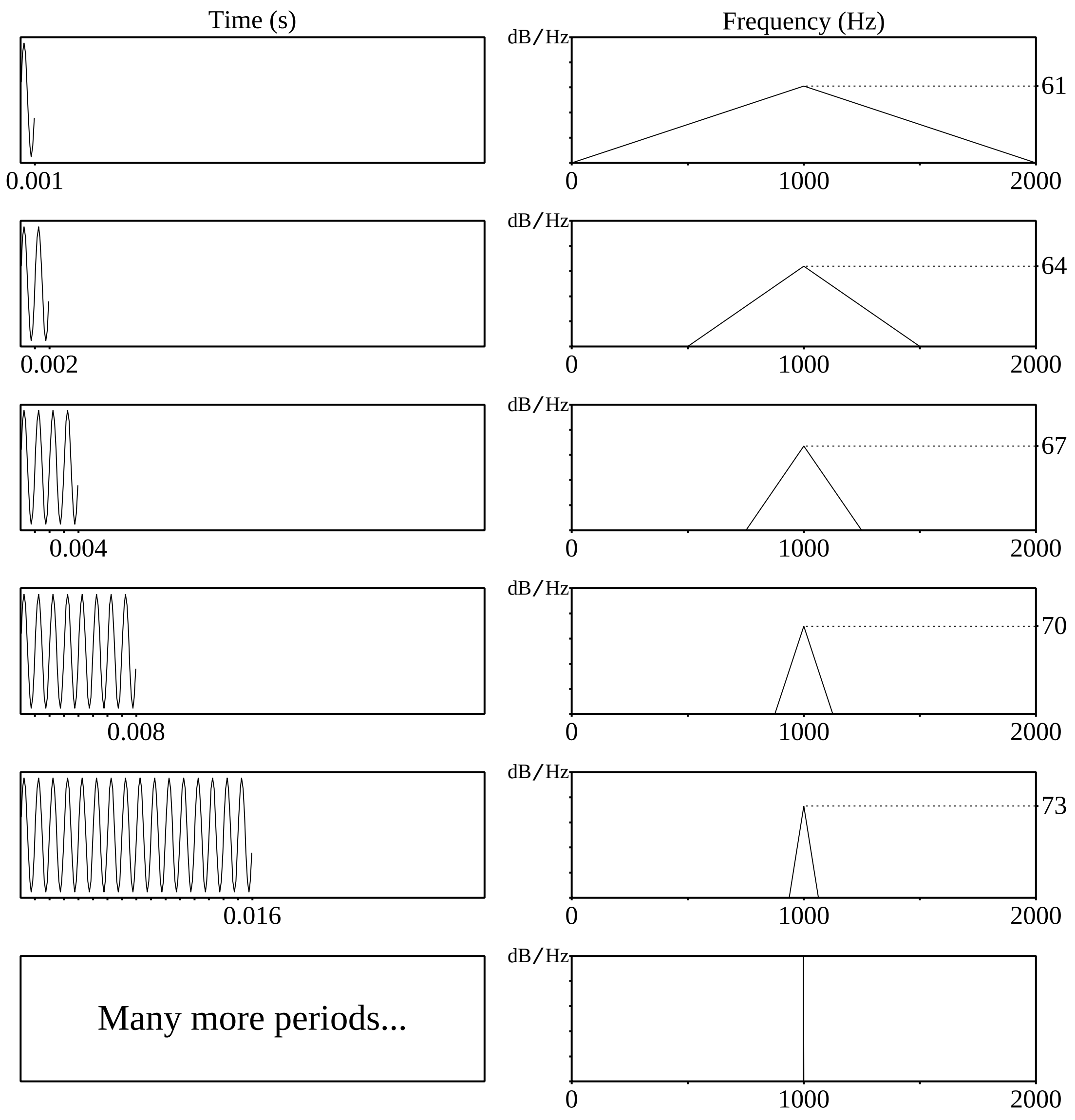


Figure 7.12.: Influence of sound's duration on spectral resolution.

We see the inverse relation between the spectral resolution and the duration of the sound. This relation is not an artifact of Fourier analysis, it is the result of the limits nature imposes upon us. In physics this is stated by the Heisenberg uncertainty principle, here it simply says that the more precise you want to determine a signal's frequency the longer you have to measure: precision takes time.

7.1.8. Why do we also need cosines?

Many examples in the previous sections used sines as the building block for Fourier analysis and synthesis. These sines were all prototypical sines that start at zero then increase to their maximum value and then decrease to their minimum, etc. We can create an infinite number of sounds with the sine as a building block. However, there is also an infinite number of sounds we *cannot* create with sines. For example all the sounds that start with an amplitude different from zero cannot be modeled by a combination of sines only. The cosine function is a natural candidate to model such functions. As the sine starts with zero amplitude the cosine starts with amplitude one. By mixing a sine function and a cosine function we can have any start value we want. In section A.1.3 we show that a mixture of a sine and a cosine function with the same argument is equivalent to a sine with a phase. We translate equation (A.2) to frequencies and write

$$a\cos(2\pi ft) + b\sin(2\pi ft) = c\sin(2\pi ft + \theta), \tag{7.3}$$

where the new amplitude is $c = \sqrt{a^2 + b^2}$ and the phase $\theta = \arctan(b/a)$.

7.1.9. Is the phase of a sound important?

We know from experiment that the ear is more sensitive to the amplitude of a frequency component than it is to the phase of these components. With the following simple script you can check this for yourself. Every time you run this script, a sound is played that has equal frequency content but the phases of the individual components differ. The only thing these sounds have in common as you can see is the periodicity. Despite the fact that they all look different, they all sound the same. This shows that the amplitude spectrum captures essential perceptual elements of a sound and is a therefore a more “stable” representation than the time signal.

```
f0 = 150
Create Sound from formula: "sines", "Mono", 0, 1, 44100, "0"
for k to 5
    phase = randomUniform (-pi/2, pi/2)
    Formula: "self + sin(2*pi*k*f0*x + phase)"
endfor
Scale peak: 0.99
# avoid clicks at start and end
Fade in: 1, 0, 0.005, "no"
Fade out: 1, 1, -0.005, "no"
Play
```

7.2. Fourier analysis

In *Fourier analysis* a sound of duration T is decomposed in components that are harmonics of the fundamental analysis frequency $1/T$. Each component is mixture of a cosine and a sine. In shorthand mathematical notation we write that for a sampled sound $s(t)$ its Fourier decomposition is

$$s(t) = \sum_{k=0}^{N/2-1} (a_k \cos(2\pi k f_0 t) + b_k \sin(2\pi k f_0 t)), \tag{7.4}$$

where $f_0 = 1/T$ is the fundamental analysis frequency and N is the number of samples in the sound. The Fourier analysis determines for each harmonic of the analyzing frequency f_0 the coefficients a_k and b_k. We can rewrite the above equation in terms of phases as

$$s(t) = \sum_{k=0}^{N/2-1} c_k \sin(2\pi k f_0 t + \theta_k) \tag{7.5}$$

In figure 7.13 we present a detailed example of a Fourier analysis of a short sound. In the top row on the left is the sound of duration T that will be analyzed. The actual value of T is not important now. The duration T specifies that the analyzing cosine and sine frequencies have to be harmonics of $f_0 = 1/T$. The following pseudo script shows the analysis structure. For

Script 7.4 The Fourier analysis pseudo script

```
1  s = selected ("Sound")
2  duration = Get total duration
3  f0 = 1 / duration
4  for k from 0 to ncomponents-1
5      <Calculate a[k] and b[k] from sound s>
6      select s
7      Formula: "self - a[k]*cos(2*pi*k*f0*x)-b[k]*sin(2*pi*k*f0*x)"
8  endfor
```

the first frequency, when $k = 0$, the Fourier analysis only determines the coefficient a_0 which equals the average value of the sound. In the script this corresponds to line 5, the first line in the for loop. For each frequency component the coefficients a_k and b_k can be determined by the technique of section A.1.4: multiply the sound “s” with $\cos(2\pi k f_0 t)$ to calculate a_k and multiply “s” with $\sin(2\pi k f_0 t)$ to calculate b_k. The coefficient b_0 is zero, because a sine of frequency zero is always zero. The analysis cosine of frequency zero with amplitude a_0 is shown in the second column of the figure, it happens to be a straight line because a cosine of frequency zero equals one. The dotted line shows the zero level. This function has exactly the same duration as the sound and is next subtracted from the sound. In the script this corresponds to the last line in the for loop, line 7. This results in the sound displayed in the third column. In the amplitude spectrum in the fourth column the value a_0 is shown with a vertical line at a frequency of 0 Hz. For displaying purposes only, the vertical scale of the amplitude spectrum is a linear one instead of the usual logarithmic one.

In the next step for $k = 1$, this new sound is now analyzed with a cosine of frequency $1/T$ and the number a_1 determined and then analyzed with a sine of the same frequency and the number b_1 is determined. In the figure this is shown in the second row: the left most figure is the sound corrected for the 0 Hz frequency. The second column shows the cosine and the sine components with amplitudes a_1 and b_1, respectively. In the third column the cosine and the sine component are subtracted from the sound. In the script we are after line 7 now. The sound does not contain any component of frequency $1/T$ any more. In the amplitude spectrum the value $\sqrt{a_1^2 + b_1^2}$ is shown at distance 1 from the previous value because the size of one unit on this axis was chosen to be $1/T$ Hz.

For $k = 2$ the sound where the lower frequency components have been removed is analyzed with a cosine and a sine of frequency $2/T$ and the coefficients a_2 and b_2 are determined. These frequency components are removed from the sound and in the amplitude spectrum the value $\sqrt{a_2^2 + b_2^2}$ is drawn at unit 2. The third row in the figure shows these steps.

The same process continues for $k = 3$ as the fourth row shows. At the end of the next step, $k = 4$, we are done because after subtraction of the components of frequency $4/T$ there is nothing left in the sound: all sample values are zero. All amplitude values a_k and b_k for $k > 4$ will be zero. The amplitude spectrum will look like the one in the fifth row. This completes the Fourier analysis part.

In the last row in the figure we show the sum of all the components in the second column, i.e. all the cosines and sines from the first five rows. This is the Fourier synthesis and we get the sound where we started from back, the one at the top row at the left.

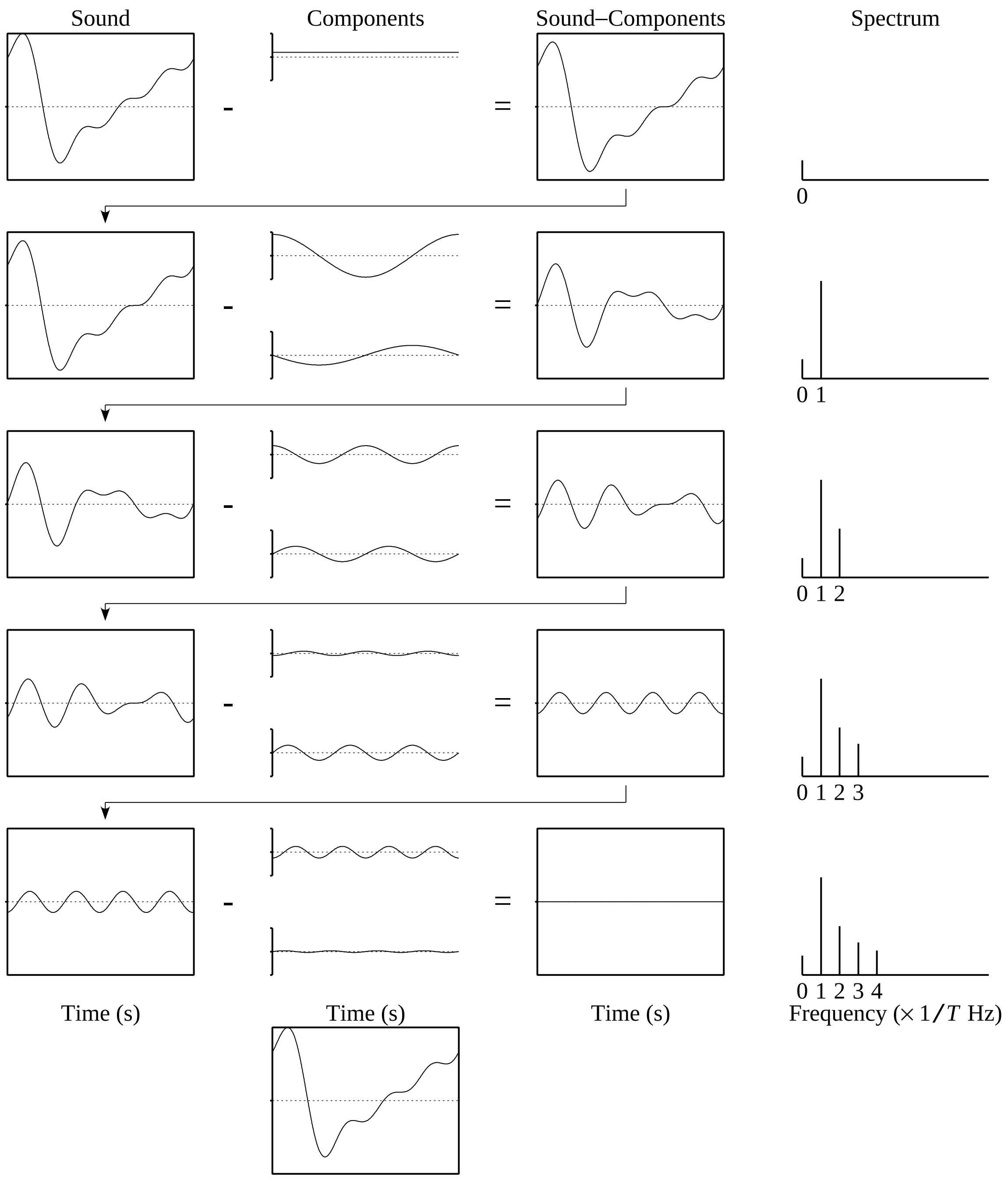


Figure 7.13.: An example of the Fourier analysis of a sound.

7.3. The spectrum of pulses

To show how the spectrum of a pulse looks like we go back to figure 7.12 again. In this figure we increased the number of periods of a tone to show that the more exact you want to determine the frequency of a tone, the more periods you need to measure: precision in frequency determination takes time. We now will go in the opposite direction: we will decrease the length of a sound until only one sample value is left and we will find the spectrum of such a signal. A sound with only one sample value different from zero is so special and important that it receives a special name: it is called a pulse function. In mathematics a pulse is infinitely thin and is called a Dirac pulse. For sampled signals a pulse has a width of one sample period.

A pulse is important because a sampled sound can be modeled as the product of an analog sound and a pulse train with pulses one sampling period apart as figure 7.14 shows. From

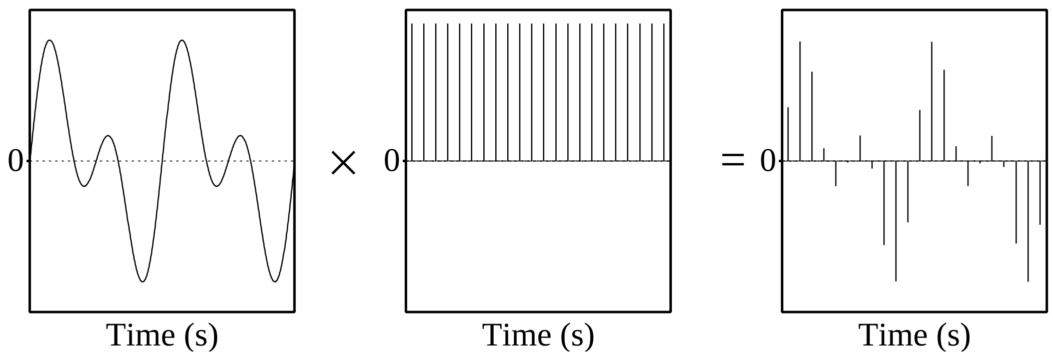


Figure 7.14.: An analog sound times a pulse train is a sampled sound.

figure 7.12 we note that the spectrum gets broader as the time of the signal decreases. In the limit when the sound is reduced to only one sample the spectrum is at its broadest. The spectrum of a pulse has all frequencies present at equal strengths, i.e. from zero hertz to the Nyquist frequency. The spectrum of a pulse is a constant amplitude “straight” spectrum.

The following script creates a pulse and draws the spectrum of figure 7.15.

```
Create Sound from formula: "pulse", "Mono", 0, 0.1, 44100, "col=100"
To Spectrum: "no"
Draw: 0, 0, 0, 40, "no"
```

7.4. Praat’s internal representation of the Spectrum object

In Praat the Spectrum object is a complex object and it is represented as two rows of numbers. The first row represents the strengths of the cosine frequency components and the second row the strengths of the sine frequency components. The amplitude and an approximation of the phase can be calculated from these sine and cosine components. As rows of numbers have

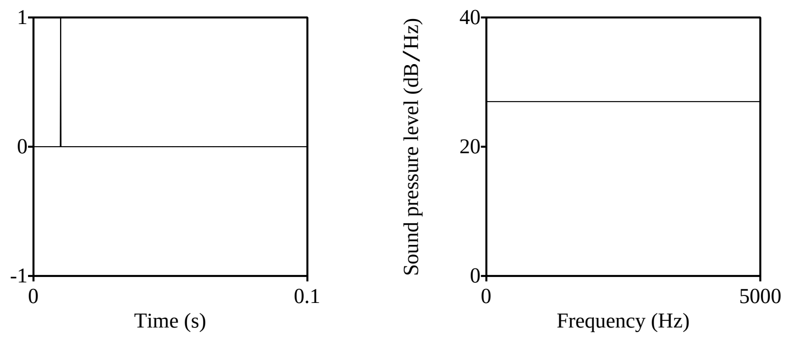


Figure 7.15.: The spectrum of a pulse.

no meaning, the information about what these numbers represent must also be stored in the Spectrum object. The extra data is

xmin the lowest frequency in the spectrum. This will equal 0 Hz for a spectrum that was calculated from a sound.

xmax the highest frequency in the spectrum. This will be equal to half the sampling frequency if the spectrum was calculated from a sound.

nx the number of frequencies in the spectrum.

dx the distance between the frequencies. This will be $1/T$ if the spectrum was calculated from a sound and T is the duration of the sound.

x1 the first frequency in the spectrum. This will equal 0 Hz for a spectrum that was calculated from a sound.

7.5. Filtering with the spectrum

Filtering is the process in which the spectral components in certain frequency regions are changed, i.e. the strengths and/or the phases of individual components can be changed in the filtering process. In the Spectrum object we have all the frequency components at our disposal (as two rows of numbers). When we change some of these numbers we are actually performing some kind of filtering operation: we are *filtering in the frequency domain*. In the sequel we implicitly assume that the spectrum is the result of applying a Fourier analysis on a sound. Thinking about the filtering process becomes easier when we think about an underlying sound. Therefore, we filter a sound by applying a frequency domain technique. This technique is very powerful because we can filter the sound as a whole thing. Remember, we can transform from the sound to the spectrum and from the spectrum back to the sound without information loss. In the real world filters have to be *causal* which means that there cannot be any output signal before an input signal is applied. Filtering in the frequency domain is an *acausal* technique because the spectrum is processed as a whole, in one batch, without regard to time ordering. There is no time ordering in the spectrum, time ordering will only reappear when the sound is created from the spectral components by Fourier synthesis. By applying this acausal frequency domain filtering we create more possibilities than any causal techniques can realise.[14] If we are not bound to any real-time application we prefer this method. We will illustrate filtering with the spectrum editor.

7.5.1. The spectrum editor

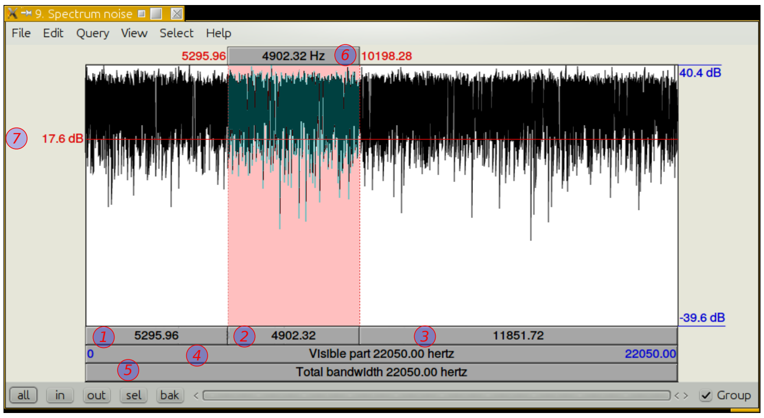


Figure 7.16.: The spectrum editor with the spectrum of a random Gaussian noise sound. The pink area shows a selected frequency interval between 5295.96 and 10198.28 Hz.

The spectrum editor appears after clicking **View & Edit** for a selected spectrum object. In figure 7.16 you see an example of the spectrum of a random Gaussian noise sound.[15] The layout of the spectrum editor is very similar to the layout of the sound editor (see figure 2.7). However, the horizontal axis here displays frequency instead of time and runs from 0 Hz to the Nyquist frequency, which, for this spectrum is 22050 Hz. The vertical scale shows only the *amplitude* of the spectrum at a logarithmic scale in dB's. In the figure the scale shows a view range of 70 dB and runs from a maximum which happen to be at 40.6 dB to a lower value at -39.4 dB. Theoretically, the spectrum of a random Gaussian noise is a flat spectrum and the figure shows that the spectrum displayed is indeed (almost) flat. The figure displayed also shows that we have made a selection in the spectrum. The pink area represents the selected part and rectangles show the widths in hertz of the corresponding intervals. The rectangle labeled with the marker 1 happens to be 5295.96 Hz wide and starts at a frequency of 0 Hz

[14]In general the frequency domain technique also results in less dispersion of phases, the filters can have sharper filter edges, and the filter responses are less asymmetric. We will try to explain these terms in the next parts of this chapter. . .

[15]A spectrum like this can be created as follows:

```
Create Sound from formula:  "noise", 1, 0, 1, 44100, "randomGauss (0,0.1)"
To Spectrum:  "no"
View & Edit
```

and therefore ends at 5295.96 Hz. The second interval, which is the selected interval, is represented by the two equally sized rectangles, the one below labeled with a 2 and the one above labeled with a 6. It starts at 5295.96 Hz and ends at 10198.28 (= 5294.96 + 4903.32) Hz. The interval labeled 3 starts at 10198.28 and ends at 22050 Hz. Intervals indicated with a 4 and a 5 here represent the complete frequency range of the sound.

The rectangles suggest that clicking them will generate sound and, indeed, this is the case. If you click in the rectangles labeled 4 and 5 you will hear the "original" sound. However, there is no copy of the original sound in the spectrum, but instead, the original sound is created by a *Fourier synthesis* of all the frequency amplitude and phase components in the spectrum, i.e. the information of the original sound is stored in all the frequency-phase pairs of the spectrum. If, however, you click successively in the rectangles labeled 1, 2 and 3, you will hear different noise sounds who seem to increase in frequency. What goes on is as follows: if you click in a rectangle all the spectral components below the start frequency and above the end frequency of the rectangle are first set to zero and the played sound is created from the remaining frequency components only. Therefore, instead of the whole frequency range of the sound being played, only a *selected frequency band* has been played. By selecting different parts of the spectrum and clicking the corresponding rectangle we can listen to different frequency bands of the original sound. Each time it appears as if the original sound has been *band-filtered* before playing. Three of the four possible different kinds of band-filtering of a sound are represented in the figure.

1. **Low-pass filtering**. This name is used for band-filtering if the lowest frequency of the band equals 0 Hz. For rectangle 1 this is the case as it lowest frequency equals 0 Hz and only frequencies between 0 and 5295.96 Hz are played. In low-pass filtering the low frequencies are allowed to pass and the high frequencies are blocked.

2. **High-pass filtering.** If the lower frequency is higher than zero and the higher frequency equals the Nyquist frequency. In playing rectangle 3 it appears as if the sound were high-pass filtered as only frequencies between 10198.28 and 22050 Hz are played. In high-pass filtering the highest frequencies in the sound are allowed to pass while the lowest frequencies are blocked.

3. **Band-pass filtering**. This name is used for filtering if the lowest frequency is greater than zero and the highest frequency is lower than then Nyquist frequency. Playing rectangle 2 implements band-pass filtering. In band-pass filtering both the lowest as well as the highest frequencies are blocked. Band-pass filtering can also be performed by successively applying high-pass filtering and low-pass filtering.

4. **Band-stop filtering**. Band-stop filtering blocks frequencies in the band and lets frequencies outside the band pass. In contrast to band-pass filtering, band-stop filtering *cannot* be performed by successively applying low- and high-pass filtering. In the figure it is as if rectangle 1 and 3 were played together.

After playing one of the rectangles the corresponding sound is destroyed. If you want to keep the corresponding sound then choose **Publish band-filtered sound** from the File menu.

7.5.2. Examples of scripts that filter

Script 7.5 Filtering in the frequency domain.

```
selectObject: "Sound s"
To Spectrum: "no"
Formula: "<some formula>"
To Sound
Play
```

Filtering in Praat is very easy to do. Script 7.5 shows the skeleton filter setup. By substitution of a formula different filters can be realized. Note that the formula is applied to the spectrum!

The filter terminology we introduced in the previous section:

Low-pass filter: Low frequencies pass the filter, high frequencies not. For example a low-pass filter that only passes frequencies lower than 3000 Hz can be defined by using the formula:

```
Formula: "if x<3000 then self else 0 fi"
```

The frequency from which the suppression starts is also called the *cut-off* frequency.

High-pass filter: high frequencies pass, low frequencies are suppressed. The following filter formula only passes frequencies above 300 Hz.

```
Formula: "if x>300 then self else 0 fi"
```

Band-pass filter: frequencies within an interval pass, frequencies outside the interval are suppressed. The following filter formula only passes frequencies between 300 and 3000 Hz, approximately the bandwidth of a plain old telephony signal.

```
Formula: "if x>300 and x<3000 then self else 0 fi"
```

Band-stop filter: frequencies outside the interval pass, frequencies inside the interval are suppressed.

```
Formula: "if x>300 and x<3000 then 0 else self fi"
```

All-pass filter: all frequencies pass but phases are modified. The following filter makes the sound completely not understandable (time reversal).

```
Formula: "if row=2 then -self else self fi"
```

The formulas above were very simple and show abrupt changes in the filter near the cut-off frequencies. Smoother transitions can be created by modifying the formulas above.

In order to filter a sound we can also skip the **To Spectrum** step and apply the filter formula directly on a selected sound with the **Filter (formula)...** command. Praat then first calculates the spectrum, then applies your formula on the spectrum and transforms the modified spectrum back to a sound. The following script summarizes:

```
selectObject: "Sound s"
Filter (formula): "<some formula>"
Play
```

7.5.3. Shifting frequencies: modulation and demodulation

We can use sounds and their spectra to simulate a technique used in radio transmission or in telephone transmission. Both transmissions can only function if the frequencies of the information source, i.e. speech or music, are shifted to higher frequencies. In this section we investigate what happens if we shift the frequencies of a band-limited signal. We will use a form of so-called *amplitude modulation*. The technique we will explore is used, for example, by telephone companies to transport many conversations in parallel over one telephone cable. It is also used in radio transmission in the AM band. Although in the real world this technique is implemented in electronic circuits we can get a feeling for it by working with sounds. We will demonstrate this shifting by first combining two sounds into one sound. This is the *modulation* step. In the *demodulation* step we show how to get the separate sounds back from the combined sound. We start from the fact that a sound can be decomposed as a sum of sines and cosines of harmonically related frequencies. Let the highest frequency in the sound be F_N Hz. Suppose we multiply the sound with a tone of frequency f_1 Hz. We only have to investigate what happens with one component of the sound, say at frequency f_2 to be able to calculate what happens to the whole sound. This is the power of the decomposition method.

We start from equation (A.28) which shows that if we multiply two sines with frequencies f_1 and f_2 we can write the product as a sum of two terms.

$$\sin(2\pi f_1 t)\sin(2\pi f_2 t) = 1/2\cos(2\pi(f_1 - f_2)t) - 1/2\cos(2\pi(f_1 + f_2)t) \tag{7.6}$$

The right-hand side shows two components with frequencies that are the sum and the difference of the frequencies f_1 and f_2. Suppose that the frequency f_1 is higher than the highest frequency in the sound. The single frequency f_2 in the interval from 0 to F_N is now split into two frequency components one at f_2 Hz above f_1 and the other at f_2 Hz below the frequency f_1. This argument goes on for all components of the sound. The result is a spectrum that runs from $f_1 - f_N$ to $f_1 + f_N$, i.e. the bandwidth of the spectrum has doubled. However, this spectrum is symmetric about the frequency f_1. This means that half of the spectrum is redundant. If we high-pass filter the part above f_1 then we have a copy of the original spectrum but all frequencies are shifted up by f_1 Hz. The frequency f_1 is called the *carrier* frequency.[16] No information is lost in the shifted spectrum. We have found a technique to shift a spectrum upwards to any frequency we like by choosing the appropriate carrier frequency f_1 followed by high-pass filtering.

In the demodulation step, applying the same technique, i.e. multiplication with the carrier frequency and filtering returns the original sounds. You can easily check by working out the product $\sin(2\pi f_1 t)\cos(2\pi(f_1 + f_2)t)$ with the help of equation (A.30). In figure 7.17 the process is visualized. The top row shows the modulation of the carrier amplitude. The "information source" is on the left, the carrier frequency is in the middle panel and the modulated amplitude is on the right. The amplitude of the information source is displayed with a dotted

[16]For the technical purist, this technique is called single sideband suppressed carrier modulation(SSSCM).

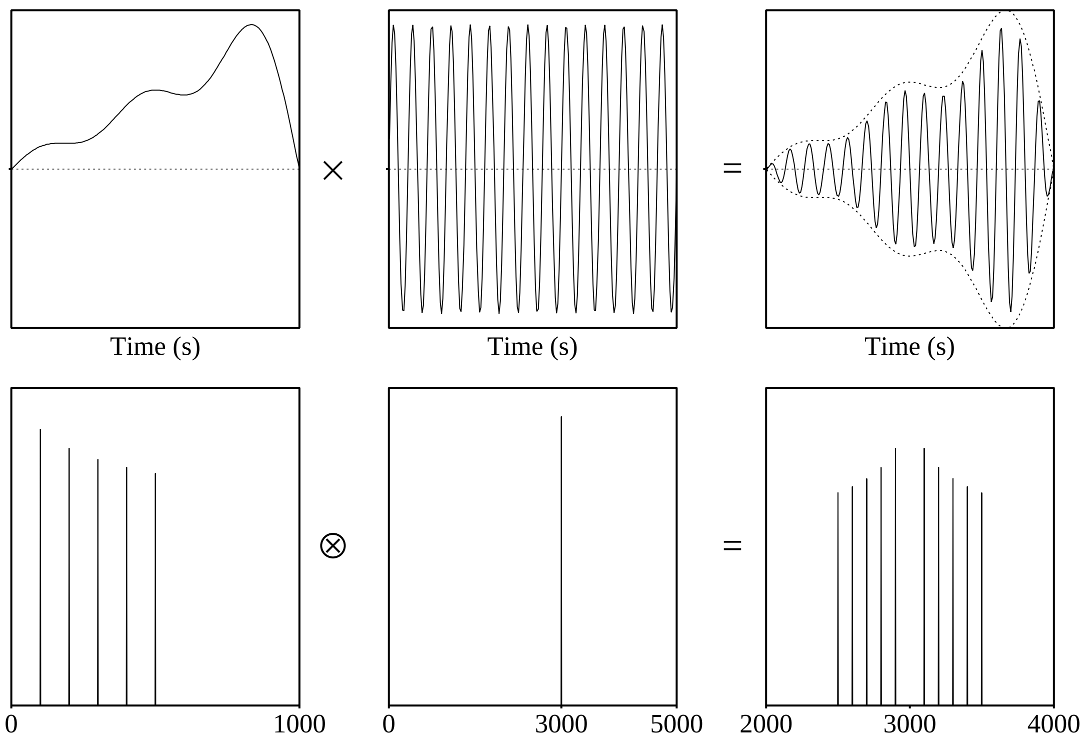


Figure 7.17.: Modulation of a band-limited sound with a carrier frequency.

line in this panel (after multiplication by a factor of 1.1 to separate it from the carrier). The second row shows the amplitude spectra of the signals in the higher row. The symmetry of the spectrum with respect to the (absent) carrier frequency is very clear.

7.6. The spectrum of a finite sound

7.6.1. The spectrum of a rectangular block function

The rectangular block function is a very important function because it is used as a windowing function. If we want to select any finite sound or part of a sound then this can be modeled as the multiplication of a sound of infinite duration with the finite block function as is depicted in figure 7.18. If the duration of the 1-part of the block is T_0, the function that describes the *spectrum* of the block varies like $\text{sinc}(fT_0)$.[17] In section A.3 a mathematical introduction

[17]The spectrum of the block function that equals 1 for x between 0 and T_0 and zero elsewhere, is given by $T_0 e^{-i\pi f T_0} \sin(\pi f T_0)/(\pi f T_0)$. The factor $e^{i\pi f T_0}$ is a phase that does not influence the amplitude of a frequency component, the factor T_0 is a scale factor that influences all frequency components with the same amount. The derivation of the spectrum is described in section A.16.2.1.

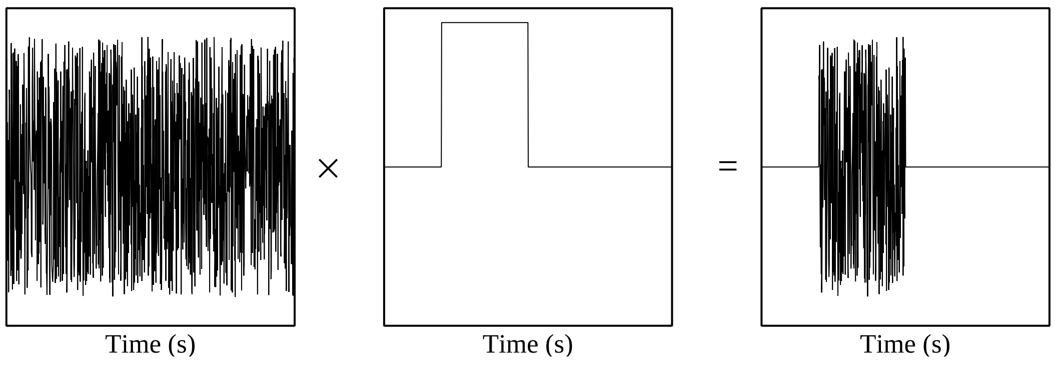


Figure 7.18.: Selecting a part from a sound by a rectangular block function.

of the sinc function is given. In the formula sinc(fT_0) f is a continuous frequency and the formula describes the continuous spectrum. The spectrum is zero for those values of f where the product fT_0 is an integer because the sine part of the sinc function is zero for these values.

If we want to calculate the spectrum from a sampled block function of "duration" T, where sample values in the first T_0 seconds equal one and zero elsewhere, we can do this by sampling the continuous spectrum at frequencies $f_k = k/T$. The values of the spectrum at the k frequency points f_k are sinc(kT_0/T). Let us investigate how this spectrum looks for various values of T_0. In figure 7.19 the amplitude spectra of block functions of varying durations are shown. The left column shows the sounds of a fixed duration of one second where the samples that have times below T_0 have value one and samples above T_0 are zero. The right column shows the corresponding amplitude spectra limited to a frequency range from 0 to 250 Hz. The lobe-valley form of the sinc function is clearly visible in all these spectra and the distance of the valleys decreases as the block's T_0 increases. If we look more carefully at these valleys we see that they are not equal. Some are very deep but most are not. Why don't they all go very deep, like for example figure A.11 shows?

The answer lies in the argument of the sinc function, kT_0/T. If the argument is an integer value, then the sinc function will be zero and only then will the amplitude spectrum show a deep valley, for all other values of the argument the sinc function will not reach zero and consequently the valleys will not be as deep. With this knowledge in mind we will now show the numbers behind these plots. For all sounds we have $T = 1$ and therefore all frequency points are at multiples of 1 Hz and k values correspond to hertz values. For the plot in the first row we have $T_0 = 0.01$ and the argument of the sinc equals $k \times 0.01$. For k equal to 100, 200, . . . this product is an integer value. The zeros in the spectrum therefore start at 100 Hz and are 100 Hz separated from each other.

In the second row the block is of 0.11 s duration. The amplitude spectrum now corresponds to sinc($k \times 0.11$). Integer values of the argument occur for k equal to multiples of 100 and the zeros are again at multiples of 100 Hz.

The third row has $T_0 = 0.22$. Now the zeros are at multiples of 50 Hz because $k \times 0.22$ is an integer value for k equal to any multiple of 50.

For a T_0 equal to 0.33, 0.44 or 0.55 as happens in rows four, five and six, the kT_0 argument is integer for multiples of 100, 25 and 20 Hz, respectively.

From the figure it is also clear that the absolute level of the peak at $f = 0$ increases. This increase corresponds directly to the increase in duration. For example, the difference between the last and the first sound is $20 \log(0.55/0.01) = 20 \log 55 \approx 34.8$ dB. This corresponds nicely with the numbers indicated in the plots: the peak for the $T_0 = 0.55$ block function in the bottom row is at 91.8 dB, the peak for the $T_0 = 0.01$ block function in the top row is at 57.0 dB. The difference between the two being 34.8 dB.

7.6.2. The spectrum of a short tone

We have a tone of a certain frequency, say f_1 Hz, it lasts for T_0 seconds and then suddenly stops and is followed by silence. We want the spectrum of this signal. In section 7.1.6 we discussed the situation for tones whose duration didn't fit an integer number of periods. Because in Fourier analysis the sine and cosine analysis frequencies are continuous and last forever the only sensible thing we could do was to analyze as if the our tone also lasts forever. The analysis pretends that the analyzed sound is a sequence of repeated versions of itself.

The actual derivation of the continuous spectrum of a short tone is mathematically too involved to be shown here.[18] Instead we show part of the amplitude spectrum of a 1000 Hz tone that lasts 0.1117 seconds. The following script generates the sound and the spectrum.

```
f1 = 1000
t0 = 0.1117
Create Sound from formula: "st", "Mono", 0, 1, 44100, "0"
Formula: "if x < t0 then sin(2*pi*f1*x) else 0 fi"
To Spectrum: "no"
```

In figure 7.20 a short part of the sound is shown around the time 0.1117 s where the tone abruptly changes to silence. In the right plot the spectrum around the 1000 Hz frequency is shown. The appearance is the already familiar sinc-like spectrum of the previous section.

We are now ready for the full truth: the spectrum of any finite tone is not a single line in the spectrum, it is more like a sinc function. Sometimes it may appear in the amplitude spectrum as one line but this is only in exceptional cases where the frequency of the tone, the duration of the tone and the duration of the analysis window all cooperate.

In general we have a "true" underlying continuous spectrum which is sampled at discrete frequency points. As we saw in the previous section for the block function, these sample points of the sinc function when applied to a sampled sound of duration T are at kT_0/T. For only one line to appear in the spectrum two conditions have to be fulfilled:

1. The frequency of the tone must be equal to one of the frequency points in the spectrum (at k/T)

2. The zeros of the sinc function are at frequency points of the spectrum.

[18] If you know a little bit about complex numbers and integrals, you can have a look at the full derivation in section A.16.2.2.

7.7. Technical intermezzo: the Discrete Fourier Transform (DFT)

Instead of working with sines and cosines separately it is custom to use complex number notation. We use the *complex sinusoid* $e^{2\pi i f t}$ to refer to both the cosine and the sine of frequency f at the same time (because of the mathematical identity $e^{2\pi i f t} = \cos(2\pi f t) + i \sin(2\pi f t)$). For a sampled sound with sample values $s_0, s_1, \ldots, s_{N-1}$ time is discretized in units of T, the sampling period. The k^{th} sample value is then at time kT.

The Discrete Fourier Transform (DFT) translates a sequency of N sample values into a sequence of coefficients of complex sinusoids. It translates a signal from the time domain to the frequency domain. The inverse transform translates a signal from the frequency domain to the time domain again. The DFT and the IDFT form a transform pair. We do not loose any information going from one domain to the other: if we start with the sampled signal and then apply a DFT and then an IDFT of the output of the DFT we get our original samples back. The transform pair is defined as:

$$\begin{aligned} S_k &= \sum_{m=0}^{N-1} s_m e^{-2\pi i k m/N} \quad \text{(analysis)} \\ s_k &= \sum_{m=0}^{N-1} S_m e^{-2\pi i k m/N} \quad \text{(synthesis)} \end{aligned} \tag{7.7}$$

The coefficients S_k in general are complex numbers and define the amplitudes of the complex sinusoids.

7.7.1. The Fast Fourier Transform (FFT)

A very fast computation of the Fourier Transform (DFT) is possible if the number of data points,N, is a power of two, i.e. $N = 2^p$ for some natural number p. In the FFT the computing time increases as $N\,{}^2\log N$, whereas the computing time of the algorithm of equation (7.7) goes like N^2. Whenever the FFT technique is used, the sound is extended with silence until the number of samples equals the next power of two. For example, in the calculation of the spectrum of a 0.1 second sound with sampling frequency 44100 Hz the number of samples involved is $0.1 \times 44100 = 4410$. This number is not a power of two, the nearest powers of two are $2^{12} = 4096$ and $2^{13} = 8192$. Therefore the FFT is calculated from 8192 values, the first 4410 values equal the sound, the next 3782 values are filled with zeros.

7.8. Sound: To Spectrum...

The way to make a spectrum is by first selecting a sound object and then clicking the `To Spectrum...` button. A form pops up and you choose OK. The choice to be made is whether you want to use the Fast Fourier Transform or not. If you choose to do so and the number of samples in the sound is not a power of two, the sound is extended with silence until the number of samples reaches the next power of two. Of course, the sound will not be changed and all this happens in the Fourier transform algorithm. The following script calculates the number of samples for the FFT.

```
nfft = 1
while nfft < numberOfSamples
    nfft = nfft * 2
endwhile
```

Because of the extension with silence, the number of samples in the sound has increased and the analysis frequencies will be at a smaller frequency distance.

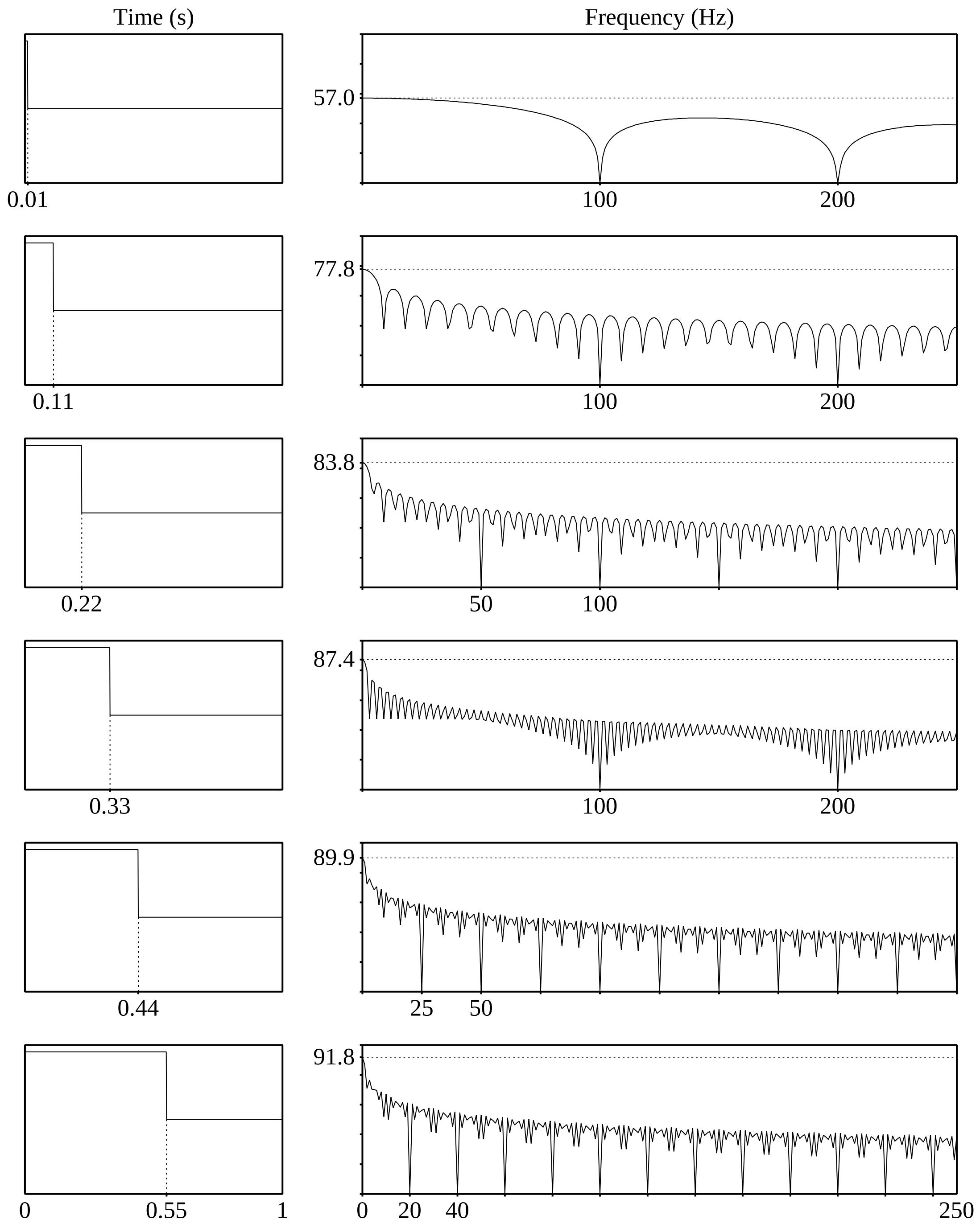


Figure 7.19.: In the left column are sampled sounds of block functions with variable durations. The block's duration T_0 is indicated below the sounds. In the right column are the amplitude spectra.

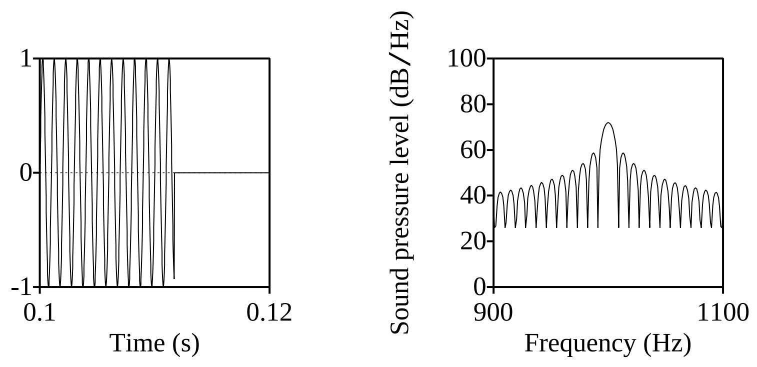


Figure 7.20.: On the left a selection from a 1000 Hz short tone. On the right the amplitude spectrum around 1000 Hz.

8. The Spectrogram

In the spectrum we have a perfect overview of all the frequencies in a sound. However, every information with respect to time has been lost. The spectrum is ideal for sounds that don't change too much during their lifetime, like a vowel. For sounds that change in the course of time, like real speech, the spectrum does not provide us with the information we want. We like to have an overview of spectral change, i.e. how frequency content changes as function of time. The *spectrogram* represents an acoustical time-frequency representation of a sound: the *power spectral density*. It is expressed in units of Pa^2/Hz.

Because the notion frequency doesn't make sense at too small a time scale, spectro-temporal representations always involve some averaging over a time interval. When we assume that the speech signal is reasonably constant during time intervals of some 10 to 30 ms we may take spectra from these short slices of the sound and display these slices as a spectrogram. We have obtained a *spectro-temporal* representation of the speech sound. The horizontal dimension of a spectrogram represents time. The vertical dimension represents frequency in hertz. The time -frequency strip is divided in cells. The strength of a frequency in a certain cell is indicated by its blackness. Black cells have a strong frequency presence while white cells have very weak presence.

8.1. How to get a spectrogram from a sound

The easiest way is to open the sound in the sound editor. If you don't see a greyish image you click Spectrum>Show Spectrogram. A number of parameters determine how the spectrogram will be calculated from the sound and other parameters determine how the spectrogram will be displayed.

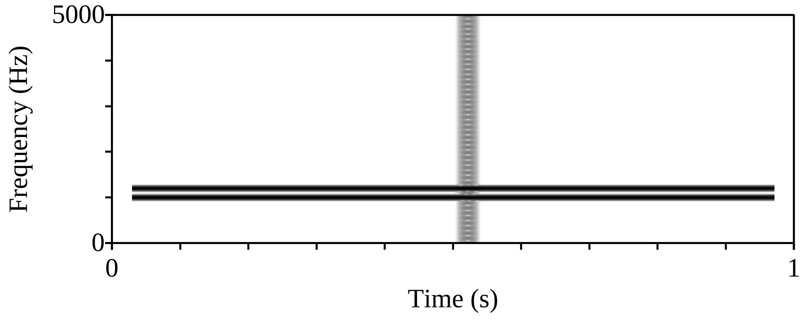
- Spectrum>`Spectrogram settings...`
- Spectrum>`Advanced spectrogram settings...`

.

8.2. Time versus frequency

The following script creates a sound that can be used to show the difference between a broad-band and a narrow-band spectrogram. The sound shows two frequencies that are 200 Hz apart and two pulses that are 9.07 ms apart.

```
Create Sound from formula: "bn", "Mono", 0, 1, 11025,
    ... "0.3*(sin(2*pi*1000*x) + sin(2*pi*1200*x)) + (col=5700) + (col=5800)"
```



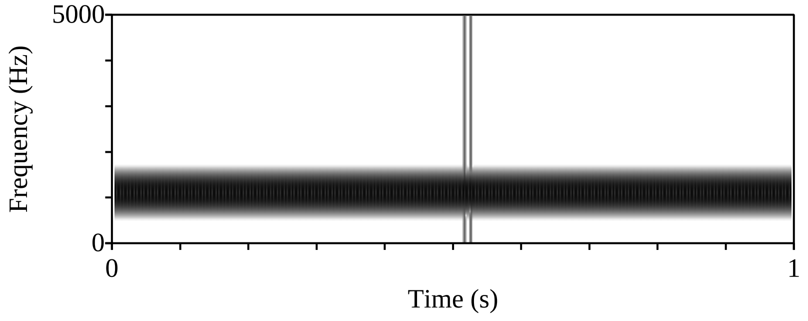


Figure 8.1.: Narrow-band versus broad-band spectrogram.

8.3. Windowing functions

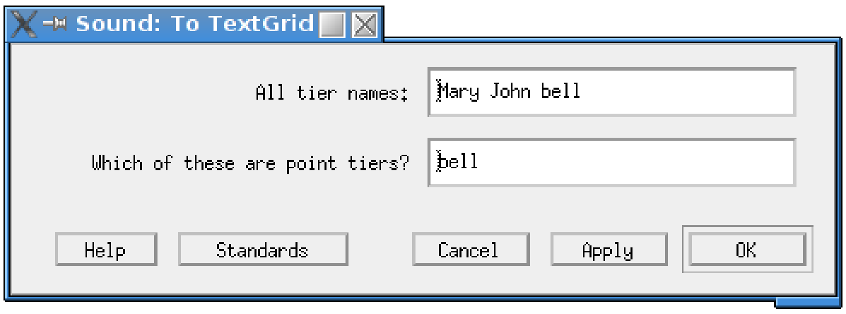
9. Annotating sounds

Annotation of sounds means adding meta data to a sound. The meta data in Praat can be any text you like. This meta data is either coupled to a time interval or to a point in time and is stored in a TextGrid object. To annotate you select a sound and a textgrid together and choose the “Edit” option. A textgrid editor appears on the screen and you can start annotating.

We start with a simple example from scratch. Select the sound that you want to annotate and then choose Annotation ->To TextGrid.... from the dynamic menu. Form 9.1 pops up and you can choose your tiers. [1] Praat distinguishes two types of tiers:

1. Interval tier. An interval tier represents a series of contiguous intervals in time. An interval is characterized with a start time and an end time where always the end time is later/larger than the start time. You typically use an intervaltier when you want to mark phenomena that extend in time like words or phonemes in a speech sound.

2. Text tier (or point tier). A text tier represents a series of marked points (sorted) in time.[2] You will typically use a text tier for labeling points in time where something interesting happens, like for example the moments of glottal closure.


Sound: To TextGrid

All tier names: Mary John bell

Which of these are point tiers? bell

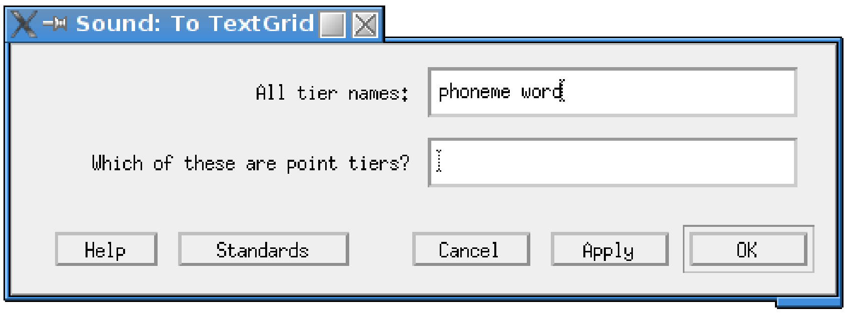
Figure 9.1.: The Sound:To TextGid... form.

The textgrid is the container for any number of interval tiers and text tiers. In the top field in form 9.1 you fill out names for all tiers you intend to use. In the bottom field you fill out which of the names in the top field are text tiers. The label of this field explicitly mentions “point” tier. If you do not intend to use text tiers or just don’t know yet, you can leave the field blank. Default all tiers will be interval tiers unless you select one or more point tiers. Praat

[1]The meaning of “tier” that is used here is “layer”. A tier can be seen as an extra layer to a sound. Therefore, in Praat a tier is a function on a domain.

[2]A text tier is a marked point process (see chapter 15).

isn't too picky about how you fill out this form, the only thing that matters is that at least one name in the top field exists. In the textgrid editor you can always add, remove or rename tiers.



All tier names: phoneme word

Which of these are point tiers?

Figure 9.2.: The Sound:To TextGrid... form for two interval tiers.

Because the example in the next section will show you how to annotate a sound at the word and at the phoneme level, we will choose a textgrid that consist of two interval tiers named "phoneme" and "word". Of course the names you can choose are free. Figure 9.2 shows how to fill out the form. Once the textgrid exist, you will select it together with the sound, choose "Edit" and a textgrid editor appears. Figure 9.3 shows a textgrideditor.
The top of the editor window shows the identification number of the textgrid (here 15) fol-

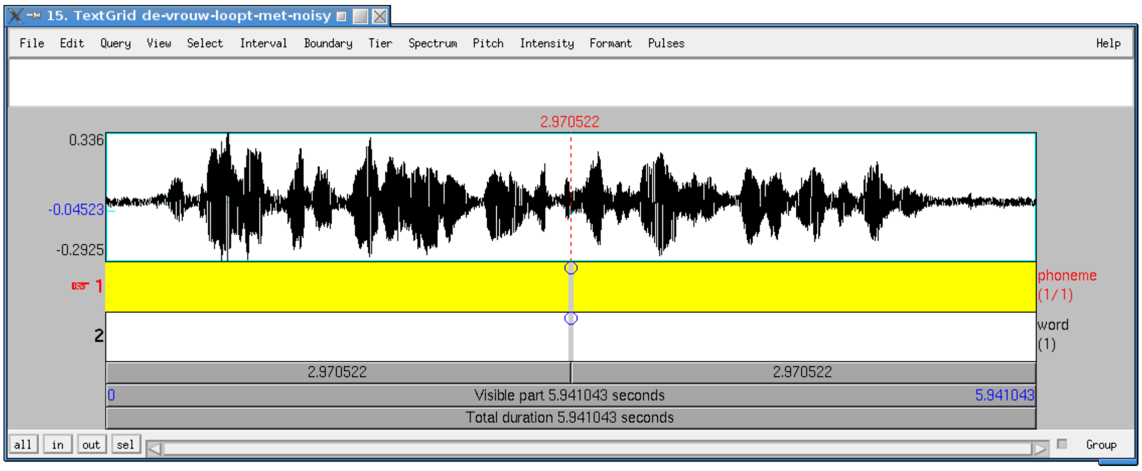


Figure 9.3.: The textgrideditor for an existing sound with a newly created textgrid that has two interval tiers.

lowed by the text "TextGrid de-vrouw-loopt-met-noisy" which happens to be the name given to the textgrid object. The next line shows all the menu choices of the textgrideditor. More on these menus will follow in the rest of this chapter. Below these menus is a white field extending over the complete width of the editor. This text window will show the text from the selected interval. It is empty now because we have not done any annotation so far. Next, in the middle panel, the sound is displayed above the two interval tiers. The top and bottom numbers at the left of the sound panel indicate the maximum and the minimum amplitude of the displayed sound, in the figure for this particular sound these numbers happen to be 0.336 and -0.2925, respectively. The blue coloured number -0.04524 indicates the value of the amplitude at the position of the cursor. The cursor is positioned in the middle of the display area and is shown with a dotted red line at position 2.970522 s which is also the midpoint of this sound of duration 5.941043 s.

In a newly created textgrid any interval tier consists of only one interval, stretching the entire time domain. The two interval tiers are shown below the sound. The interval tier number is shown on the left of the pane and the assigned name on the right. The first interval tier named "phoneme" is selected and this is indicated by the red pointing finger at the left. The selected interval is yellow. In the "(1/1)" text on the right, below the name of the selected tier, the number after the slash signals the total number of intervals in the tier while the number before the slash signals the index of the selected interval.[3] Since we have only one interval at the start and it is also selected, these numbers are both equal to one. The number between parentheses below the name of the second interval tier only shows the total number of intervals in this tier. Once you select an interval in the second tier, the pointing finger will move to the front of this tier and below the name the index of the selected interval will also be indicated.

Below the interval tiers, the gray rectangles show what parts of the sound can be played. These rectangles have the same functionality as they have in the soundeditor. The four small boxes at the bottom-left position also behave identically as the ones in the soundeditor.

9.1. Making the first annotations.

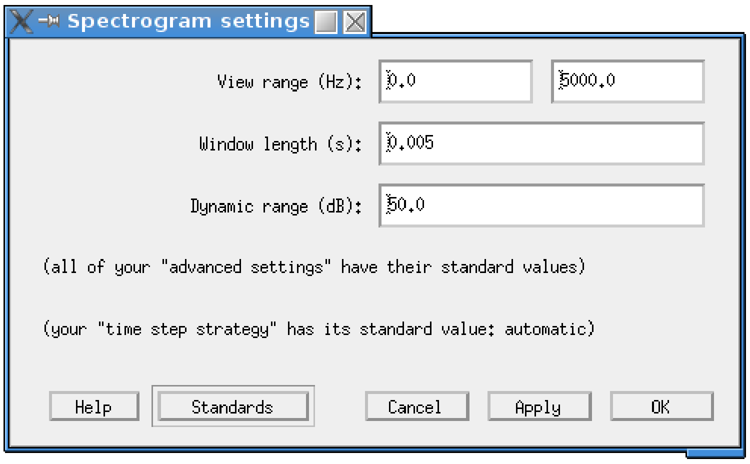
We will now show you how to make annotations at the word and phoneme level with the textgrideditor. We start with the "easy" part which is marking the start and end times of all the words in a sentence. We use the following Dutch example sentence "də vrɑu loːpt mɛt haːr dyːrə leːrə sχunə ɔp tə nɑtə døːr ɪn ət fizə ɛiχə hœys" (Dutch: De vrouw loopt met haar dure leren schoenen op de natte deur in het vieze eigen huis. Eng: The women walks in her expensive leather shoes on the wet door in the dirty house of her own). It contains the twelve Dutch vowels and the three Dutch diphthongs.[4] The example was recorded with the voice of the author on the computer with a simple microphone and a sampling frequency of 44100 Hz. This sound was displayed in the textgrid editor in figure 9.3. As one can easily see from the noisy parts at the start and the end of the sound, the quality of the recording is not very optimal. This was deliberately done because many of the recordings that people use for annotations are not of studio quality either. In this way we hope to show and be able to explain many of the problems that show up during annotation and during subsequent analysis. Because these types of microphones always show a considerable offset we have used the Sound>Modify ->Subtract mean command to compensate for this offset.

Because annotating sound is very time consuming and difficult, we like to use every bit of information that we can during this process. We will therefore also include spectral informa-

[3]This is the default behavior. With the File>Preferences... form you can modify this default.

[4]The Dutch vowel ʏ is missing in the sentence. However, the first ə in the first word də can be taken as a substitute.

tion in the editor. Before we can use this the additional information we have to make some adjustments to the textgrideditor. Choose Spectrum>Spectrogram settings... and form 9.4 pops up. The settings in the figure are the default settings for a broad band spectrogram. The window length of 0.005 s guarantees enough time resolution to be helpful during annotation. To be able to better visualize the first and second formant in the spectrogram we modify the



View range (Hz): 0.0 5000.0

Window length (s): 0.005

Dynamic range (dB): 50.0

(all of your "advanced settings" have their standard values)

(your "time step strategy" has its standard value: automatic)

Figure 9.4.: TextGridEditor: Spectrogram settings.

upper view range to 3000 Hz and then click OK. The spectrogram will become visible in a newly created pane below the sound once you have selected Spectrum>Show Spectrogram. With the spectrogram now visible in the editor we can start with marking the words in the sound. We zoom in to the first part of the sound as is shown in figure 9.5.

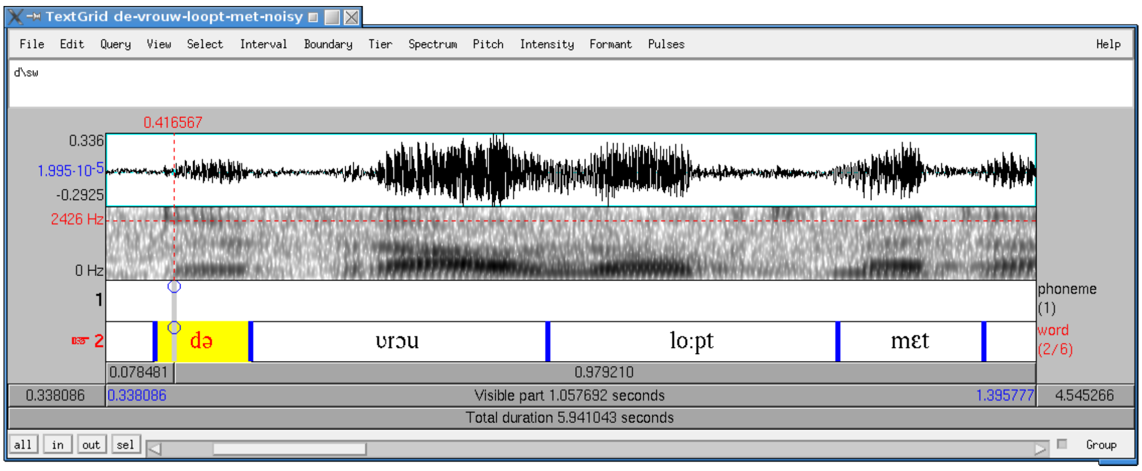


Figure 9.5.: Textgrideditor. Labeling words.

The rectangle at the bottom of the textgrideditor that stretches from left border to the right border, shows that the sound's total duration is 5.941043 s. The rectangle above it of the same width as the sound panel, shows that only 1.057692 s of the sound is actually visible in the editor. The rectangle at the bottom that shows the number 0.338086 indicates that the initial part of the sound of this duration, is not displayed in the editor. In the same way the rectangle at the bottom right shows that the final 4.545266 s of the signal aren't displayed either. The first four words in the sound have been labeled already. The second interval has been selected and its text is shown in the interval tier as də and in the text window as "d\sw". The text combination "\sw" is Praat's internal representation of the schwa symbol and guarantees that textgrids written to a file will be interchangeable on all computers.[5]

9.2. Annotate a new interval

Before you can annotate an interval, you have to create one. Creating a new interval in Praat equals splitting an existing interval into two intervals at a newly created boundary. Clicking in the sound produces a cursor at that time position in the sound panel: the red dotted line in the figure at time 0.416567. At the same time position this cursor is extended into the tiers as potential interval boundaries and these are drawn with a fat gray line. At the top of each potential boundary you see a blue circle. After clicking the circle, the gray line changes to a red color and the circle disappears: you have created one new boundary and the number of intervals has increased by one. Now you can start typing text in the newly selected interval.

9.3. Finding zero crossings

It is very important that every boundary in an interval tier is placed on a zero crossing. Failing to do so produces audible distortions when you play intervals because of discontinuities at the start and the end of sound intervals. This placing is so important that Praat has several commands for automatically finding the right positions in a sound. First of all we have three commands in the Select menu that apply only to the sound panel. Two of these commands apply to the selection and one command applies to the cursor of the sound panel. "Move start of selection to nearest zero crossing" and "Move end of selection to nearest zero crossing" find the exact position of the nearest zero crossing and move the corresponding marker to this position. The exact position of the zero crossing is found by a very precise interpolation technique. If no selection is active in the sound panel and only a cursor is present, the "Move cursor to nearest zero crossing" command is available.

To move a boundary in the interval tier to a zero crossing the Boundary>Move to nearest zero crossing command suffices.

[5] Most of the phonetic signs for vowels and consonants and many mathematical symbols are represented in Praat in a computer independent way by three character representations that always start with a backslash. For example the representations for ʊ, ɔ and ɛ are \vs, \ct and \ep, respectively. For more information about possible representations see the TextGridEditor>Help>About special symbols and Phonetic symbols.

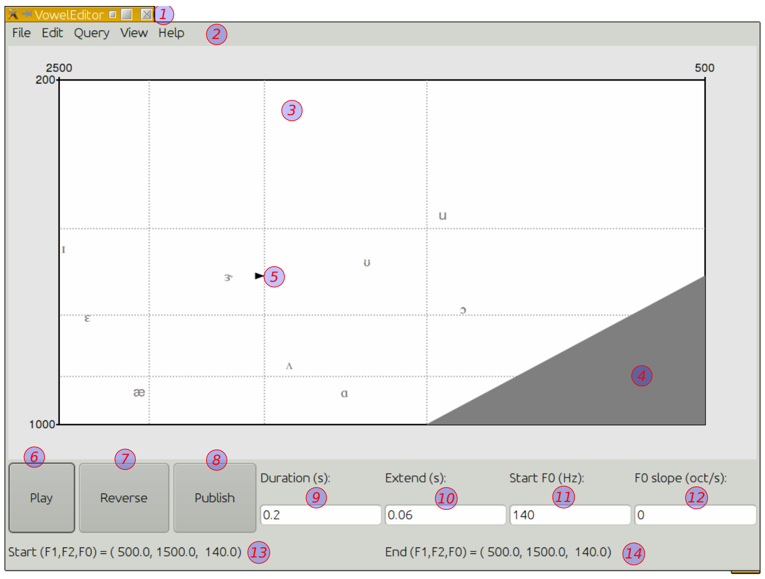
9.4. Removing an interval

To remove one interval you have to remove one boundary. If you remove a boundary the text of the interval on the right is appended to the text of interval on the left. If you don't want this to happen you either have to correct the remaining interval afterwards or you have to take care that one of the two adjacent intervals is empty before you remove the boundary. To remove a boundary you click on the boundary, the boundary changes color from blue to red and you choose Boundary>Remove boundary or simply press Alt-Backspace.

Removing annotations is much harder than adding them. This is on purpose considering the amount of work involved in positioning these boundaries.

10. The vowel editor

The vowel editor in Praat is a tool to generate vowel-like sounds by moving the mouse pointer around in the formant plane. It shows that vowels characterized by two formants only, sound reasonably well. Figure 10.1 gives an impression of the vowel editor window. This screen pops up after giving the New>Sound>Create Sound from VowelEditor... command.



Duration (s): 0.2
Extend (s): 0.06
Start F0 (Hz): 140
F0 slope (oct/s): 0

Start (F1,F2,F0) = (500.0, 1500.0, 140.0)
End (F1,F2,F0) = (500.0, 1500.0, 140.0)

Figure 10.1.: The vowel editor.

The significant parts of the vowel editor have been numbered as follows:

1. The title bar. It only shows the text "VowelEditor" and no object ID because there is no corresponding VowelEditor object in the list of objects.

2. The menu bar shows the available menu commands. Some of the available menu commands will be described later on.

3. The formant plane. This plane is spanned by two axes that represent the first and the second formant frequencies. The first formant axis runs vertically and it starts at 200 Hz and goes down to 1200 Hz. The axis has a logarithmic frequency scale. The horizontal dotted lines are at frequency steps of 200 Hz. The second formant frequency axis which runs horizontally from right to left also has a logarithmic scale. It starts at 500 Hz and

ends at 3500 Hz and shows with vertically drawn dotted lines multiples of 500 Hz. The ten vowel symbols shown in the plane are the formant frequency averages for the male speakers of American-English in the Peterson and Barney [1952] study.

4. The region where the second formant frequency would fall under the first formant frequency, indicated with a dark grey color.

5. The mouse position at start, indicated with the black arrow point. This indicates that a static vowel-like sound can be heard after hitting the “Play” button.

6. The “Play” button syhthesizes the sound from its formant trajectory an its pitch specifications (see numbers 11 and 12)

7. The “Reverse button”. Reverses the trajectory and plays it.

8. The “Publish” button. Synthesizes the trajectory and publishes the sound in the list of objects.

9. The Duration field. This field shows the duration of the trajectory. If you change this number and then hit the Play button, the sound will be synthesized having the new duration. In Fig. 10.1 the value of this field is 0.2 s.

10. The Extend field. Shows or modifies the duration of an extension of the trajectory. If you press the SHIFT button while you click the left mouse button the trajectory is extended from its end point to the position of the mouse pointer with the duration in the extend field. This gives the posibility of creating a trajectory by extending the current one. In Fig. 10.1 the extend value is 0.06 s.

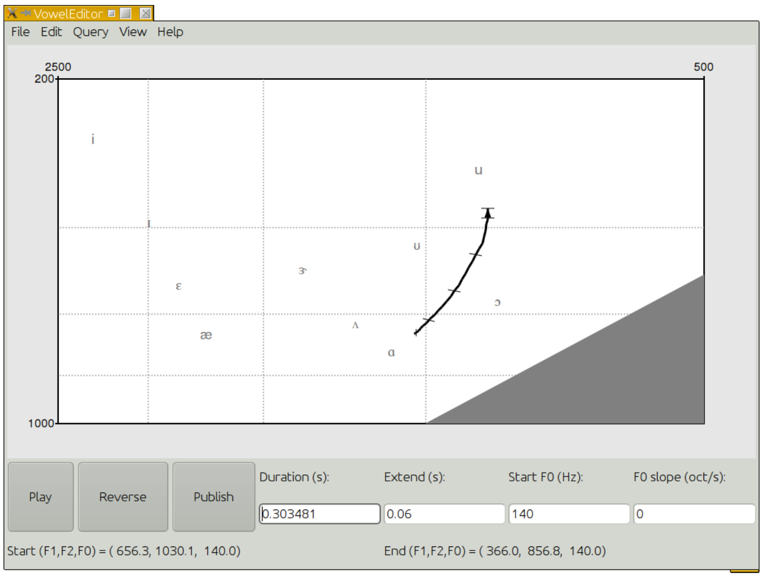
11. The Start pitch field in units of hertz. Shows or changes the initial pitch value. Vowel segments will always start with this pitch value. In Fig. 10.1 the start pitch value is 140 Hz.

12. The pitch slope field in units of octaves per second. Determines how much the pitch will change. Because the duration of a trajectory cannot be known beforehand, a lower pitch limit of 40 Hz has been chosen. The pitch slope value in Fig. 10.1 is 0 Hz/oct.

13. The trajectory start values. The values of the first two formant frequencies and the pitch at the start point of the trajectory. In Fig. 10.1 these start values are 500.0, 1500.0 and 140.0 Hz, respectively.

14. The trajectory end values. The values of the first two formant frequencies and the pitch at the end point of the trajectory. Because the trajectory consists of one point only the end values equal the begin values in Fig. 10.1 (500.0, 1500.0 and 140.0 Hz, respectively).

Moving the mouse around while keeping the left mouse button pushed down creates a trajectory in the plane (indicated while moving with a dotted blue line). After release of the mouse button, a sound is synthesized with the formant frequency values of the trajectory. Fig. 10.2, shows such a trajectory for an /aʊ/-like vowel sound. It was created by first moving the mouse pointer to the $(F_1, F_2) = (656.3, 1030.1)$ position then clicking the left mouse and moving the mouse pointer to the $(366.0, 856.8)$ position and releasing the mouse button. The small bars that cross the trajectory are located 50 ms apart and give you both an impression of the speed of the mouse movement as well as the total duration of the trajectory. We count six cross bars in this trajectory which accounts for a duration between 0.3 and 0.35 s. The duration field shows a duration of 0.303481 s. You can turn them off by using a large number for the distance value in the **Show trajectory time marks every...** command in the View menu.



Duration (s): 0.303481
Extend (s): 0.06
Start F0 (Hz): 140
F0 slope (oct/s): 0
Start (F1,F2,F0) = (656.3, 1030.1, 140.0)
End (F1,F2,F0) = (366.0, 856.8, 140.0)

Figure 10.2.: The trajectory of the diphthong /au/.

11. Digital filters

In this chapter we will be concerned with filtering in the time domain, more specifically filtering of sampled sounds. A digital filter is a system that performs operations on a sampled sound to modify certain aspects of it. In speech analysis and synthesis digital filters are commonly used. In speech analysis for example we often use a special digital filter called the pre-emphasis filter to boost certain frequencies. In speech synthesis another well known digital filter, the formant filter, is used to supply a source sound with formant structure. The initial descriptions of digital filters was rather vague so let us get more specific and start with a description of the simplest type of digital filters, the non-recursive filters.

11.1. Non-recursive filters

Non-recursive filters are those types of filters whose output can be calculated as a linear combination of *input* sample values only (there are also more complex filters, called *recursive* filters that also use previous *outputs* to calculate a result). The specification of digital filters is often done at the level of sound samples and we now introduce the notation to do so. One often refers to the value of the n-th input sound samples as x_n, where n is an index in the range from 1 to the last sample number of the sound. An output sample value is referred to as y_n. A digital filter is a system that accepts one sample value x_n at a time and then produces at that same time one output value y_n, i.e. for each sample x_n that enters the system one output value y_n results. Inside the filter all kinds of calculations can go on but in essence it is always a one-sample-in-one-sample-out system. As an example consider the very simple simple digital filter specification $y_n = x_n$. This notation is shorthand for $y_1 = x_1$, $y_2 = x_2$, $y_3 = x_3$, etc. The filter simply copies its input to its output and at each discrete time index n its output value equals its input value. Such a filter is probably not very useful. Let consider a slightly more complicated filter description $y_n = 0.5{\cdot}x_n$. This again is shorthand for $y_1 = 0.5 \cdot x_1, y_2 = 0.5 \cdot x_2$, etc. Each output sample value is one half times the input sample value, this filter attenuates all its inputs by a factor 0.5. Of course any other value instead of 0.5 can be used and we could write $y_n = a \cdot x_n$, where the *filter coefficient* a might be any real number. If a is larger than 1 the input will be amplified if a is smaller than 1 the input will be attenuated. Still not very interesting. Let us consider a more complicated filter $y_n = 0.5 \cdot x_{n-1} + 0.5 \cdot x_n$[1]. To calculate one output sample value y_n, this filter considers, besides the current sample value x_n, also the previous sample value x_{n-1}. The filter is specified by two filter coefficients which happen to be equal. In the following table the calculation to determine the output value is made explicit.

[1]Instead of the notation with subscript we often find alternative notations like y[n]=0.5x[n-1]+0.5x[n].

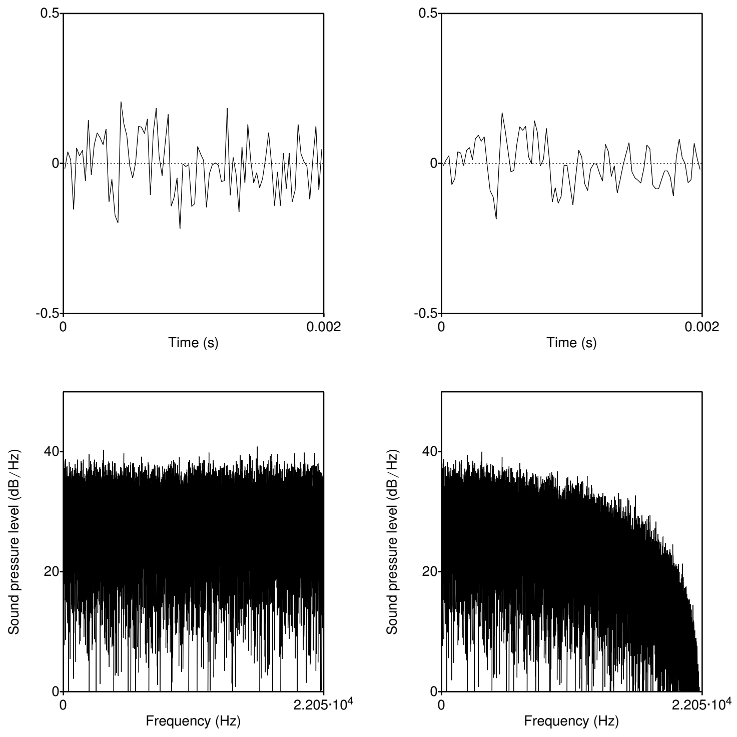


Figure 11.1.: The effect of applying the digital filter $y_n = 0.5 \cdot x_n + 0.5 \cdot x_{n-1}$ on a noise sound with a sampling period of 1/44100 s. Top panes: the first 2 ms of the noise sound (left) and the filtered noise (right). At the bottom the spectra of the 1 second duration sounds.

n	x_{n-1}	x_n	$0.5 \cdot x_{n-1} + 0.5 \cdot x_n$	y_n
1	0.0	1.0	$0.5 \cdot x_0 + 0.5 \cdot x_1$	0.5
2	1.0	2.0	$0.5 \cdot x_1 + 0.5 \cdot x_2$	1.5
3	2.0	1.0	$0.5 \cdot x_2 + 0.5 \cdot x_3$	1.5
4	1.0	3.0	$0.5 \cdot x_3 + 0.5 \cdot x_4$	2.0
5	3.0	4.0	$0.5 \cdot x_4 + 0.5 \cdot x_5$	3.5
6	4.0	3.0	$0.5 \cdot x_5 + 0.5 \cdot x_6$	3.5
..	..	..		..

The output of the filter is determined for the example input sequence (1.0, 2.0, 1.0, 3.0, 4.0, 3.0) which is shorthand for $x_1 = 1.0, x_2 = 2.0, x_3 = 1.0, x_4 = 3.0, x_5 = 4.0$ and $x_6 = 3$. The table shows step by step how each output value is calculated from two input values. For the first line after the header n equals 1, the input x_1 happens to be 1.0 and the previous input x_0 is assumed to be 0.0. The first output of the filter is therefore $0.5 \cdot 0.0 + 0.5 \cdot 1.0$ which equals 0.5, the number in the last column labeled y_n. The second line where n is 2 now shows in the x_{n-1}

column the value of the previous input x_1 (=1.0). The output of the filter for $n = 2$ is now $0.5 \cdot 1.0 + 0.5 \cdot 2.0$ which equals 1.5. If we compare the numbers in the input column labeled x_n with the numbers in the output column y_n we can easily see that there is less variation between consecutive numbers in the output column. The output is smoother than the input was.

The discussion above was a purely mathematical exercise, let us make it a little bit more operational by showing how we can apply a digital filter to a real sound. We start by making the input sound which will be named x and it will contain white noise because this makes it straightforward to see the effects of the applied filter in the spectrum, as we will see in the sequel.

```
Create Sound from formula: "x", 1, 0, 1, 44100, "randomGauss (0,1)"
```

The randomGauss function to create noise was explained in the previous section. Next we need another sound where the output of the filter can be stored. We name it *y* and create it with zero values inside as follows:

```
Create Sound from formula: "y", 1, 0, 1, 44100, "0"
```

We perform the digital filter operation by applying the following formula on the selected sound object *y*:

```
Formula: "0.5*Sound_x[col]+0.5*Sound_x[col-1]"
```

In section 4.7.1.2 we explained what goes on in a formula. The formula above essentially says: the value at the current sample number *col* in the selected sound *y* is obtained by adding together the contents at the corresponding sample number from the sound named *x* (after multiplication by 0.5) with the contents at the previous sample number from the sound x (after multiplication by 0.5). We use a special trick here in the formula because we use in the calculation sample values from an object *that is not the selected object*, i.e. from the sound named *x* (remember the sound *y* is selected and a formula applies to the selected sound). To refer to an object of type Sound named *x* we use the special *<Object type>_<object name>* syntax. In the formula the item *Sound_x[col]* refers to the contents of the sound object named *x* at sample number *col*. The formula is applied to every sample of sound *y*.
After having performed the three actions described above we have two sounds x and y. In figure 11.1 the results of the filtering are shown. In the lop left figure the first 2 ms of the noise sound *x* is shown, while the top right figure shows the first 2 ms of the filtered noise sound *y*. Although both sounds are spiky the one on the right does not fluctuate as wildly as the one on the left and seems somewhat smoother. The effect of the filter is a smoother appearance of the resulting filtered sound *y*. The smoother appearance was the result of averaging pairs of consecutive sample values. This type of filter, where the output is a linear combination of input values, is called a *moving average* filter.

11.2. The impulse response

Let us now consider a somewhat more general filter, general in the sense that we do not specify the values of the filter coefficients. The new filter uses three inputs to determine its output $y_n = a_3 \cdot x_{n-2} + a_2 \cdot x_{n-1} + a_1 \cdot x_n$. We want to determine what is called the *impulse response*, i.e. the output of the filter when the input consists of a single pulse. For a digital filter this means that you have to determine its output sample values when the input consists of a single value 1 followed by zeros, i.e. $x_1 = 1.0, x_2 = 0.0, x_3 = 0.0$,... or equivalently the input is (1.0, 0.0, 0.0, 0.0, ...). Lets do this in a table:

n	x_{n-2}	x_{n-1}	x_n	$a_3 \cdot x_{n-2} + a_2 \cdot x_{n-1} + a_1 \cdot x_n$	y_n
1	0.0	0.0	1.0	$a_3 \cdot x_{-1} + a_2 \cdot x_0 + a_1 \cdot x_1$	a_1
2	0.0	1.0	0.0	$a_3 \cdot x_0 + a_2 \cdot x_1 + a_1 \cdot x_2$	a_2
3	1.0	0.0	0.0	$a_3 \cdot x_1 + a_2 \cdot x_2 + a_1 \cdot x_3$	a_3
4	0.0	0.0	0.0	$a_3 \cdot x_2 + a_2 \cdot x_3 + a_1 \cdot x_4$	0
5	0.0	0.0	0.0	$a_3 \cdot x_3 + a_2 \cdot x_4 + a_1 \cdot x_5$	0
..	..	..	..	..	0

In the first row of the table the number n equals 1 and the filter formula then says that the output is $a_3 \cdot x_{-1} + a_2 \cdot x_0 + a_1 \cdot x_1$. We consider input values x_{-1} and x_0 to be zero. The output will therefore be equal to $a_1 \cdot x_1$ which equals a_1. For the next input, when n equals 2, the outcome $a_3 \cdot x_0 + a_2 \cdot x_1 + a_1 \cdot x_2$ will equal a_2 since x_0 and x_2 equal zero and x_1 equals 1. A same argument goes for the next input when n equals 3, resulting in an output a_3. In line 4 the x_2, x_3 and x_4 all equal zero and the output will be zero too. The rest of the outputs will also be zero. The output of the filter when the input is (1.0, 0.0, 0.0, ...) is therefore (a_1, a_2, a_3, 0.0, 0.0, ...). This impulse response has only the first three values different from zero, its impulse response is finite. Another way to classify this filter of type moving average is to say that it is a *finite impulse response* filter, or FIR filter. For this filter the impulse response, often written as h_n, is $h_n = (a_1, a_2, a_3)$. Why do we need this impulse response? Because the spectrum of the impulse response shows the frequency characteristics of the filter. For the type of non-recursive filters that we discuss in this section there is a relation between the impulse response and the filter coefficients as the current filter shows: $h_1 = a_1, h_2 = a_2$ and $h_3 = a_3$. The calculations in the table also show us another way of expressing how the output of a filter can be constructed from its input values if we use the impulse responses h_n instead of the filter coefficients a_n. The filter formula then becomes $y_n = h_3 \cdot x_{n-2} + h_2 \cdot x_{n-1} + h_1 \cdot x_n$. This shows that besides the formulation in terms of filter coefficients, we also can obtain the output as the result of multiplying the input sample values with the impulse response *backward in time* (in the formula above the x_{n-2}, x_{n-1} and x_n are ordered in time while the impulse response values are reversed in time as h_3, h_2 and h_1, i.e. x_{n-2} is multiplied by h_3, x_{n-1} is multiplied by h_2 and x_n is multiplied by h_1). The description of filtering as the multiplication of the input with the impulse response backward in time is so universal that it has gotten a special name: *convolution*. One says that the output of a digital filter can be obtained by the convolution of the input with the impulse response of the filter. The special notation is then $y_n = h_n * x_n$, where the $*$ means convolution. Note that in the praat script language, as in many other programming languages, the * is used for ordinary multiplication. If we want to write the filter output in terms of the impulse response for a filter that has an impulse response with p values h_1 to h_p then we can write that the n-th output sample will be $y_n = h_p \cdot x_{n-p-1} + ...h_3 \cdot x_{n-2} + h_2 \cdot x_{n-1} + h_1 \cdot x_n$.

The spectrum of the impulse response shows the behavior of the filter in the frequency domain. This is to say that this spectrum will show which frequencies will be attenuated and which frequencies will be amplified. As an example let us calculate the spectrum of filter described above $y_n = 0.5 \cdot x_{n-1} + 0.5 \cdot x_n$. The frequency response h_n of this filter consists of two values only (0.5, 0.5). To create the spectrum of the frequency response the following script suffices:

```
Create Sound from formula: "ir", 1, 0, 1, 44100,
    ... "if col < 3 then 0.5 else 0 fi"
To Spectrum: "no"
```

In figure 11.2 we show the spectrum of the impulse response with a solid line. Although this

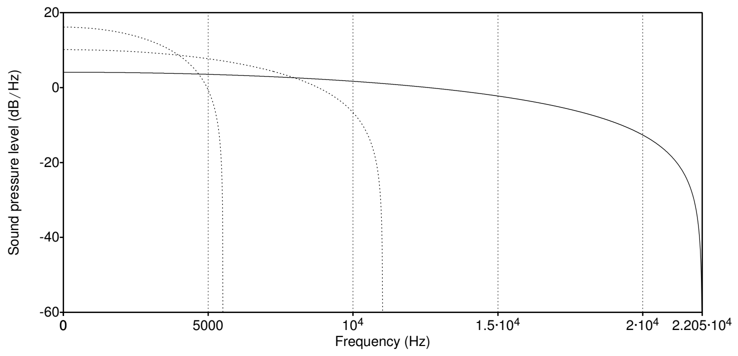


Figure 11.2.: Spectrum of the impulse response (0.5, 0.5) of the digital filter $y_n = 0.5 \cdot x_{n-1} + 0.5 \cdot x_n$ for sampling periods of 1/44100 s, 1/22050 s and 1/11025 s.

spectrum and the spectrum in the bottom right panel of figure 11.1 at first sight do look very different, we note that both spectra show the same amplitude fall-off as a function of frequency. This is no coincidence. In fact the spectrum of the filtered sound could also be obtained by multiplying the spectrum of the input sound with the spectrum of the impulse response. The two ways that we have described to determine the output of the filter are equivalent and it can be mathematically proven that the convolution of two functions is equivalent to the multiplication of their spectra. Or more generally that convolution in one domain is equivalent to multiplication in the other domain.

As a conclusion of this section we resume the very important result that we have found: the filtering process can be described in two ways that are equivalent. In the time domain filtering can be described as the convolution of the impulse response of the filter with the input sound. In the frequency domain filtering can be described as the multiplication of the spectrum of the impulse response by the spectrum of the input sound. The spectrum of the impulse response of the filter is also called the spectrum of the filter.[2]In the description of the filtering process the sampling period never entered explicitly in the formulation of filtering. Results were always obtained by the mathematical operation of multiplying numbers. The sampling

[2]In mathematical notation one writes that if s and S are the input sound and the spectrum of the input sound, respectively, and, if h and H are the impulse response of the filter and its spectrum, respectively, then $s * h \Longleftrightarrow S \cdot H$. This expresses that the spectrum of the convolution of s and h can be obtained by multiplying the spectra of s and h. Or if $s \Longleftrightarrow S$ and $h \Longleftrightarrow H$ then $s * h \Longleftrightarrow S \cdot H$. The $\Longleftrightarrow$means some thing like “form a transform pair” which translates here to “form a Fourier transform pair” since a spectrum is the Fourier transform of the sound.

period T however specifies that samples lie at distances of T s apart. At the same time, as we saw in section 3.6.4 about analog to digital conversion, the highest frequency component in the spectrum is at the Nyquist frequency which is $0.5/T$. The form of the spectrum of the filter must therefore be independent of the sampling period or, equivalently, the form of the spectrum is the same for all sampling frequencies. We can see this in figure 11.2 where the spectra of the impulse response with different sampling periods are displayed. The forms of the spectra for the different sampling periods are similar, only their frequency ranges depend on the sampling frequency, only their frequency ranges depend on the sampling frequency.

11.3. Recursive filters

For the non-recursive filters in the section above, the output values only depended on the input values. For recursive filters the output *also depends on previous outputs* which is why they are called recursive. A simple but not so useful filter to illustrate this is the filter $y_n = x_n - 0.5 \cdot y_{n-1}$. This filter specification says that in order to obtain the current output one has to subtract the previous output, scaled by 0.5, from the current input. Lets show the calculation of the impulse response of this filter in the following table.

n	x_n	y_{n-1}	$x_n - 0.5 \cdot y_{n-1}$	y_n
1	1	0.0	$x_1 - 0.5 \cdot y_0$	1.0
2	0	1.0	$x_2 - 0.5 \cdot y_1$	-0.5
3	0	-0.5	$x_3 - 0.5 \cdot y_2$	0.25
4	0	0.25	$x_4 - 0.5 \cdot y_3$	-0.125
5	0	-0.125	$x_5 - 0.5 \cdot y_4$	0.0625
6	0	0.0625	$x_6 - 0.5 \cdot y_5$	-0.03125
..	0	..	..	..
k	0	$(-0.5)^k$	$x_k - 0.5 \cdot y_{k-1}$	$(-0.5)^{k-1}$

The input of the filter is the impulse in the column labeled x_n, it is a 1 followed by zeros, i.e. $x_1 = 1, x_2 = 0, x_3 = 0, \ldots$. For n is 1 the output is equal to x_1 since we assume that the previous output y_{n-1} equals zero. For n is 2 the input x_2 equals zero and the previous output equals y_1. The output $y_2$will therefore be equal to $-0.5 \cdot y_1$which is -0.5. For n is 3 the x_3 equals 0, the previous output y_2 equal -0.5 and therefore the output y_3 will equal $-0.5 \cdot -0.5$ which equals 0.25. For n equals 4 the result will be positive again, for n equals 5 negative and so on. Although the absolute value of the output becomes smaller and smaller for increasing n, it will never be zero. This is why this filter is called an *infinite impulse response filter* or IIR filter. In fact all recursive filters are IIR filters. Because of this recursion, the calculation of the filter's impulse response becomes tedious and so we will let the computer do the calculations from now on.

The filters that have only one recursive term are not so interesting from our point of view and we move on to the filters that have two recursive terms.

11.4. The formant filter

Filters with two recursive terms that are of special interest to us have the general form

$$y_n = ax_n + by_{n-1} + cy_{n-2}, \tag{11.1}$$

where a, b and c are real numbers.[3] However, not all combinations of a, b, and c are equally interesting. It is easy to see that the coefficients b and c must be of different sign because otherwise each output would be larger than the previous output and hence the outputs could grow with every time step and eventually become infinite.[4] Only when $b^2 + 4c < 0$ interesting phenomena occur which implies that c always is a negative number. A filter where b and c obey this constraint is called a *formant filter* or *resonator*. One can show that a resonance at frequency F with bandwidth B occurs if the coefficients b and c satisfy

$$b = 2r\cos(2\pi FT) \quad \text{and} \quad c = -r^2, \tag{11.2}$$

where $r = e^{-\pi BT}$ and T is the sampling period; the resonance characteristics only depends on b and c and do not depend on the coefficient a. The a only functions as an overall scale factor and can thus be chosen to satisfy some constraint. If we choose $a = 1 - b - c$ then the filter response is normalized such that at zero frequency the filter response $H(0) = 1$. Given equations (11.2) where coefficients b and c are calculated given formant frequency F and bandwidth B, the reverse calculation is also possible, i.e. given coefficients b and c, calculate F and B:

$$B = -\frac{\ln(-c)}{2\pi T} \quad \text{and} \quad F = \frac{1}{2\pi T}\arccos\left(\frac{b}{2\sqrt{-c}}\right). \tag{11.3}$$

These formula's already make clear that the coefficient c has to be negative, otherwise $\ln(-c)$ would be undefined.

Without input the filter of course does not generate any output. To investigate how the a filter transforms its input, we have to calculate its impulse response. The following script does so. The first two lines define the formant frequency, 500 Hz, and the bandwidth of the filter, 50 Hz. In line three the sampling period is defined which is the inverse of the sampling frequency. Given these values the filter coefficients can now be calculated as happens in lines four to seven. The coefficients a, b and c in the script to 3 digits precision yield 0.00507, 1.988 and -0.993, respectively, from which we can easily calculate that $b^2 + 4c$ is negative. The penultimate line creates the output sound. The recursive filter of equation (11.1) is then implemented in the last line of the script. The first part of the formula, i.e. the part between parentheses, implements the impulse input function. In this part “col=1” refers to the very first sample in the sound. Clearly, only for the first sample the result of the complete expression between parenthesis will result in the value 1, for all other sample numbers it will result in 0.

[3] One often sees the alternative notation where the recursive coefficients have a minus sign: $y_n = a \cdot x_n - p \cdot y_n - q \cdot y_{n-2}$.

[4] It is easy to see that for a stable filter the numbers a, b, and c cannot be chosen as all positive numbers. For example, if we set $a = b = c = 1$, and we use a pulse as input, i.e. only $x_1 = 1$ and all the other inputs are zero then the output y_n would become: $y_1 = 1, y_2 = 1, y_3 = 2, y_3 = 3, y_4 = 5, y_5 = 8, \ldots$, i.e. the output would grow without bounds. Actually this particular growth sequence 1, 1, 2, 3, 5, 8, ... where each new term is the sum of the two previous ones is called a Fibonacci sequence.

Script 11.1 Script to generate the impulse response for a formant filter with a frequency of 500 Hz and a bandwidth of 50 Hz.

```
f1 = 500
b1 = 50
dT = 1 / 44100
r = exp (-pi * b1 * dT)
c = - r^2
b = 2 * r * cos (2 * pi * f1 * dT)
a = 1 - b - c
y = Create Sound from formula: "y", "Mono", 0, 0.1, 44100, "0"
Formula:
    ... "(if col = 1 then 1 else 0 fi) + b * self[col-1] + c * self[col-2]"
```

The recursive part of the filter is implemented by the last two expressions and there is explicit reference to two previous sample numbers.[5] In figure 11.3 we show 0.015 s of the impulse

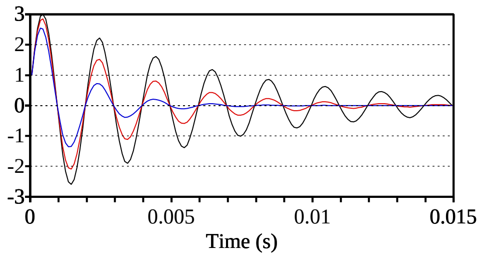


Figure 11.3.: The impulse response from the digital formant filter (11.1)with formant frequency 500 Hz and bandwidths 50 Hz (black), 100 Hz (red) and 200 Hz (blue).

response of a formant filter with constant frequency of 500 Hz for three different bandwidths, 50 Hz in black, 100 Hz in red and 200 Hz in blue. Again, like the example in the previous section, we see that despite the fact that the input was limited in duration, i.e. there was only one sample in the input that differed from zero, the output of the filter is not limited in duration. The output of the formant filters, as displayed in the figure looks like a sine function whose amplitude gradually decreases. We count approximately five periods in the first 0.01 s and conclude that this sine component has a frequency of approximately 500 Hz. In fact the output of this filter is a damped sinusoid that can be described by the following function

$$s(t) = Ae^{-\pi Bt}\sin(2\pi Ft + \phi), \tag{11.4}$$

where F is the formant frequency, B is the bandwidth, A is the amplitude and ϕ is the phase at $t = 0$. The function shows that the amplitude of the sine wave with frequency F decreases

[5]The formula line in the script can also be shortened to:
```
Formula...  if col=1 then 1 else b * self[col-1] + c * self[col-2]
```

exponentially with time. The exponent which determines the rate of decay, depends on the bandwidth. The larger the bandwidth the faster the amplitude decreases. The signals in figure (11.3) show this: the damped sine in black has the smallest bandwidth, the red one the largest. We already showed that the frequency of the sines in this figure were all 500 Hz. We can also check for the bandwidths in the following way. We measure the amplitudes at two different times, t_1 and t_2, where the phases of the sine components are equal to each other.[6] At times where the sine is a maximum, the amplitude at time t_1 is $s(t_1) = Ae^{-\pi B t_1}$ and at time t_2 is $s(t_2) = Ae^{-\pi B t_2}$. Dividing these two expressions gives $s(t_1)/s(t_2) = e^{-\pi B(t_1 - t_2)}$, from which we express B as $B = \ln(s(t_1)/s(t_2))/(\pi(t_1 - t_2))$. To calculate B we take the first time point at the first maximum of the black sine and the second time point seven periods later, at the last maximum. This gives us $s(t_1) \approx 3$, $s(t_2) \approx 0.33$ and $t_1 - t_2 = 7 \times 0.002$, and we calculate $B \approx 50.2$. Given the uncertainties in the amplitude determination this is very close to the 50 Hz bandwidth that we used to generate the impulse response. Therefore, the agreement of the signal in figure 11.3 with the function in equation (11.4) is excellent.

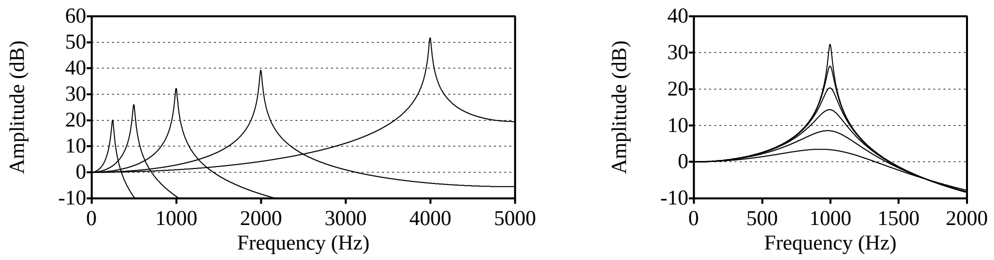


Figure 11.4.: The effect of formant frequency and bandwidth on the filter amplitude of equation (11.5). On the left: keeping bandwidth constant at 50 Hz and doubling the formant frequency, starting at 250 Hz. On the right: keeping the formant frequency constant at 1000 Hz and doubling the bandwidth, starting at 25 Hz.

It can be shown that the filter response of a digital filter with formant frequency F and bandwidth B is:

$$|H(f)| = \frac{|a|}{\sqrt{2r^2 \cos 4\pi fT - (4r\cos\theta + 4r^3\cos\theta)\cos 2\pi fT + r^4 + 4r^2\cos^2\theta + 1}}, \tag{11.5}$$

where $\theta = 2\pi FT$. In terms of the coefficients a, b, and c, the filter response would read:

$$|H(f)| = \frac{|a|}{\sqrt{1 + b^2 + c^2 + 2b(c-1)\cos 2\pi fT - 2c\cos 4\pi fT}} \tag{11.6}$$

In figure 11.4 we show the filter responses of equation (11.5) for varying formant frequencies and bandwidths. In the left display we show the effect of doubling the formant frequency

[6]Of course "equal" in the sense of "equal modulo 2π". At times t_1 and t_2 we choose the phases to be equal $2\pi F t_1 + \phi = 2\pi F t_2 + \phi + k2\pi$. This can be simplified as $t_1 = t_2 + k/F$. This boils down to choosing times that lie at integer periods apart.

F while keeping the bandwidth at the constant value of $B = 50\,\text{Hz}$. It shows that the formant amplitude is proportional to frequency: for each doubling of the formant frequency F, the formant amplitude increases approximately with 6 dB. The display on the right of the figure shows how the formant amplitude changes when we keep the frequency constant at $F = 1000\,\text{Hz}$ while we vary the bandwidths by doubling them, starting at $B = 50\,\text{Hz}$. Formant amplitude is approximately inversely proportional to bandwidth because a doubling of bandwidth results in a 6 dB decrease in amplitude.

11.5. The antiformant filter

With antiformant filters, also called *notch* filters, we can model the absence of frequencies at a certain frequency range in the spectrum. This absence can be noticed by dips in the frequency spectrum. These dips, however, are very difficult to hear because our perceptual system is much more oriented towards the detection of spectral peaks than towards the detection of spectral valleys. The antiformant filters are implemented in the standard way as digital antiresonators, i.e. as second order digital filters:

$$y_n = a'x_n + b'x_{n-1} + c'x_{n-2}. \tag{11.7}$$

In contrast to the digital formant filter 11.1 this filter is non-recursive and therefore always stable for all values of a', b', and c'. Because the filter has no output recursion terms it can not feed itself once started, and therefore the impulse response is finite and only three samples long. The reason we did not mention this filter at the start of this section where we treated non-recursive filters is the special relation it bears to formant filters.

For special combinations of the coefficients this filter removes the frequencies in an interval. However if we want an antiformant with frequency F and bandwidth B, we can easily get the values for the coefficients in the following way. We first calculate coefficients a, b, and c as if it were a formant filter:

$$\begin{aligned} c &= -r^2 \\ b &= 2r\cos 2\pi FT \\ a &= 1 - b - c \end{aligned}$$

where $r = e^{-\pi BT}$ and T is the sampling time. The second, and last step, is a simple transformation of these coefficients:

$$\begin{aligned} a' &= 1/a \\ b' &= -b/a \\ c' &= -c/a. \end{aligned}$$

It can be shown that the frequency response of an antiformant filter is:

$$H(f) = \sqrt{a'^2 + b'^2 + c'^2 + 2b'(a'+c')\cos 2\pi fT + 2a'c'\cos 4\pi fT} \tag{11.8}$$

In figure 11.5 we show the antiformant filters with the same frequencies and bandwidths as the formant filters of figure 11.4. It is clear that the antiformant filters look very much like the inverse of the formant filters. However, some of the side effects of these filters are undesirable.

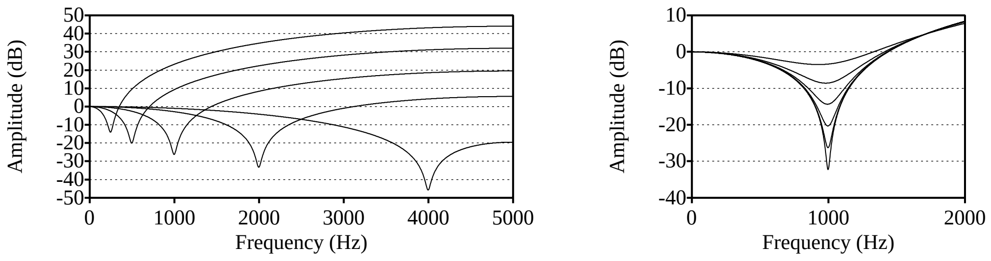


Figure 11.5.: The effect of antiformant frequency and bandwidth on the filter amplitude of equation (11.8). On the left: keeping bandwidth constant at 50 Hz and doubling the antiformant frequency, starting at 250 Hz. On the right: keeping the antiformant frequency constant at 1000 Hz and doubling the bandwidth, starting at 25 Hz.

The left display of figure 11.5 shows the frequency response of an antiformant filter for varying antiformant frequencies. We note a considerable boost of the high frequencies: the lower the antiformant the more the high frequencies are boosted; the antiformant filter almost acts like a high-pass filter. This is undesirable: we want an antiformant filter to remove frequencies within a certain frequency range and *not* to boost frequencies outside that region.

11.6. Applying a formant and an antiformant filter

In the preceding section we saw that removing frequencies from the sound by applying an antiformant filter had the negative effect of boosting the high frequencies. The remedy for this behaviour is to have this high-pass effect counteracted by a nearby formant. We know that a formant filter attenuates frequencies above and below its centre frequency. The effect of the high-frequency boost of the antiformant filter might therefore be counteracted by the high-frequency fall off of the formant filter. By applying these two filters in succession, one tries to do this. If the two filters would have the same centre frequencies and bandwidths they would balance each other and there would hardly be any effect at all. Therefore both filters need different frequencies and or bandwidths. We need to find out what happens if we combine a formant and an antiformant filter with different centre frequencies and/or bandwidths.

In figure 11.6 we investigate some of the possible formant-antiformant combinations. In the display on the top left, a constant formant at zero frequency with a small bandwidth of 25 Hz is combined with antiformants of the same constant bandwidth of 25 Hz but at varying antiformant frequencies. These antiformant frequencies start at 250 Hz and are doubled each time to frequencies 500 Hz, 1000 Hz, 2000 Hz and 4000 Hz. The spectra in the top left display show that, at frequencies above the antiformant frequency, the spectra are almost flat which

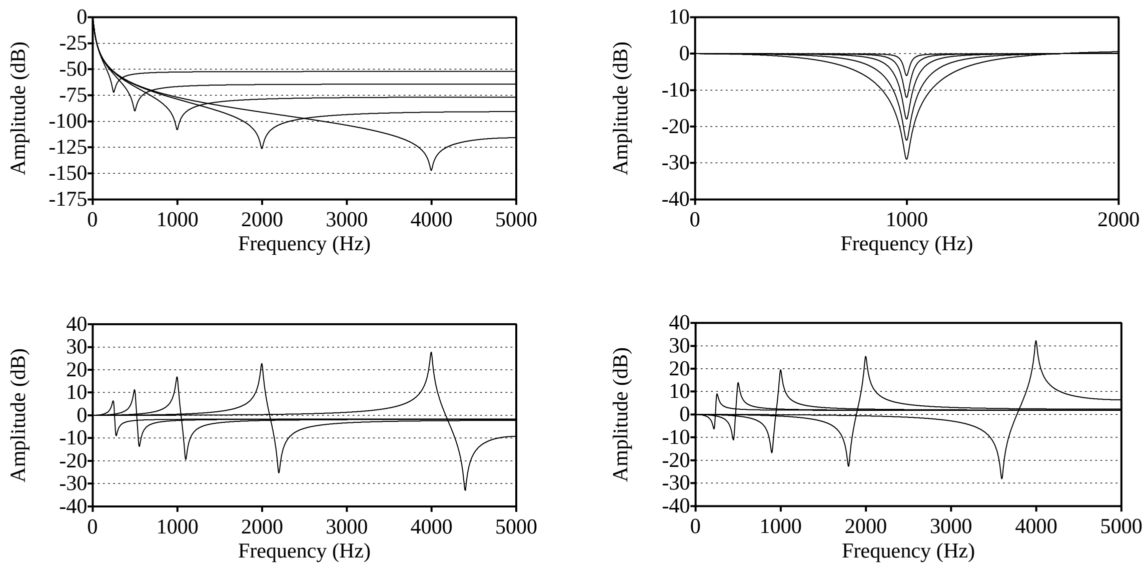


Figure 11.6.: The effect of combining a formant filter with an antiformant filter. Top left: constant resonator at $f = 0$ Hz with antiresonators at several frequencies. Bottom left: formant at somewhat lower frequency than antiformant. Bottom right: formant at somewhat higher frequency than antiformant. Top right: antiformant and formant with same frequency but different bandwidths.

is a good thing. However, below this frequency a sharp low-pass filter effects results which is very undesirable. For the lowest frequency, 250 Hz, the flat part is already 40 dB below the response at zero frequency. For the 4000 Hz antiformant this level is almost -100 dB! This is not very useful for synthesis.

In the two displays at the bottom of figure 11.6 we show what happens if we combine a formant and an antiformant whose frequencies differ only by a small amount and a small 25 Hz bandwidth. In the bottom left display the antiformant frequency is 10% higher than the formant frequency and in the bottom right display it is 10% lower. The effect is the combination of a peak and a valley, or a valley and a peak, respectively, on a rather flat spectrum. This is usable if we want to add a peak and valley without having this any influence on the existing spectrum.

In the display on the top right we combined a formant and antiformant of the same frequency but with different bandwidths. The frequency is constant and 1000 Hz. The bandwidth of the antiformant filter was fixed at 25 Hz and the bandwidth of the formant filter started at 50 Hz and was doubled in each step to 100 Hz, 200 Hz, 400 Hz and 800 Hz. A perfect valley results without any other effect on the spectral amplitude. The best result is obtained when the bandwidth is approximately 500 Hz. If the bandwidth is larger, the valley will not become any deeper but instead the flatness of the spectrum disappears and especially the higher frequencies will be amplified substantially.

12. The KlattGrid synthesizer

12.1. Introduction

The KlattGrid synthesizer in Praat is modeled after the one described in the Klatt and Klatt [1990] paper. Fig. 12.1 displays a possible synthesizer.

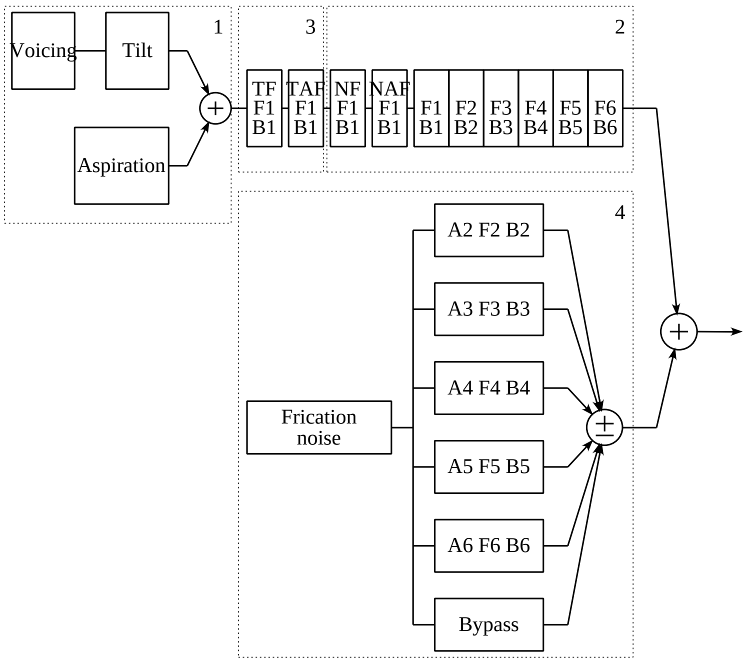


Figure 12.1.: A KlattGrid with vocal tract part as filters in cascade and frication part as filters in parallel.

The KlattGrid synthesizer consists of four parts. Each part is controlled by a number of parameters.

1. The phonation part. This part generates the source signal that drives the vocal tract filter. In the figure this part is situated at the top left and shows a part responsible for voicing and a part responsible for aspiration.

2. The vocal tract part. This part filters the source signal with formant and/or antiformant filters that are either in cascade or in parallel. In the figure this part is displayed at the top and connects directly with the phonation part. It shows a number of labeled boxes where each box represents one filter. The boxes labeled NF and NAF represent one nasal formant and one nasal antiformant filter, respectively. The boxes that only have labels like, for example, F_1 and B_1, represent the six oral formant filters. The F's and B's represent the user-controlled formant frequencies and formant bandwidths, respectively.

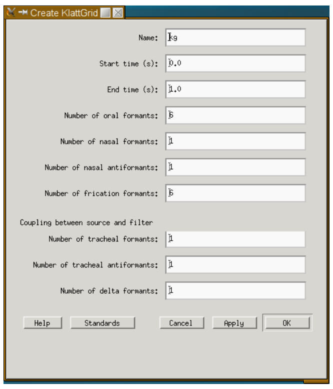
3. The coupling between phonation and vocal tract. Only one part of the coupling is drawn, this part is the tracheal formant and antiformant at the start of the cascade filtering. The part that is not displayed may modify formant frequencies and bandwidths during the open phase of the glottis.

4. The frication part. In the figure it is shown at the bottom and its output is added with the output of the vocal tract part. It is realised as a noise source that is filtered by set of parallel filters. These filters have the amplitudes A as extra user-controlled variables. There is an extra bypass pathway for the noise to mix with the filtered noise.

Before going into the details of these four parts in the KlattGrid, we first show you how you can create a KlattGrid.

12.1.1. How to create an empty KlattGrid

A KlattGrid object will be created from values supplied by the form that appears if you click the "Create KlattGrid..." option in the "Acoustic synthesis (Klatt)" cascading menu from the New menu. There are quite a lot of options that can be chosen in this form. The most obvious



Name:	kg
Start time (s):	0.0
End time (s):	1.0
Number of oral formants:	6
Number of nasal formants:	1
Number of nasal antiformants:	1
Number of frication formants:	6
Coupling between source and filter	
Number of tracheal formants:	1
Number of tracheal antiformants:	1
Number of delta formants:	1

Figure 12.2.: The `Create KlattGrid...` form.

ones are: the name you want to assign to the KlattGrid and the start and end time. The defaults for the number of formants of a specific type are in conformance with the maximum numbers that were possible in the original Klatt editor as described in the Klatt and Klatt [1990] article. However, in a KlattGrid these numbers are not limited in any way.[1] The choices you make

[1]See section 12.6 for more differences between KlattGrid and the Klatt synthesizer.

here are not definitive nor mandatory, you just specify what you think you might need. For example, if you accept these defaults and click OK and you do not set any tracheal formant frequency value and bandwidth, then the tracheal formant frequency filters will not be used during synthesis. You might also increase the number of formants at a later time if you need to.

Once the KlattGrid is created, the dynamic menu changes, as is depicted in the left display of Fig. 12.3. Notice that some menus have been split to avoid too many choices to pop up. The Edit menu has been split into two parts like the query part. The first split is always for the phonation part. Because of the many options to modify the KlattGrid this menu has been split into four parts, one for each of the four parts of the synthesizer. The KlattGrid as defined

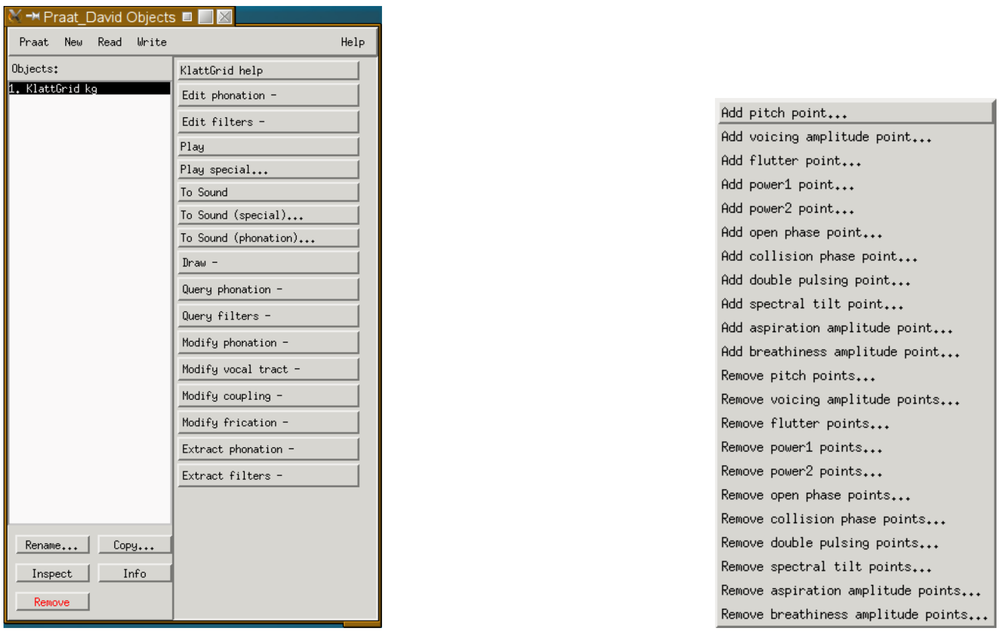


Figure 12.3.: On the left KlattGrid's dynamic menu, on the right the cascading menu options available from the "Modify phonation -" button.

now is empty: none of the tiers has any points defined in it. If you would try to play the empty KlattGrid, you would receive a warning that Praat cannot execute this command. The right display of figure 12.3 show the available options from the cascading "Modify phonation -" menu button. The options in this menu are divided into two parts. On top are the options to add points to one of the phonation tiers like the pitch tier and the voicing amplitude tier. The bottom part shows options to remove points from the phonation tiers. Before we explain the KlattGrid in detail we will first make a small excursion on how to synthesize some basic sounds.

12.1.2. How to create an /a/ and an /au/ sound

Now we know how to create a KlattGrid, we first show you with a simple script what you can do with it and how to synthesize a basic sound.

Script 12.1 The script to create the a and the au sounds.

```
 1  Create KlattGrid: "kg", 0, 0.3, 6, 1, 1, 6, 0, 0, 0
 2  Add pitch point: 0.1, 120
 3  Add voicing amplitude point: 0.1, 90
 4  Play
 5  Add oral formant frequency point: 1, 0.1, 800
 6  Add oral formant bandwidth point: 1, 0.1, 50
 7  Add oral formant frequency point: 2, 0.1, 1200
 8  Add oral formant bandwidth point: 2, 0.1, 50
 9  Play
10  Add oral formant frequency point: 1, 0.3, 300
11  Add oral formant frequency point: 2, 0.3, 600
12  Play
```

In the first line of script 12.1 we define an empty KlattGrid with a duration of 0.3 s. The second line starts filling the KlattGrid with information, it defines a pitch point of 120 Hz at time 0.1 s. The next line adds a voicing amplitude point of 90 dB at the same time. With the pitch and voicing amplitude defined, there is enough information in the KlattGrid to produce a sound and we can now Play the KlattGrid in line 4. During 300 ms you will hear some unidentifiable sound. It is the sound as produced by the glottal source alone. This sound normally would be filtered by a vocal tract filter. But we have not defined the vocal tract filter yet and in this case the vocal tract part will just behave like an all-pass filter for the phonation sound.

In lines 5 and 6 we add a first oral formant with a frequency of 800 Hz at time 0.1 s, and a bandwidth of 50 Hz also at time 0.1 s. The next two lines add a second oral formant at 1200 Hz with a bandwidth of 50 Hz.

If you now play the KlattGrid, at line 9, it will sound like the vowel /a/, with a constant pitch of 120 Hz.[2]

Lines 10 and 11 add some dynamics to this sound; the first and second oral formant frequency are set to the values 300 and 600 Hz of the vowel /u/; the bandwidths do not change and stay constant as they were defined in lines 6 and 8. In the interval between times 0.1 and 0.3 s, the oral formant frequencies will be interpolated. The result will now sound approximately as an /au/ diphthong.[3] The script shows that with only a few commands we already can create interesting sounds. In the next sections we will discuss all the tiers that we can use to modify the characteristics of a sound. We will start with the phonation part.

[2]Because of the constant extrapolation in a tier, we need to specify only one value in a tier if we want to keep the parameter constant over the domain of the tier.

[3]As you see in section 12.2, the specification in script 12.1 is not sufficient because the tiers that specify the glottal flow function have not been defined. However, in absence of any points in these tiers a default open phase of 0.7 is used and the power1 and power2 values are default set to 3 and 4, respectively.

12.2. The phonation part

In this section we will show you how the phonation part of the KlattGrid can be specified or modified. The phonation part serves two functions:

1. It generates voicing. Part of voicing are the timings for the glottal cycle. The part responsible for these timings is shown by the box labeled “Voicing” in figure 12.1. These timings serve three purposes:
   a) It gives the start and ending times for the glottal flow function.
   b) It gives the start and ending times for the generation of breathiness. Breathiness is noise that occurs only during the open phase of the glottis.
   c) It gives the start and ending times for the delta formants in the coupling section. Formant frequency values and bandwidths may change during the open phase of the glottis if delta formants have been specified.
2. It generates aspiration. This function is indicated by the box labeled “Aspiration” in figure 12.1. In contrast with breathiness, aspiration may take place independently of any glottal timing.

The phonation part in the KlattGrid can be modeled by one or more of the following tiers.

Pitch tier For voiced sounds the pitch tier models the global fundamental frequency as a function of time. Pitch equals the number of glottal open/closing cycles within a time interval. In the absence of flutter and double pulsing the pitch tier is the only determiner for the instants of glottal closure. The “Extract>Extract PointProcess (glottal closures)” command will create a PointProcess with the glottal closure times in it. A PointProcess is a sequence of times on a domain (see section 15).

Voicing amplitude tier The voicing amplitude regulates the maximum amplitude of the glottal flow in dB SPL. The reference amplitude at 0 dB is $2 \cdot 10^{-5}$. This means that a flow with amplitude 1 corresponds to $20 \log 1/(2 \cdot 10^{-5}) \cong 94$ dB SPL. To produce a voiced sound the voicing amplitude tier may not be empty.

Flutter tier The flutter models a kind of “random” variation of the pitch and is input as a number between zero and one. This random variation can be introduced to avoid the mechanical monotonic sound whenever the pitch remains constant during a longer time interval. The fundamental frequency is modified by a flutter component according to the following semi-periodic function that we adapted from the Klatt and Klatt [1990] article $F_0'(t) = 0.01 \cdot \text{flutter} \cdot F_0(\sin(2\pi 12.7t) + \sin(2\pi 7.1t) + \sin(2\pi 4.7t))$. In Fig. 12.4 we display the relative variation of the fundamental frequency $r(t) = 0.01(\sin(2\pi 12.7t) + \sin(2\pi 7.1t) + \sin(2\pi 4.7t)$. The graph clearly shows the semi-periodicity character of the pitch variation. It also shows that the maximum pitch change is around three percent if the flutter variable is at a maximum. For a constant pitch of say 100 Hz and flutter = 1, the synthesized pitch may vary between approximately 97 and 103 Hz. Script 12.2 synthesizes a glottal source sound with a duration of 0.5 s, a fundamental frequency of 100 Hz and maximum flutter.

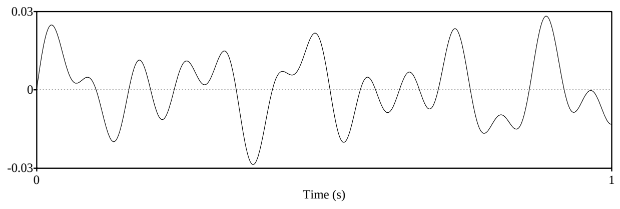


Figure 12.4.: The Klatt flutter function for flutter = 1.

Script 12.2 Script to hear the effect of adding flutter to a monotone sound.

```
Create KlattGrid: "kg", 0, 0.5, 6, 1, 1, 6, 0, 0, 0
Add pitch point: 0.25, 100
Add voicing amplitude point: 0.25, 90
Add flutter point: 0.25, 1
```

Open phase tier The open phase tier models the open phase of the glottis with a number between zero and one. The open phase is the fraction of one glottal period that the glottis is open. The open phase tier is an optional tier, i.e. if no points are defined then a sensible default for the open phase is taken (0.7). In figure 12.5 in the top panel we show a short series

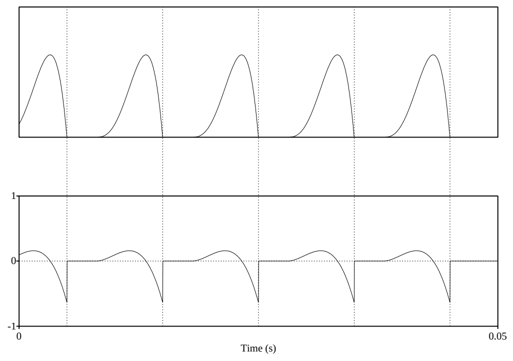


Figure 12.5.: On top a series of glottal flow pulses, at the bottom the derivative of the flow. The moments of glottal closures have been indicated with a vertical dotted line. The open phase was held at the constant value of 0.7, the pitch was fixed at 100 Hz and the amplitude of voicing was fixed at 90 dB.

of glottal pulses with an open phase of 0.7. The moments of glottal closure are indicated with dotted lines. A glottal period is the time between two successive glottal closure markers. In the figure each glottal period starts with an interval where the flow amplitude is zero, this part is called the *closed phase*. After the closed phase we enter the open phase where flow starts slowly to increase and then after reaching a maximum decreases rapidly. The exact moment in time where the open phase starts can be difficult to locate from the figure because of the slow increase of the flow function at the start of the open phase (i.e. for small values of t in equation (12.1) in the next section).

In the bottom panel the derivative of the glottal flow is shown and one can clearly see that the extreme values of the derivative are at the moments of glottal closure where the derivative function shows negative values. From these minimums the function instantly jumps to zero at the start of the next closed phase.

Power1 and power2 tiers The power1 and power2 model the form of one glottal flow function during one open phase of the glottis as

$$\text{flow}(t) = t^{\text{power1}} - t^{\text{power2}} \tag{12.1}$$

for $0 \le t \le 1$. Here $t = 0$ is the time the glottis opens and at $t = 1$ the glottis is closed. The flow is the result of a rising polynomial, t^{power1}, counteracted by a falling polynomial, $-t^{\text{power2}}$. To have a positive flow, i.e. air going out of the lungs, power2 has to be larger than power1. For synthesis, these tiers are optional and if absent, default values power1 = 3 and power2 = 4 will be used. Figure 12.6 on the left shows the flow function from equation (12.1)

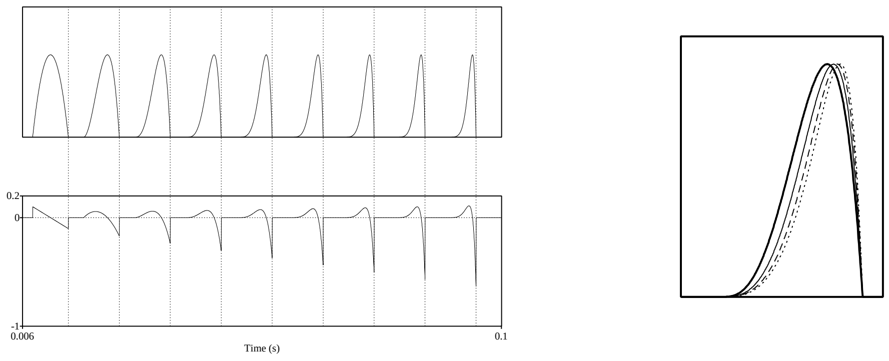


Figure 12.6.: The effect of the exponents in equation (12.1) on the form of the flow function. Left: On top, nine glottal pulses synthesized with different (power1, power2) combinations. Power1 increases linearly from 1, and always power2 = power1 + 1. Consequently, the first pulse on the left has power1 = 1 and power2 = 2, while the last one on the right has power1 = 9 and power2 = 10. The bottom panel on the left shows the derivatives of these flow functions. Moments of glottal closure have been indicated with a vertical dotted line. The open phase was held at the constant value of 0.7, the pitch was fixed at 100 Hz and the amplitude of voicing was fixed at 90 dB. Right: the flow function for constant power1 = 3 but varying power2. Power2 values are: 4: thick solid line; 6: thin solid line; 8: dashed line; 10: dotted line.

for different combinations of the exponents. Power1 increases linearly starting with one, and always has power2 = power1 + 1. Consequently, the first pulse on the left has power1 = 1 and power2 = 2, the last one on the right has power1 = 9 and power2 = 10. One easily notices the gradual change in shape if the powers get at higher numbers. Although the open phase was kept constant, it gradually takes longer time to reach a reasonably large value for the flow.

It can be easily shown that the maximum flow in equation (12.1) occurs at a value

$$t_0 = \left(\frac{\text{power1}}{\text{power2}}\right)^{1/(\text{power2}-\text{power1})}$$

and this flow equals

$$f(t_0) = \left(\frac{\text{power1}}{\text{power2}}\right)^{\frac{\text{power1}}{\text{power2}-\text{power1}}} (1 - \frac{\text{power1}}{\text{power2}}).$$

For the default values power1 = 3 and power2 = 4, we get $t_0 = 0.75$ and $f(t_0) \approx 0.105$.

Collision phase tier The collision phase parameter models the last part of the flow function with an exponential decay function instead of a polynomial one. A value of 0.04 for example means that the amplitude will decay by a factor of e ($\approx$ 2.7183) every 4 percent of a period.[4] We define the moment in time where polynomial decay changes to exponential decay, i.e. the junction point, as the instance of glottal closure. This instance has to be determined from the glottal flow function by two continuity constraints. The first constraint implies that at the junction the polynomial flow function and the exponential flow function must have equal amplitudes, i.e. there is no sudden jump at that point. The second constraint implies that at the junction the derivatives of both functions must also be equal. In figure 12.7 we show the effect of a collision phase of 0.04 on the flow function in the upper part and on the derivative of the flow function in the lower part of the figure. The glottal closure points, where polynomial decay changes to exponential decay is shown by dotted lines. It is clear from this figure that at the glottal closure points the two constraints defined above are met. In figure 12.5 where the flow function had no collision phase, the derivative instantly jumps from the minimum value to zero at the glottal closure point. In figure 12.7 the derivative doesn't jump instantly back to zero value but gradually approaches zero.

Spectral tilt tier Spectral tilt represents the extra number of dB the voicing spectrum should be down at 3000 hertz. According to Klatt and Klatt [1990] this parameter is necessary to model "corner rounding", i.e. when glottal closure is non simultaneous along the length of the vocal folds. If no points are defined in this tier, spectral tilt defaults to 0 dB and no spectral modifications are made. Spectral tilt makes it possible to attenuate the higher frequencies. The attenuation is implemented by a first order recursive filter

$$y_n = ax + by_{n-1}.$$

Its frequency response is given by

$$|H(f)| = a/\sqrt{1 - 2b\cos(2\pi fT) + b^2}, \tag{12.2}$$

[4]The exponential flow function is $f(t) = Ae^{-(1/\text{collisionPhase})\cdot t/T}$, where A is the amplitude at the glottal closure point and T is the period. For a collisionPhase of 0.04 this function is $f(t) = Ae^{-25t/T}$.

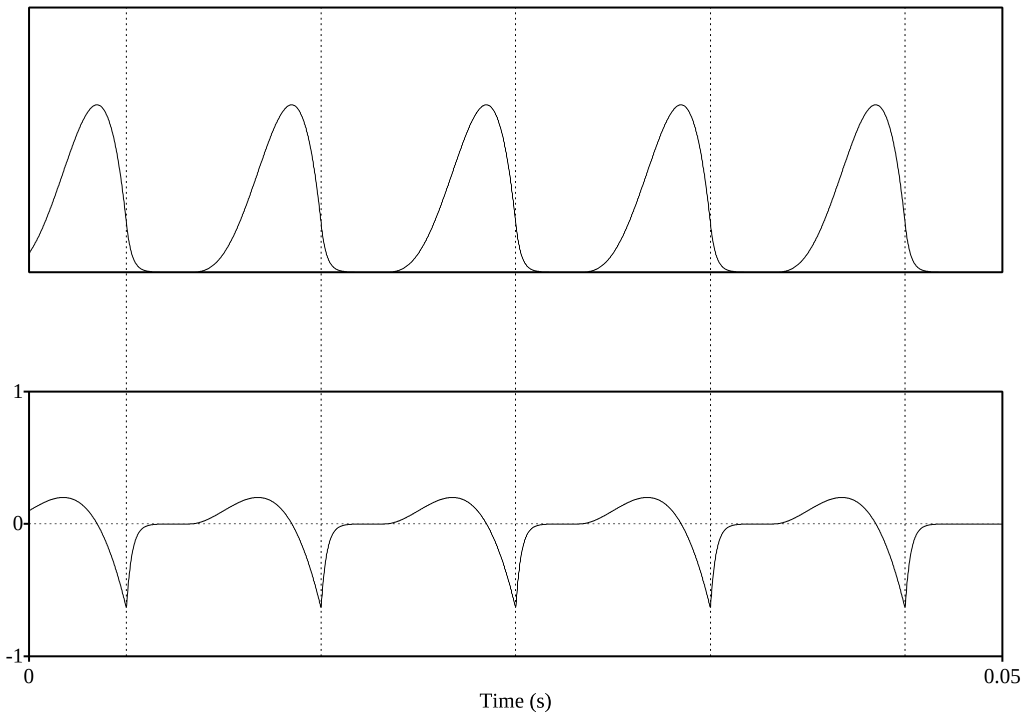


Figure 12.7.: Effect of collision phase on the glottal flow signal (top) and its derivative (bottom). The vertical dotted lines mark the glottal closure points where the polynomial flow function changes into an exponential decaying one.

where T is the sampling time. By choosing $a = 1 - b$ we have $H(0) = 1$. The response of this filter then only depends on b, different values for b give different frequency responses. For a prescribed spectral tilt, we can obtain the value for b by requiring that at $F = 3000$ Hz the following equivalence holds

$$20 \log |H(F)| = -\text{spectralTilt}.$$

In the left part of figure 12.8 we show the decibel values of the frequency response from 0 Hz to 5000 Hz of equation (12.2). The values of spectralTilt were varied from 0 to 50 dB in steps of 5 dB. The reference at 3000 Hz is show by a vertical dotted line.

The right display of figure 12.8 shows the result of spectral tilt measurements on the glottal source sounds. The horizontal axis shows the prescribed spectral tilt and the vertical axis the measured spectral tilt. The skeleton script shows how the point for $spectralTilt = 10$ was obtained. We start by defining a standard KlattGrid and add a pitch point of 100 Hz and a voicing amplitude point. We start a loop to increase the value of spectralTilt by 5 in each iteration. For the first iteration spectralTilt = 0. We select the KlattGrid and add the spectral

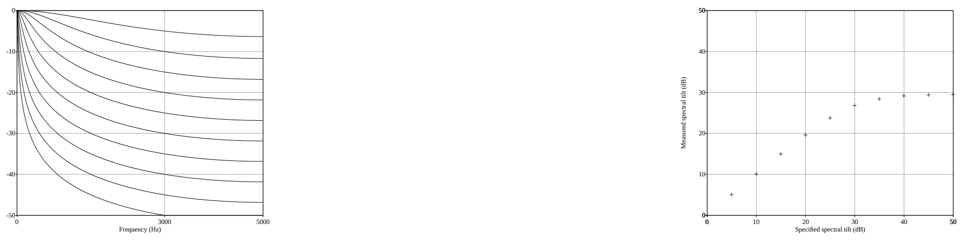


Figure 12.8.: On the left the theoretical filter curves for spectral tilt values that vary in steps of 5 dB from 0 dB to 50 dB. On the right a plot of the actual spectral tilt as measured from synthesized sounds versus the specified spectral tilt.

Script 12.3 A skeleton to measure the actual spectral tilt values of figure 12.8.

```
kg = Create KlattGrid: "kg", 0, 1, 6, 1, 1, 6, 0, 0, 0
Add pitch point: 0.5, 100
Add voicing amplitude point: 0.5, 90
for i to 11
    spectralTilt = (i - 1) * 5
    selectObject: kg
    Add spectral tilt point: 0.5, spectralTilt
    To Sound
    To Spectrum: "no"
    dif[i] = Get band density difference: 98, 102, 2998, 3002
    dif = 0
    if i > 1
        dif = dif[1] - dif[i]
    endif
    #...
    Remove spectral tilt points: 0, 1
    #...
endfor
```

tilt value to its tier. Next the glottal source sound will be synthesized and its spectrum will be calculated. In the next step we calculate the difference in band spectral density between the frequency bands around 100 Hz and the frequency band around 3000 Hz. These values were chosen because exactly one harmonic of the pitch is in each band. We subtract this value from the reference value for spectralTilt = 0 to obtain the values that were displayed in the figure. Then we clear the spectral tilt tier by removing all the points and before we start the next iteration, we can do some bookkeeping, like saving the density differences numbers for later graphical display and deleting the sound and the spectrum.

We note that the maximum spectral tilt that can actually be realised with this filter is approximately 30 dB. The filter honours spectral tilt values up to 20 dB and then flattens off. More than 30 dB of spectral tilt can not be accomplished.

Aspiration amplitude tier The aspiration amplitude tier models the (maximum) amplitude of noise generated at the glottis. The noise amplitude is given in dB SPL. The aspiration noise is independent of glottal timings and is generated from random uniform noise which is filtered by a very soft low-pass filter.[5]

Breathiness amplitude tier The breathiness amplitude tier models the maximum noise amplitude during the open phase of the glottis (in dB SPL). The amplitude of the breathiness noise is modulated by the glottal flow as is shown in figure 12.9 where we show the glottal

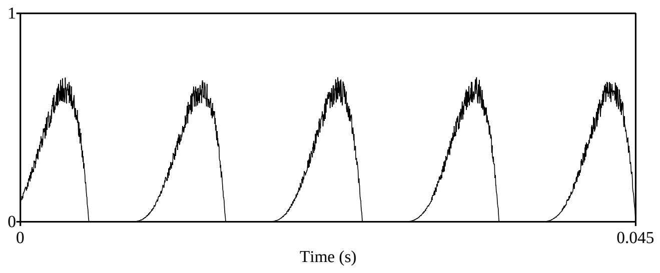


Figure 12.9.: The glottal flow for a 100 Hz pitch with 70 dB breathiness. The voicing amplitude was 90 dB SPL.

flow for a KlattGrid with a 100 Hz pitch point, a voicing amplitude point at 90 dB SPL and breathiness amplitude point at 70 dB SPL. The breathiness amplitude nicely follows the flow amplitude.

Double pulsing tier The double pulsing tier models diplophonia (by a fraction between zero and one). Whenever this parameter is greater than zero, alternate pulses are modified. For the time being, a pulse is modified with this *single* parameter tier in *two* ways: it is *delayed in time* and its amplitude is *attenuated*. If the double pulsing value is maximum (= 1), the time of closure of the first peak coincides with the opening time of the second one. Figure 12.10 shows an example of the flow signal for double pulsing. The KlattGrid is defined by script 12.4. The pitch is set to 100 Hz with an open phase of 0.5. The double pulsing tier has two values: at time 0.05 s it is set to 0, at time 0.15 it is set to the maximum value 1. The figure shows that the flow signal before time 0.05 s is a normal 100 Hz flow signal, showing 5 peaks in the 0.05 s time interval. From time 0.05 s on we see a gradual decrease of the first pulse of every pair and a displacement towards the second pulse of the pair until the first one totally disappears.

12.3. The vocal tract part

After a sound is generated in the phonation part of a KlattGrid it may be modified by the filters in the vocal tract part. These filters are formant filters and antiformant filters. A formant

[5]The low pass filter is $y_n = x_n + 0.75y_{n-1}$. From equation (12.2) we calculate that the ratio of the filter outputs at zero frequency and the Nyquist frequency is $20\log(H(0)/H(f_{Nyquist})) = 20\log((1+0.75)/(1-0.75)) \approx 12$ dB.

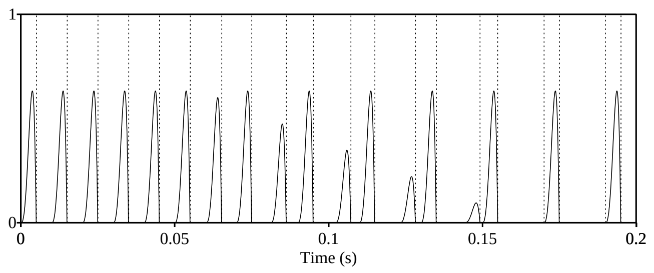


Figure 12.10.: Double pulsing for a 100 Hz pitch which increases linearly from 0 to 1 between times 0.05 s and 0.15 s. The vertical dotted lines are the moments of glottal closure.

Script 12.4 The script that generates the flow signal of figure 12.10.

```
Add pitch point: 0, 100
Add voicing amplitude point: 0, 90
Add open phase point: 0, 0.5
Add double pulsing point: 0.05, 0
Add double pulsing point: 0.15, 1
```

filter boosts frequencies and an antiformant filter attenuates frequencies in a certain frequency region. If these filters are applied one after the other, this type of filtering is called *cascade* or *series* filtering. A cascade type filter set is shown in the vocal tract part of figure 12.1. In the figure the sound signal flow goes from left to right. The sound source signal is generated in the upper left part of the figure where the three boxes symbolize the specifications of the phonation. The sound then enters the box labeled "TF F1 B1", this happens to be a tracheal formant filter. If some tracheal formant frequency and bandwidth has been defined, the source signal will be filtered in this box, otherwise it will leave this box unchanged. The signal then enters the next box labeled "TAF F1 B1" which happens to be a tracheal antiformant. It will leave this boxed filtered if some frequency and bandwidth values were defined, or unmodified. After it leaves this box the next one follows and this goes on and on until the last box.

In contrast to cascade filtering there is *parallel* filtering. In parallel filtering the input sound is applied to all the parallel filters at the same time, all filters process the same signal and then the output of all these parallel filters is summed so that only one output signal results. The frication section, for example, is always modeled with parallel filters as can be seen in figure 12.1. The vocal tract filters can also work in parallel, as is shown by figure 12.11. In fact, vocal tract filtering can be done in either of two ways: cascade or parallel. When sound is synthesized a choice has to be made for one of the two. The default is cascade filtering. Klatt [1980] mentions some reasons in favor of the cascade type of vocal tract filtering:

- The relative amplitudes for vowels are synthesized correctly. There is no need for individual formant amplitude regulation.

- It is a better model of the vocal tract transfer function during the production of non-nasal

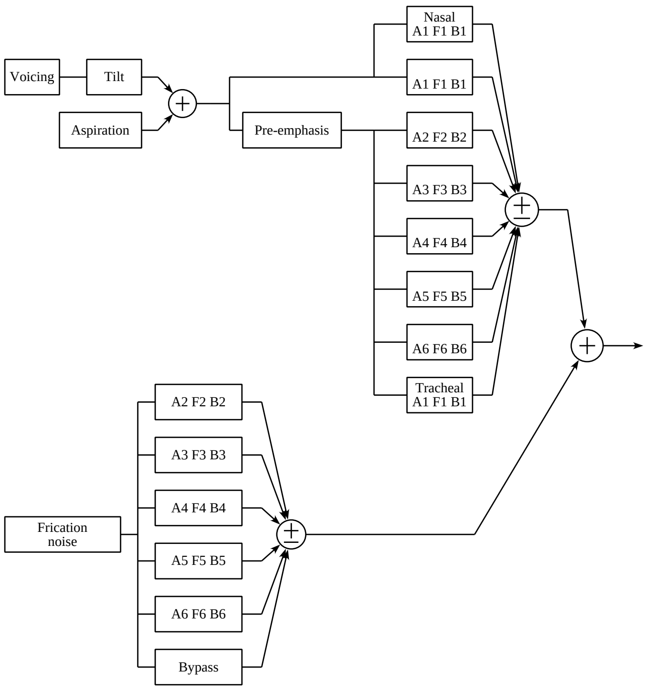


Figure 12.11.: The parallel version of the vocal tract filters.

sonorants.

However for sounds that do not follow the amplitude relations of vowels, for example fricatives and plosives, we have to use parallel filtering.

For the cascade version of the vocal tract filter, each formant filter is modeled by two tiers, a frequency tier and a bandwidth tier. For the parallel version each formant filter has to be modeled by three tiers: besides the frequency and bandwidth tiers we need an additional amplitude tier. The tiers that model each formant are completely decoupled. This means that for a formant its frequency as a function of time, its bandwidth as a function of time and its amplitude as a function of time can be set completely independent of each other.[6] In the cascade filtering of the vocal tract part the amplitude relations between formants resemble automatically the relations found in natural speech. For parallel filtering we have to explicitly define each formants' amplitude which may be difficult to determine.

[6]For example, in the KlattGrid you could specify only one point in the bandwidth tier, i.e. effectively keeping the bandwidth of the formant constant in time. In the formant frequency tier you can vary the formant frequency by defining multiple points. For frame-based synthesizers you would have to specify frequency and bandwidth values in synchrony for each frame. Besides, the only way to simulate in a frame-based way that formants and bandwidths change "independently" is by increasing the number of frames per second.

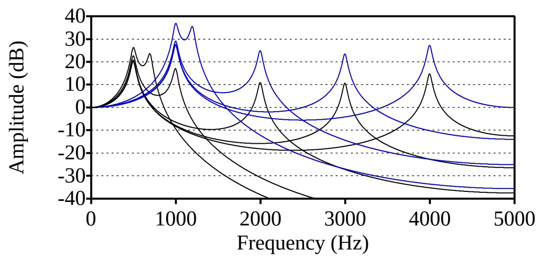


Figure 12.12.: The effect of two formants in cascade.

In section 11.4 we have shown the effect of frequency and bandwidth on the amplitude of *one* formant. We will now show the effect on the amplitude spectrum of more than one formant filter if they are applied, one after the other, in cascade. In figure 12.12 we show what happens to the spectrum of two formants if these formants approach each other. We first show what happens for a first formant at a constant frequency of 500 Hz and a second formant whose frequency changes in steps from 3000 Hz to 2000 Hz and then to 1000 Hz and finally to 700 Hz. These four spectra are in the lower part of the figure. First of all we see that the peak of the second formant is always lower than the peak of the first formant, the farther the second is from the first the larger the difference in amplitude. The amplitude of the second formant falls off at approximately −6 dB/oct. It is only when the two formants are near to each other that the amplitude of the first formant increases. This effect is already noticeable when the second formant is at 1000 Hz and the formants are at a distance of 500 Hz. When the formants are at 200 Hz distance the first formant has raised an extra 6 dB. We see the same behaviour when the first formant starts at another frequency like for the upper part of the figure where it is kept constant at 1000 Hz. Now the second formant changes from 3000 Hz to 2000 Hz and then to 1200 Hz. The doubling of the first formant frequency results in a 6 dB increase in its amplitude. The amplitude of the second formant increases by approximately 12 dB. This is also in line with the observation in Klatt [1980] who states that if one of the formants in a cascade model is changed all higher formant amplitudes change by a factor proportional to frequency squared.

12.4. The coupling between phonation and vocal tract

The coupling part of the KlattGrid models most of the interactions between phonation and vocal tract. On the one hand we have the tracheal formants and antiformants which are part of this coupling on the other hand we have delta formants and delta bandwidths that model the change of oral formant frequencies and bandwidths during the open phase of the glottis. The coupling part is only partly shown in figures 12.1 and 12.11, only the tracheal formants and antiformants figure as if they form part of the vocal tract part. We show it this way because tracheal formants and antiformants are implemented as if they filter the phonation source signal.[7]

Besides the tracheal system with its formants and antiformants we can also model the change of oral formant frequencies and bandwidths during the open phase of the glottis. With the delta formant grid we can specify the amount of change of an oral formant and/or a bandwidth during the open phase of the glottis. The values of the delta tier will be added to the values of the corresponding oral formant tier but *only during the open phase of the glottis*. In figure 12.13 we show two examples of this type of coupling where extreme values of the

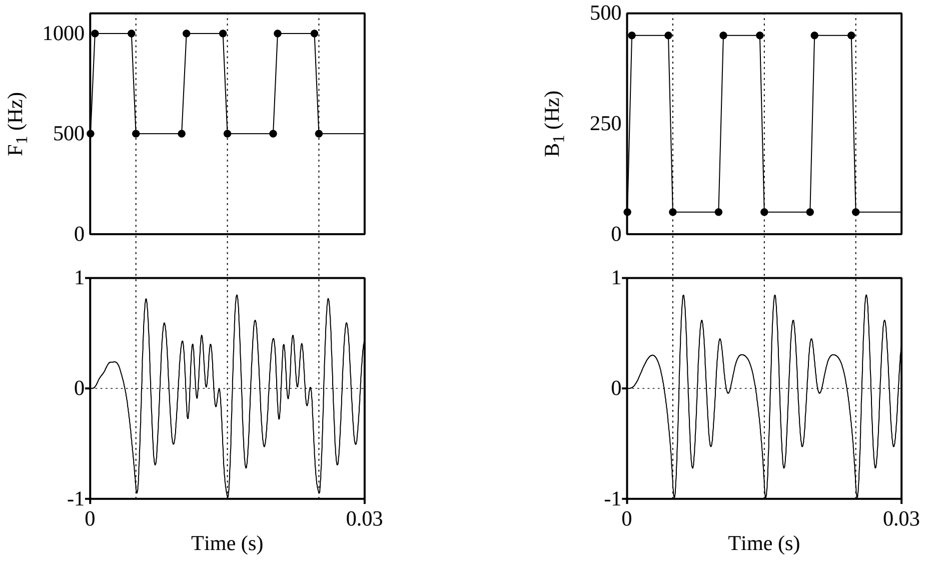


Figure 12.13.: Example of coupling between vocal tract and the glottis with some extreme coupling values. For all signals pitch was held constant at 100 Hz, the open phase of the glottis was set to 0.5, and only one oral formant and bandwidth are defined. Top left: oral formant bandwidth held constant at 50 Hz and the oral formant's frequency is 500 Hz during the closed phase and 1000 Hz during the open phase. Top right: the first oral formant frequency was constant at 500 Hz but oral bandwidth varies from 50 Hz during closed phase to 450 during open phase. At the bottom the resulting sounds. The glottal closure times are indicated with dotted lines.

coupling parameters have been taken for a clear visual effect. In both examples we generate a voiced sound with a 100 Hz pitch, an open phase of 0.5 to make the duration of the open and closed phase equal, and one oral formant. In the figure on the left the oral formant bandwidth is held constant at 50 Hz and the frequency is modified during the open phase. The oral formant frequency is set to 500 Hz. By setting a delta formant point to a value of 500 Hz we

[7]Otherwise we had to add the lungs as another source of sound besides the glottis, which would complicate things needlessly.

accomplish that during the start of the open phase of the glottis the oral formant frequency will increase by 500 Hz to 1000 Hz. At the end of the open phase it will then decrease to the 500 Hz value of the oral formant tier. To avoid instantaneous changes we let the oral formant frequency increase and decrease with this delta value in a short interval one tenth of the duration of the open phase duration. The top display shows the first oral formant frequency as a function of time during the first 0.03 s. This is exactly the duration of three pitch periods as can be easily checked because the moments of glottal closure are indicated by dotted lines. The bottom display shows the corresponding sound signal. The 100 Hz periodicity is easily visible as well as the oral formant frequency doubling in the second part of each period: we count almost two and a half periods of this formant in the first half of a period, the closed phase, and approximately four and a half periods during second half of a period, the open phase. In the figure on the right the oral formant frequency is held constant at 500 Hz and now the oral formant's bandwidth is changed during the open phase. The oral formant's bandwidth point is set to 50 Hz and the delta bandwidth point to 400 Hz. This has the effect on the oral formant's bandwidth as shown in the top right figure: a 50 Hz bandwidth during the closed phase versus a 450 Hz bandwidth during the open phase. As before, bandwidths don't jump but change within a short time interval to their extreme values. In the bottom right figure, which is the sound output, this manifest itself clearly: the oral formant is a damped sinusoid in the first part of a period and a very heavily damped signal in the second part of a period.

Script 12.5 The minimal KlattGrid to generate the data for figure 12.13.

```
kg = Create KlattGrid: "kg", 0, 1, 6, 1, 1, 6, 1, 1, 1
Add pitch point: 0.1, 100
Add voicing amplitude point: 0.1, 90
Add open phase point: 0.1, 0.5
Add oral formant frequency point: 1, 0.1, 500
Add oral formant bandwidth point: 1, 0.1, 50
Add delta formant frequency point: 1, 0.5, 500
Extract oral formant grid (open phases): 0.1
selectObject: kg
To Sound
```

12.5. The frication part

The frication part of a KlattGrid can generate, just like its name suggests, all kinds of frication noises. The frication sound is added to the output of the vocal tract part. The frication part is an independent section in the synthesizer which gives the opportunity to add a frication type of noise completely independent of the phonation and the vocal tract part characteristics.

A layout of the frication part is shown at the bottom of figures 12.1 and 12.11. It consists of a source of random uniform noise that can be modified by a set of parallel formant filters and a bypass filter. The amplitude of the noise source is regulated by the frication amplitude tier. The set of formant filters shapes the noise spectrum to any form desired.

12.6. Differences between KlattGrid and the Klatt synthesizer

- Klatt's synthesizer is frame-based. In the Klatt synthesizer, parameters are constant during each frame, they can only vary from one frame to the other. Typical frame durations are in the range from 5 to 10 ms. To realize bursts within such a time frame, special considerations have to be taken. In a KlattGrid you specify parameters as a function of time with any precision you like.
- In the Klatt synthesizer, only six normal formants, one nasal formant-antiformant pair and one tracheal formant-antiformant pair can be defined. The number of formants and antiformants in a KlattGrid is not limited, you can define any number of formants and antiformants.
- In Klatt's synthesizer there is only one set of formant frequencies that has to be used for the vocal tract part as well as for the frication filter. In a KlattGrid the formant frequencies in the frication part and the vocal tract part have been completely decoupled.
- In the Klatt synthesizer the glottal flow function has to be chosen before-hand. A KlattGrid allows varying the form of the glottal flow function as a function of time.
- In the Klatt synthesizer only the frequency and bandwidth of the first formant can be modified during the open phase. In a KlattGrid there is no limit on the number of formants and bandwidths that can be modified during the open phase of the glottis.
- In Klatt's synthesizer many parameters have been quantized before-hand. In a KlattGrid there is no before-hand quantization of parameters. Quantization only takes place when a sound has to be written to a file and amplitudes have to be quantized to, for example, 16 bits integer precision.
- The KlattGrid is fully integrated into Praat. This is a big advantage because all visualizations and analysis methods are directly available for the synthesized sounds. At the same time all the editors and objects in Praat could be used in the implementation of the KlattGrid.

13. Formant frequency analysis

13.1. Introduction

Formant frequencies are very useful in linguistic research. First of all, formant frequencies form a very low dimensional representation for vowels, i.e. most vowels can be very reliably characterized by two formants only.[1] Formant frequencies also correspond reasonably well to articulatory gestures as the first formant frequency distinguishes vowels mainly in the vowel height dimension and the second formant frequency distinguishes vowels mainly in the front-back dimension.

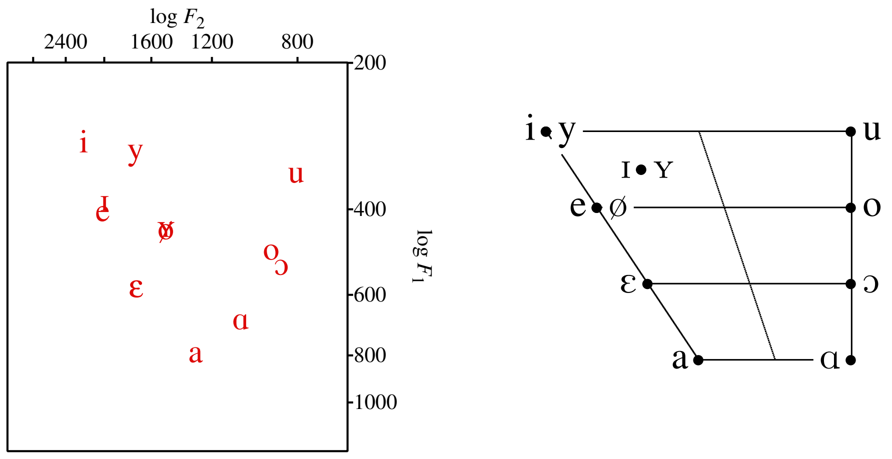


Figure 13.1.: Left pane: average formant frequencies for twelve Dutch vowels from fifty male speakers as measured by Pols et al. [1973]. Right pane: the Dutch vowel system.

As an illustration we have plotted, in the left pane of figure 13.1, the first two formant frequencies of the twelve Dutch monophthong vowels, averaged over fifty male speakers, as measured by Pols et al. [1973]. The figure agrees quite nicely with the phonological Dutch vowel system which is plotted in the right pane of the same figure.

Another advantage of formant frequencies shows itself in the coding domain: a representation of vowels with formant frequencies (and bandwidths) is extremely robust against noisy

[1]If we restrict ourselves to vowel synthesis, Praat's vowel editor, described in chapter 10, might convince you that this is indeed the case.

transmission channels, i.e. a small distortion of a formant frequency (or a bandwidth) hardly influences vowel quality at all.

13.2. What is a formant?

We can define formants from three different viewpoints.

1. From a *production* oriented view one defines a formant as a resonance of the vocal tract. This resonance is the result of interference between sound waves traveling in opposite directions in the vocal tract. If the frequency components are in symphony they can boost each other, if not, they cancel each other. Data and model studies show that for male vocal tracts on the average there is one resonance in every successive 1000 Hz frequency interval and for females approximately one resonance in every 1100 Hz interval. Because the strengths of the frequency components in a voiced source signal fall off with approximately -12 dB/oct, the frequency content above 5000 Hz can be neglected for male vocal tracts. Therefore, approximately five formants are present within the first 5000 Hz of a male voice. For the female voice, because of their shorter vocal tract lengths, approximately five formants on average are within the first 5500 Hz frequency interval. You can find this information in any book on the acoustics of speech, see Stevens [2000] or Johnson [1997] or Harrington and Cassidy [1999].

2. From a *perception* oriented view formants are salient parts in the spectrum that can serve to distinguish, for example, different vowel phonemes from each other.

3. In *acoustics* we define formants as peaks in the spectrum envelope of a speech sound.

Under normal circumstances these definitions are more or less consistent with each other and we do not distinguish between them: a resonance of the vocal tract at a frequency f_r results in a peak in the spectrum at approximately this same frequency value[2] Formants are most clearly defined for stressed vowel sounds because these tend to be the *loudest*, the *longest* in duration and *best articulated* parts of speech over time. In the literature formant frequencies are almost always related to vowels only. In an idealized world the first two or three formants identify a vowel. With Praat's vowel editor which is discussed in chapter 10 and that you can activate via New>Sound >Create Sound from VowelEditor, you can check that with only two formants you can create reasonably sounding vowels and vowel-like glides. The first formant roughly corresponds to vowel height, the second formant to vowel place as can be deduced from the figure above.

However, from a signal analysis point of view formants are problematic because we do not know how to determine them reliably from the speech sound signal in a *fully automated fashion* and therefore automatic formant frequency analysis is not always possible. In one way or another, operator intervention seems to be necessary, even for the best read-aloud speech. For example, Pols et al. [1973] state that in a number of cases prior knowledge of where the formant should be located played a significant role in their decision on formant frequency

[2]However, if the fundamental frequency is higher than a resonance frequency of the vocal tract there will be no formant peak in the spectrum.

measurements from fifty male speakers. When the fundamental frequency increases, the problems aggregate. In a recent study on the formant frequencies of Dutch vowels from several speakers Adank et al. [2004] had to manually alter 20-25% of their automatically determined formant contours.

We just do not have enough information to automatically extract formants from the speech signal. This means that in order to measure formants we have to add a knowledge source in the measuring process. Most of the time this knowledge source is *you*, the experimenter, knowing the identity of the vowel and adjusting the measurement values until they are reasonably in accordance with the values you expect them to be for *this particular vowel*.

13.3. How are formants measured?

Before modern spectrum analysis techniques became available, the only way to measure formant frequency values was displaying the time signal and trying to deduce formant frequencies from the form of the sound signal during a voicing period. The sound during one voicing period is modeled as being the sum of a number of damped sinusoids, each sinusoid being a formant (see A.6). Nowadays we have better signal processing facilities and we can measure formants in a number of ways.

- From the oscillogram. This is like in the old days. This method is not very precise but it can still be used of course as a kind of visual check or an initial guess. In section 13.3.1 we show how to measure formant frequencies from the oscillogram.

- From the spectrogram. The invention of the spectrograph (Koenig et al. [1946]) during the 1940's, for the first time enabled researchers to get an impression of the dynamics of the sound signal in the frequency domain. It became possible to display the frequency content of a signal as a function of time by showing the strength of frequencies with gray values printed on paper. From these printed papers, formant frequencies could be measured by tracing the darker areas on the paper. Nowadays we can do this kind of analysis by a computer and compute the digital analog of this paper image, the spectrogram. In section 13.3.2 we show more details.

- From linear predictive coding analysis (LPC). The application of LPC analysis on the speech signal, starting at the mid 1960's, made it possible to fully automate formant frequency measurements. However, formant frequency measurements from LPC analysis does not always give you the formant frequency values you might "expect". Despite these deficiencies, LPC is the predominant algorithm nowadays for determining formant frequencies automatically. In section 13.3.4 we will go into more details about this method.

- From a bandfilter analysis. If the bandwidth of the filters is larger than the fundamental frequency and the number of filters is large enough we can obtain a fair representation of the smoothed spectrum. Section 13.3.3 will show more details.

13.3.1. Formant frequencies from the oscillogram

Historically seen, this is the oldest way to measure formant frequency values. To show how formant frequencies can be measured from the oscillogram we start with a very simple example, a one-formant vowel. In figure 13.2 we have displayed the oscillogram of three periods of an artificial vowel-like sound with only one formant in it that has a fundamental frequency F_0 of 100 Hz. We can clearly see the three periods of the fundamental frequency, each with a duration of 0.01 s. In each fundamental period we count five oscillations; these oscillations become smaller in amplitude towards the end of a period. This gradually dying oscillation is called a formant and its form as a function of time can be defined as $f(t) = e^{-\alpha t}\sin(2\pi Ft)$, where the parameter α is called the *damping* and the parameter F is called the *formant frequency*. Essentially this function $f(t)$ describes a pure tone whose amplitude fades away as time goes on. Because we count five oscillations in one period of the fundamental, each oscillation lasts 0.002 (= $0.01/5$) s and its frequency will therefore be 1/0.002 which equals 500 Hz. The formant function we used to generate this signal was $s(t) = e^{-\pi 50t}\sin(2\pi 500t)$ and has 500 Hz frequency and 50 Hz bandwidth. The formant frequency derived from the oscillograms agrees nicely with this value.

The formant's bandwidth for this vowel can also be deduced from the figure. This is done by relating the amplitudes of the curve at two different points in time. We start by writing down the amplitudes at times t_1 and t_2 from the formula above as

$$\begin{aligned} s(t_1) &= e^{-\pi Bt_1}\sin(2\pi Ft_1) \\ s(t_2) &= e^{-\pi Bt_2}\sin(2\pi Ft_2). \end{aligned}$$

If we divide the amplitudes we get

$$\frac{s(t_1)}{s(t_2)} = \frac{e^{-\pi Bt_1}\sin(2\pi Ft_1)}{e^{-\pi Bt_2}\sin(2\pi Ft_2)} = e^{-\pi B(t_1-t_2)}\frac{\sin(2\pi Ft_1)}{\sin(2\pi Ft_2)},$$

Next we strategically choose the points t_1 and t_2 to be an integral number of periods $T = 1/F$ apart, i.e. $t_2 = t_1 + kT$, where k is a natural number. We know that the values of the two sines in the quotient will always be equal at two time points that differ by an integral number of T, the quotient drops out of the equation. We are then left with

$$\frac{s(t_1)}{s(t_1+kT)} = e^{\pi BkT},$$

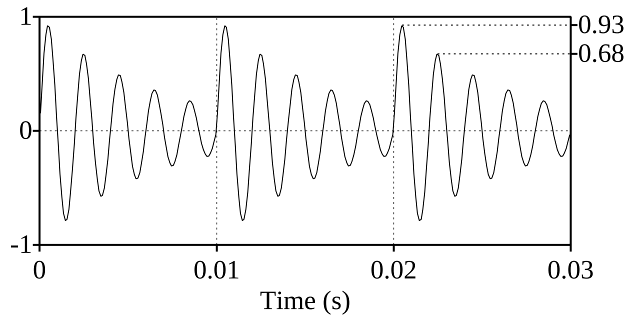
from which be can solve for B by taking the logarithm of both sides

$$\ln\left(\frac{s(t_1)}{s(t_1+kT)}\right) = \pi BkT.$$

The bandwidth B will then be equal to

$$B = \frac{1}{\pi kT}\ln\left(\frac{s(t_1)}{s(t_1+kT)}\right).$$

Therefore we can calculate the bandwidth by taking the logarithm of the quotient of two amplitudes, at times that lie for example one period T *of the formant* apart, and dividing this number by πT. Remember that we are now talking about the period of the formant and not the pitch period. In figure 13.2 we show an example how to calculate the bandwidth from the oscillogram. In the third fundamental period we have indicated two such amplitudes $s(t_1) \approx 0.93$ and $s(t_2) \approx 0.68$ that are about 0.002 seconds apart, i.e. $k = 1$. These amplitudes were obtained at local maxima because its there where they can be most easily measured. We obtain $B = \frac{1}{3.14 \cdot 1 \cdot 0.002} \ln(\frac{0.93}{0.68}) \approx 49.8$, which is very close to the 50 Hz bandwidth of the originating formant function. In general, the precision of the estimation of the bandwidth will increase if you take amplitudes further away than one period the formant frequency.



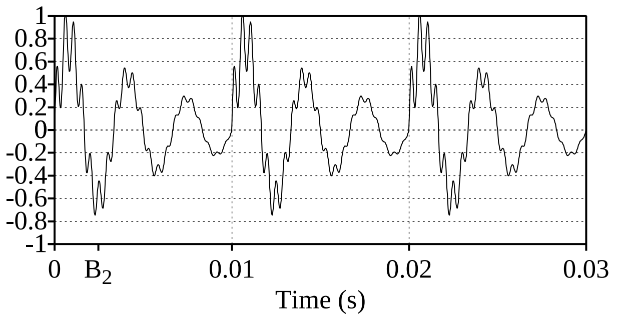


Figure 13.2.: Top: The oscillogram of two sound segment with one formant (top: $F = 500$ Hz with $B = 50$ Hz) and two formants (bottom: $F_1 = 300$ Hz with$B_1 = 50$ Hz and $F_2 = 2100$ Hz with $B_2 = 100$ Hz). For both sounds the F_0 equals 100 Hz.

With two formants the estimation of formant frequencies and bandwidths becomes more difficult. An example of an artificial two formant signal can be found in the bottom part of figure 13.2. The first formant with a frequency of 300 Hz is easily detectable. The second formant is somewhat smaller in amplitude than the first and superimposed on it. You can see this second formant as a high frequency ripple of the first formant's amplitude. Approximately seven periods of this formant fit into one period of the first one and we conclude from this that the second formant's frequency is near 2100 Hz. It is not too difficult to estimate the bandwidth of the first formant from this figure by the method indicated above. We guess $s(t_1) \approx 0.8$ at the maximum of the first period and $s(t_2) \approx 0.4$ for the second period and calculate $B \approx 66$ Hz. For the second formant the bandwidth calculation is a bit more involved because this formant is a ripple of the first and so the method above that compares the amplitudes at two well chosen points is useless. If instead of the amplitude difference between *two* points at $1/F$ distance we use the following relation between the amplitudes at *four* points $r = \frac{s(t_1)-s(t_1+T/2)}{s(t_2)-s(t_2+T/2)}$, where again $t_2 = t_1 + 1/F$, then we can calculate that $B = \frac{1}{\pi T} \ln r$. If t_1 is chosen at the top of a sine than $t_1 + T/2$ is one half period further and therefore at the valley. This means that r can be measured from the sound signal as the quotient of two differences between a top and the next valley of the damped sine. For our artificial signal in figure 13.2 the best position to measure this relation is probably in the first valley of the first formant. This position is indicated with a tick mark labeled B_2 at the bottom of the plot. The quotient of the depths of the two smaller valleys is approximately $7/5$, which gives a bandwidth of $B \approx 225$ Hz. This is more than twice the real value.

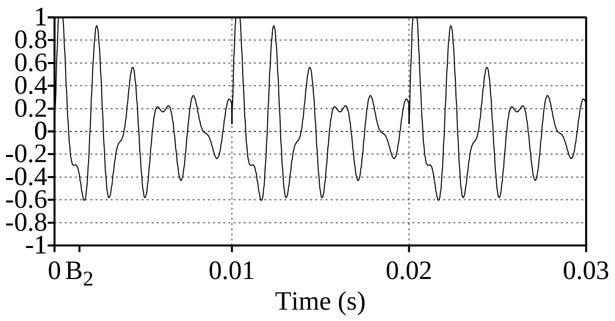


Figure 13.3.: Two formants, $F_1 = 530$ Hz, $B_1 = 50$ Hz, and $F_2 = 940$ Hz, $B_2 = 50$ Hz.

Even for these artificial signals it is tedious to get the formant frequencies right, let alone the bandwidths. In more difficult situations like in figure 13.3, i.e. when two formants are close together it becomes even harder to measure frequencies and bandwidths in this way. We have to use all kinds of knowledge/rules about how formants interfere with each other in the sound signal. This is why once the spectrogram was invented it was such a great advance in speech signal processing[3].

13.3.2. Formant frequencies from the spectrogram

From a spectrogram you can get a good impression about the formant structure of a sound segment. In figure 13.4 we show part of the Dutch speech fragment /...loːpt mɛt haːr.../ (Eng: ...walks with her...). The bottom part shows the interval tier with the segmentation of this sound. The top part shows the spectrogram with frequencies up to 5000 Hz represented.

[3]The spectrograph, a device to analyze a sound in its frequency components and to paint the spectrogram on paper, was invented during World War II to visualize speech in order to break enemy speech-scrambling methods.

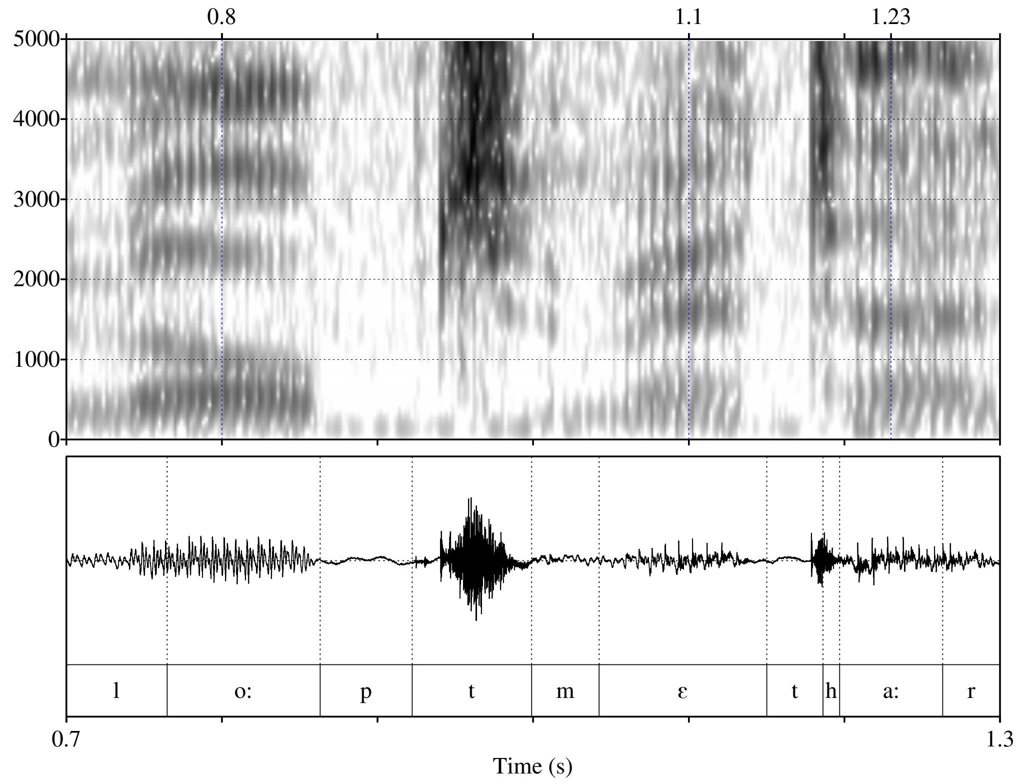


Figure 13.4.: A spectrogram of a short speech fragment.

Because we are now mainly interested in the voiced parts of the sound this upper range is sufficient. It seems the formant structure can be easily followed by eye, especially in the three vowel segments of /oː/, /ɛ/ and /aː/ where the darker parts are nicely separated from each other. We also note for the two /t/'s the relatively high burst of energy at frequencies above 2000 Hz that extends to the upper range of the displayed frequency range.

For the first vowel in the display, the /oː/, we see a number of formants in the spectrogram. The lower four can be easily identified by eye as these traces are well separated from each other, a first formant in the 500 Hz region, a second formant just above 1000 Hz, a third formant in the 2500 Hz region and a fourth raising from 3000 to 3500 Hz. The dark area between 4000 and 5000 Hz looks like a merger of a fifth and a sixth formant, the fifth formant starting at 4500 Hz in the /l/ region and raising to within the 4300 Hz range, the sixth staying constant at a frequency of approximately 4500 Hz. Although we are dealing with a monophthong /oː/ vowel and the formants can be traced by eye relatively easy, the formants are not static at all: they change during the time course of the vowel. The most stable part seems to be in the middle section of this vowel. If we had to represent the /oː/ vowel with only one set of formant frequency values then the formant frequencies from the mid part of this vowel would probably be the “best”. If we look at a spectral slices in the spectrogram, for example the slices centered at 0.8 s in the /oː/ and the one centred at 1.1 s we have figure 13.5.

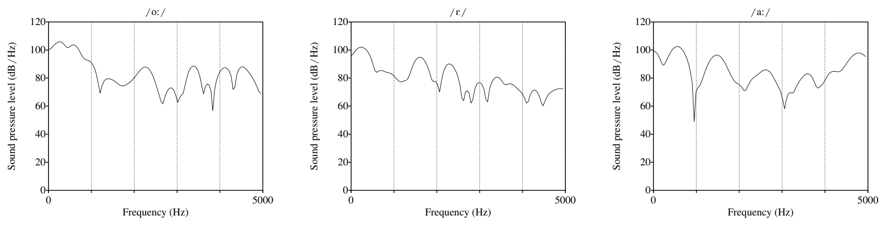


Figure 13.5.: Three spectral slices taken from the wideband spectrogram 13.4. The slices were taken at 0.8 s, 1.1 s and 1.23 s and belong to the vowels /oː/, /ɛ/ and /aː/, respectively.

For the vowel /ɛ/ there also seem to be five or six formants present. The first formant is well below 1000 Hz but according to the figure its is a very wide trace. The second formant starts approximately at 1200 Hz and increases to 1700 Hz. The third formant also increase but now from 2000 to 2500 Hz, approximately. After the third formant things start to get fuzzy: probably three other formants are visible but they are not very clearly traceable from the spectrogram. For the /aː/ vowel the first two formants are clearly visible but formants three and four are very blurred while formant five is more pronounced.

If we could only have a way to automatically measure these formant traces!

13.3.3. Formant frequencies from bandfilter analysis

13.3.4. Formant frequencies from linear prediction

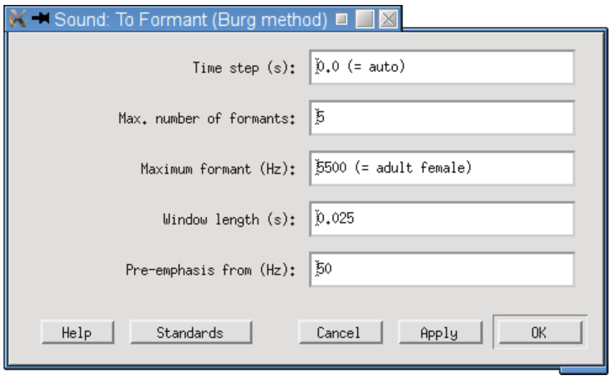
Formant frequency analysis by linear prediction coding (LPC) analysis differs from the previous analysis methods in a fundamental way. Linear prediction is a *parametric model* which means that the analysis is based on a model for speech production and the analysis therefore *fits model parameters*. Parametric models are a mixed blessing: they offer more accurate spectral estimates than nonparametric ones in the case where the data satisfy the model. However, when the data does not satisfy the model, the results may be very wrong. LPC analysis is based is on the source-filter theory of speech production Fant [1960]. Source-filter theory hypothesizes that an acoustics speech signal is the result of a *source signal* (the glottal source or noise generated at a constriction in the vocal tract) *filtered* with the resonances in the cavities of the vocal tract downstream from the glottis or the constriction. LPC analysis separates the source and the filter from the acoustic signal. In principle this separation of a sound into a source and a filter can be done in infinitely many ways. This can be most easily seen in the frequency domain where the (simplified) source-filter model translates to the multiplicative model:

$$O(f) = H(f)S(f). \tag{13.1}$$

This formula relates the spectrum $O(f)$ of the sound to the spectrum $S(f)$ of the source and the spectrum $H(f)$ of the filter. According to the source-filter model applied to speech production, the spectrum of the synthesized sound is obtained by multiplying the spectrum of the source

$S(f)$ by the spectrum of the filter $H(f)$. For sound synthesis where we can generate source and filter specifications separately, producing the synthesized sound is relatively easy then. However, from the sound analysis point of view, we only have a sound spectrum $O(f)$ that needs to be decomposed into two separate spectra $H(f)$ and $S(f)$ such that when multiplied together they result again in the spectrum $O(f)$. The filter part $H(f)$ then contains information about the resonant parts, i.e. the formants. An infinite amount of combinations of $H(f)$ and $S(f)$ can be found to fulfill equation13.1.[4]

The use of linear prediction analysis makes it possible to measure formant frequencies automatically *if* we know the number of formant peaks that are present in the sound signal. Every peak in the spectrum needs one formant to model it, where by peak we do *not* mean "every single harmonic". The spectrogram example already showed us that the number of formant peaks is not always certain. The command that is used to determine formant frequency values automatically from a selected speech sound is **To Formant (burg)...**. Let us try the standard formant analysis with default values for the parameters. The form will look as follows.


Sound: To Formant (Burg method)

Time step (s):	0.0 (= auto)
Max. number of formants:	5
Maximum formant (Hz):	5500 (= adult female)
Window length (s):	0.025
Pre-emphasis from (Hz):	50

Figure 13.6.: The Sound: To Formant (burg)... form.

Time step determines the time between the centres of two consecutive analysis frames. We discussed this parameter already in section 2.4, where the general analysis scheme was explained. If this value is 0.01 and the sound has a duration of 2 s then approximately 200 analysis frames will result. If you leave the value at 0.0 Praat will use a value that is one fourth of the analysis window length.

Maximum number of formants determines the number of formants that the algorithm tries to calculate from the analysis frame. For most analysis of human speech you will want to calculate five formants.

Maximum formant determines the ceiling of the frequency interval in which formants will be calculated. On the average we expect one formant in every 1000 Hz interval for

[4]An analog with real numbers is the following. Say you want to decompose a real number, say 1, as the product of two other real numbers a and b such that $1 = a \cdot b$. Then any number a satisfies the equation if b is chosen as $1/a$.

a male speaker. Therefore a value of 5000 Hz is adequate for a male speaker. The default value of 5500 Hz is for an adult female speaker. Women tend to have shorter vocal tract lengths than men and because the formant frequencies of a tract are inversely related to the vocal tract length, the formant frequencies of women are larger than the formant frequencies of men. As a rule of thumb we can say that for female speakers we approximately have one formant in every 1100 Hz .

Window length determines the effective duration of the analysis window.

Pre-emphasis from determines from which frequency on the spectrum will be boosted, i.e. it determines the cut-off frequency of a first order high-pass filter that boosts frequencies with +6 dB per octave. If this value is taken as 50 Hz then frequencies around 100 Hz will be amplified by +6 dB, frequencies at 200 Hz will be amplified +12 dB and so on. Because a vowel spectrum generally falls off at -6 dB per octave, the pre-emphasis filter more or less compensates for the fall-off. The pre-emphasis therefore boosts the higher formants and gives these formants a better chance of being found in the analysis. Without pre-emphasis these formant peaks would be to low to be detected.

To perform the analysis we select the sound and choose “Sound: To Formant (burg)... 0.0 5 5000 0.025 50” (the default time step, maximum frequency is 5000 Hz for a male voice, analysis window is 0.025 s and pre-emphasis from 50 Hz). Now what happens behind your back is the following:

- If the sampling frequency of the sound is higher than twice the value of the *maximum formant* parameter, the sound will be down-sampled to a sampling frequency of twice the value of the *maximum formant*. Therefore, the new sampling frequency will become 2·*maximum_formant*. For example, if your maximum formant is 5500 Hz and the sampling frequency of the sound is 44100 Hz, the sound will be down-sampled to a sampling frequency of 11000 Hz.

- According to the general analysis scheme as shown in figure 2.9 a linear prediction analysis with the burg algorithm is performed on each analysis frame. The number of linear prediction parameters that will be fitted equal twice the *maximum number of formants* parameter. For example if we wanted to find five formants, the number of parameters to be calculated in a frame equals ten.

- The linear prediction coefficients that resulted from the analysis are transformed to frequencies and bandwidth pairs. Each pair of frequency and bandwidth correspond to one formant. Frequency-bandwidth pairs whose frequencies are either smaller than 50 Hz or that are within 50 Hz of the 2·*maximum_formant* value are discarded.

- The frequency-bandwidth pairs are sorted from low to high frequency values and then stored in the Formant analysis object.

In figure 13.7 we show the formants with red dots layed over the spectrogram. At first sight the red dots are in the dark areas as they should be. This is very promising. However, closer inspection reveals a number of misfits. For example, for the /oː/ the second formant is partly missing, this formant is only present at the start of the vowel and then suddenly disappears. Also for the /l/, the second formant is in the middle of the two dark areas like the third. At the start of the /ɛ/ the first formant makes a discontinuous jump and the second formant only starts to follow the dark area after a while but makes a *downward* glide in frequency at the start while the spectrogram suggests an *upward* glide. There seems to be some mismatch between the formant contours and the spectrogram. This is understandable because at some parts in the spectrogram we see six formant peaks and if the model has only five some compromises have to be made. This may result in one or more formants fitted in-between two peaks in the spectrum.

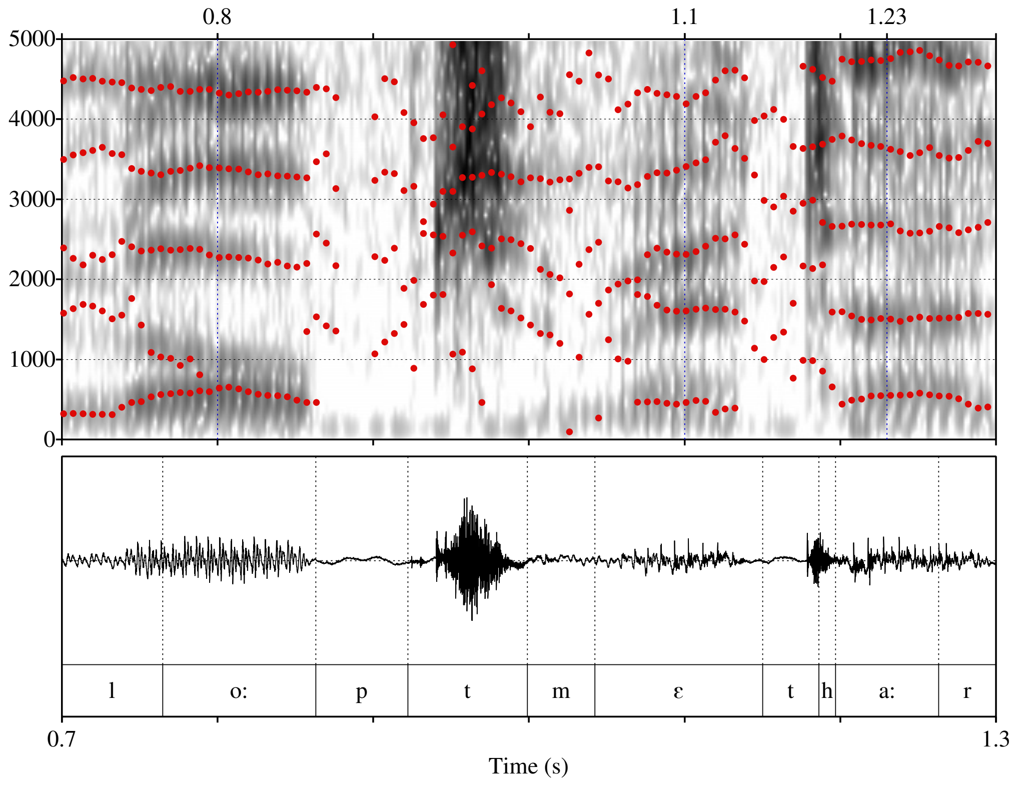


Figure 13.7.: Formant frequency analysis with Praat's default five formants.

Let us try a new analysis where we allow more formants than the default five to fit the signal. We will try to fit one formant more and use the command "Sound: To Formant (burg)... 0.0 6 5000 0.025 50". In figure 13.8 we have displayed the result of the new analysis. The complaints we had have now vanished and, at least for the vowel parts, the formants nicely trace the darker areas in the spectrogram. Only in vowel segments where the dark areas in the spectrogram are not nicely contiguous the formants seem to follow that trend and show irregularities. For example the third formant of the /aː / show irregularities near the transition to the /r/. In the same vowel some frames show five formant and other show six. But all in all, the analysis with six formants instead of five seems like a real improvement over the analysis with five.

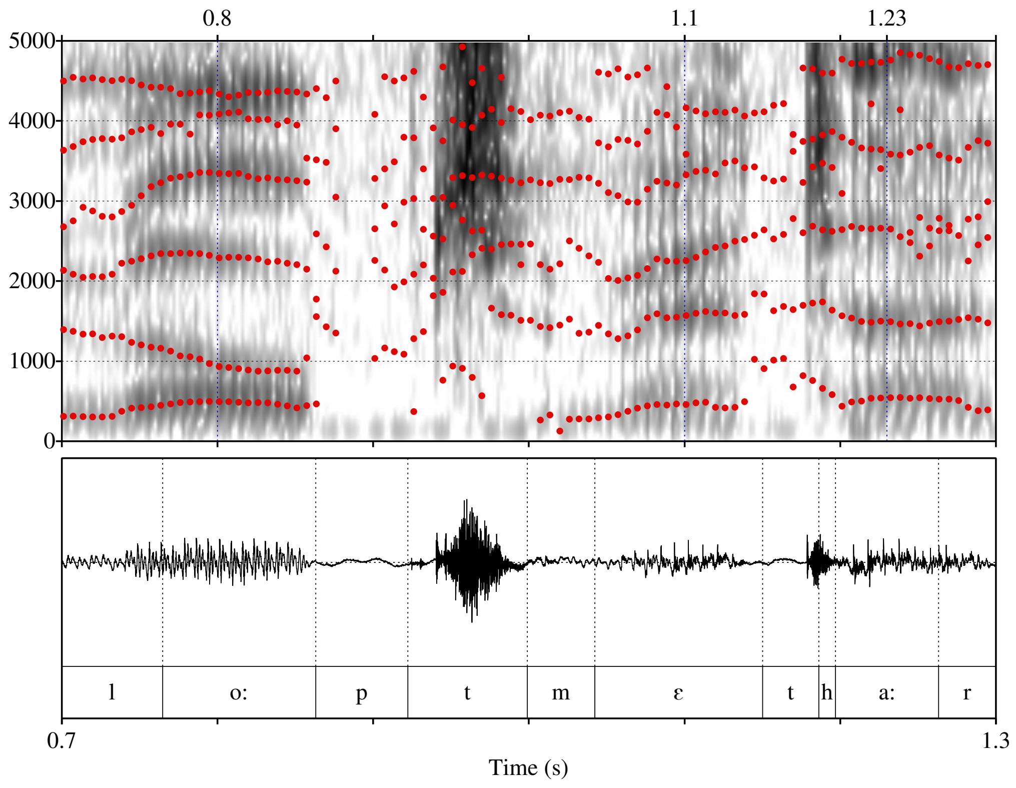


Figure 13.8.: Formant frequency analysis with six formants on the sound from figure 13.7.

However, increasing the number of formants in the analysis even more to seven results in a degradation of the formant contours. We do not show the figure here, but in-between the smooth contours of the analysis with six formants, spurious formant points turn up, disrupting the continuity that might exist.

Instead of looking for six formants in the frequency range from 0 to 5000 Hz we could also search for five formants but now in a more restricted frequency range. This can be arranged by setting the **Maximum formant** parameter to a lower value.

This is the case in figure 13.9, where the ceiling was set to 4500 Hz. This seems to result in formant tracks that at least for the lower frequency range seem to fit at least as good as was the case in figure 13.8, where 6 formants were measured. Obviously this measurement is also better than the measurement where we had the ceiling set at 5000 Hz as was the case in figure 13.7. In figure 13.10 we give a summary of the three discussed formant analysis by LPC. From left to right the three panes present spectral slices of the vowels /oː/, /ɛ/ and /aː/, respectively.

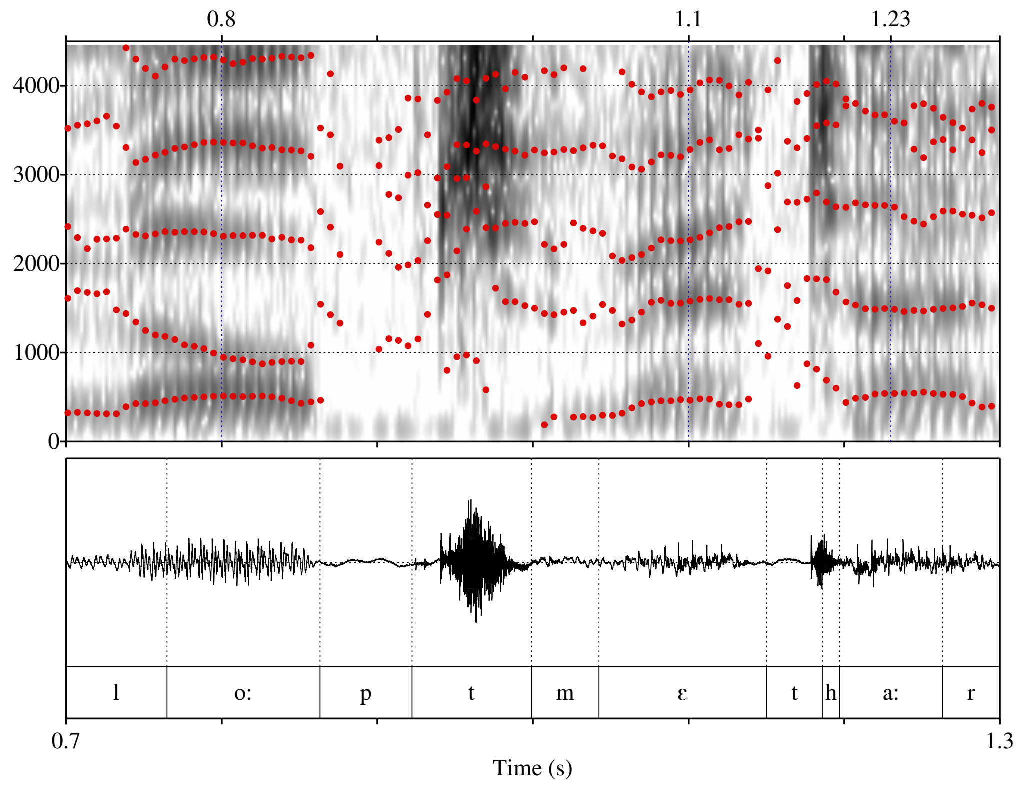


Figure 13.9.: Formant frequency analysis on the same sound as figure 13.7 with five formants and **Maximum formant** parameter set to 4500 Hz.

These spectral slices were determined from the formant frequency analysis by transforming the Formant object to an LPC object and then drawing a spectral splice at the three different time values 0.8, 1.1 and 1.23 s that are reasonably representative of these vowels. [5].

13.3.4.1. What are the correct formant frequencies?

The previous section showed that for real speech we can never be very sure where the formants are located. The formant frequency measurements showed a lot of variation with the *ceiling* parameter and with the number of formants we want to determine. For real speech we never know if we have the right formants because we never know how many there are. We have

[5]For example, in the left-most pane the spectrum drawn with a solid line is produced as:

```
Read from file...  ../Sounds/de-vrouw-loopt.wav
To Formant (burg)...  0 5 4500 0.025 50
To LPC...  10000
To Spectrum (slice)...  0.8 20 0 50
Draw...  0 5000 0 80 y
```

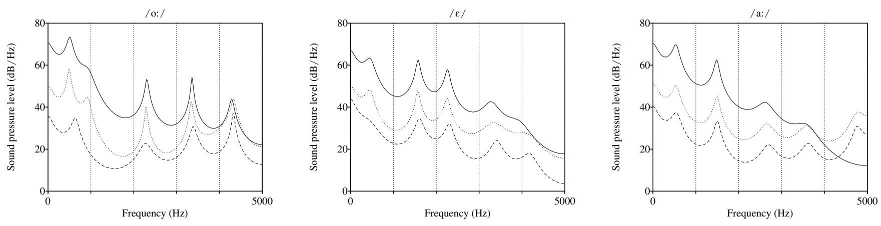


Figure 13.10.: LPC spectra of the vowels /oː/, /ɛ/ and /aː/, respectively.

Ceiling	F_1	B_1	F_2	B_2	F_3	B_3	F_4	B_4	F_5	B_5
4000	493	48	930	122	2290	53	3355	80	3650	1377
4100	493	50	932	127	2291	50	3350	82	3784	1057
4200	496	60	939	152	2297	49	3348	55	–	–
4300	498	66	943	171	2300	49	3349	51	–	–
4400	502	79	949	209	2305	53	3353	43	4244	207
4500	511	109	954	306	2314	68	3363	42	4294	102
4600	528	150	948	481	2324	96	3374	47	4306	62
4700	563	190	902	823	2330	143	3384	60	4317	49
4800	604	182	730	1537	2325	205	3392	82	4326	47
4900	565	602	2006	233	3213	132	4166	58	–	–
5000	636	138	2278	259	3390	130	4335	57	–	–
	535	152	1137	402	2489	93	3524	60	4132*	415*
	47	150	481	409	385	50	344	14	266*	517*

Table 13.1.: The first five formant frequencies and bandwidths for vowel /oː/ measured with different values of the *ceiling* parameter.

only the indication from acoustic models that approximately every 1000 Hz there might be a formant. Therefore if we want to say anything about the accuracy of formant frequency measurements we have to use sounds where we know beforehand the number of formants. This automatically forces us to use artificial vowel sounds. Let us start by defining three artificial vowels /u/, /i/ and /a/ whose formant frequencies are given in table 13.2.

These three vowels are at the extremes of the vowel triangle an they will be used in the automatic formant frequency measurements. For these vowels the third, forth and fifth formant were were not varied at all and they were the same for these vowels as we know that vowel identity is mainly determined by the first two formants. The fundamental frequency was held constant at 125 Hz. The artificial vowels were produced according to the source-filter model, i.e. a sound source was filtered by a number of consecutive formant filters. The details of these sounds can be found in the following section.

Vowel	F1	F2	F3	F4	F5	B1	B2	B3	B4	B5
u	300	600	2500	3500	4500	30	60	250	350	100
a	800	1200	2500	3500	4500	80	120	250	350	100
i	300	2300	2500	3500	4500	30	230	250	350	100

Table 13.2.: The formant frequency values and bandwidths in Hz of three artificial vowels /u/, /i/ and /a/.

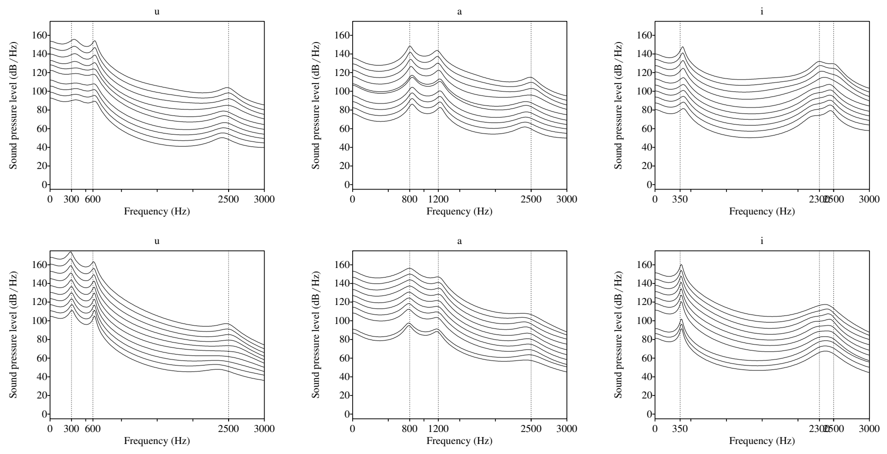


Figure 13.11.: Spectral slices determined from LPC analysis of three different artificial vowel sounds /u/, /a/ and /i/.

13.3.4.2. How did we produce the vowels?

In this section we describe how the vowels that were used for testing the LPC formant frequency analysis were produced. The formant frequency values of these vowels /u/, /a/ and /i/ are given in table 13.2. According to the source-filter model a vowel sound originates by filtering the source signal that originates at the glottis by the resonance filters of the vocal tract [Fant, 1960]. These resonances are called formants. At first order the filter is independent from the source. Therefore to produce vowels we need a specification of the source and a specification of the vocal tract filter.

13.3.5. Practical guidelines for measuring formant frequencies

Most of the time formant frequency values have to be measured for vowels. In order to make the measurements reproducible it is desirable to be able to automate the measuring process as much as possible. This means that already at the *sound file name* level enough structure should be present to be able to deduce as much information from the file name as is needed. The next step is annotating the segments we are interested in. The last step is probably to write a script to perform automatic measurements.

13.3.5.1. File naming

For example, formant analysis needs different parameters depending on whether the sound is from a male, female or child speaker. Depending on the amount of speakers and sounds one has several possibilities. For example in the TIMIT acoustic-phonetic speech database files were organized per speaker and all the files of one speaker resided in a directory with a name that beside the speaker initials started with "m" or "f" to indicate either a male or female speaker. See section B.5.1 for more details on the TIMIT file name conventions.

13.3.5.2. Annotating segments

This is a very important step in the measuring process, annotating the segments that you are interested in with meaningful labels. Chapter 9 introduces how to annotate sounds. Once we have labeled our segments they can be processed automatically because the start and ending times of the labeled segments can be queried from the TextGrid and all kinds of decisions can be based upon these times.

13.3.5.3. Scripting example

Example script B.8 in section B.5.1.4 is an example how to perform a standard formant frequency analysis of all the vowels in the TIMIT database.

13.4. Why are formant frequencies still so difficult to measure?

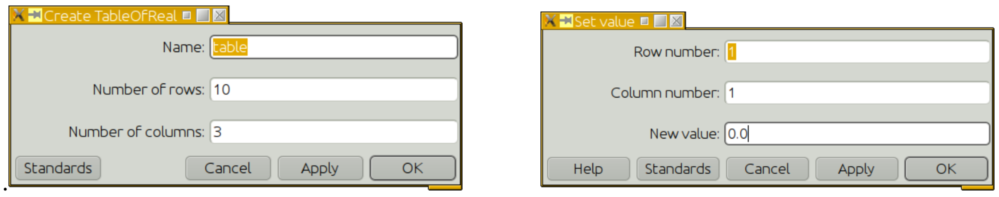
14. Useful objects

14.1. Introduction

In this chapter we describe a number of object types that can be very useful in accomplishing several tasks. For example, if a *Strings* object holds a list of files we can use this list to access all these files from a script. A *TableOfReal* object can be used as intermediate storage of labeled data and a Table object can function as a simple database while both can be exported as tab-separated files to be used in other programs besides Praat. A Permutation object can be used to randomize things.

14.2. TableOfReal

A TableOfReal contains cells that are ordered like a matrix, i.e. each cell can be identified with a row index and a column index. As the name indicates the cells can only contain *real* numbers. The rows and the columns of the TableOfReal may have a label text for identification. If we only need one-dimensional information as row information, like for example vowel labels, a TableOfReal object suffices for holding the data. A TableOfReal can be created from the New > Tables menu with the Create TableOfReal... command. The form that appears allows to fill out a name for the new table as well as the number of rows and columns. After clicking the OK



Name: table
Number of rows: 10
Number of columns: 3

Row number: 1
Column number: 1
New value: 0.0

Figure 14.1.: The form of the Create TableOfReal... and the Set value... commands.

button a new TableOfReal object appears in the list of objects. The cell values will be filled with zeros and both row and column labels will be empty. Column labels can be assigned with the Modify > Set column label (index)... command, while we have the Modify > Set row label (index)... command. Assigning values to the cells can be done in two ways. For assigning the cell values one by one we can use the Set value... command. As the right-hand panel in figure 14.1 shows only one cell value at a particular row and column index can be changed at a time. Filling a table in this way by hand is a lot of work. The other way to directly fill the table is with the Formula... command. The following script fills a selected table with random Gaussian numbers with per column different standard deviations.

```
Formula: "randomGauss (0, col*0.1)"
```

The data in the first column will have 0.1 standard deviation, the data in the second column 0.2, and so forth. We note that if we generate these data with the given specification and afterwards measure the values they will almost never be exactly equal. For example, the following script will generate 1000 random Gaussian numbers with zero mean and standard deviation one.

```
Create TableOfReal: "rg", 1000, 1
Formula: "randomGauss(0,1)"
mean = Get column mean (index): 1
std = Get column stdev (index): 1
writeInfoLine: "mean= ", mean, ", stdev= ", stdev
```

The info window could show something like `mean=0.02489238257436717, stdev=1.0071015745735157`. Each time you run the script above, the mean and standard deviation numbers will be slightly different but always near zero and one, respectively.

In a TableOfReal rows can only differ in one dimension, i.e. they can have different row labels. A TableOfReal is therefore well suited to represent data that do not need more than one label. This is why the TableOfReal is the starting point for a number of data analysis tools in Praat. We mention principal component analysis, discriminant analysis or canonical correlation analysis.

14.2.1. Drawing data from a TableOfReal

The TableOfReal has a number of useful drawing options that may turn out useful. Let us illustrate these options with the well-known data set, the [Pols et al., 1973] data set of the first three formant frequencies of the twelve Dutch monophthongs as spoken by fifty Dutch male speakers. This data set is included in Praat and the following command will make it available as a TableOfReal object in the list of objects.

```
Create TableOfReal (Pols 1973): "yes"
```

The resulting object will have 600 rows and 6 columns. The columns will be labeled “F1”, “F2”, “F3”, “L1”, “L2” and “L3” respectively. The rows will be labeled with the twelve different vowel labels “u”, “a”, “o”, “\as”, “\o/”, “i”, “y”, “e”, “\yc”, “\ep”, “\ct”, and “\ic”. The vowel labels which have a backslash in them are special and use an encoding that guarantees they are always displayed in the same way, irrespective of the computer platform you happen to work on. The twelve vowel labels will display as “u”, “a”, “o”, “ɑ”, “ø”, “i”, “y”, “e”, ʏ”, “ɛ”, ”ɔ”, and “ɪ”.

14.2.1.1. Draw as numbers...

14.2.1.2. Draw as numbers if...

14.2.1.3. Draw scatter plot...

14.2.1.4. Draw box plots...

14.2.1.5. Draw column as distribution...

14.3. Table

A Table has just, like the TableOfReal, cells that are ordered like a matrix. However, in the TableOfReal the cells can only contain numbers while in the Table the cells also may contain text. The columns of the table may have a label. Because cells may contain text, a Table doesn't have row labels.

14.4. Permutation

A Permutation object represents one of the possible permutations of n things.

14.5. Strings

A Strings object represents an ordered list of strings. For example, one of the uses of a Strings object in Praat is to hold a list of files.

Part II.

Advanced features

15. The point process

The PointProcess object represents a point process which is a sequence of times t_i defined on a domain $[t_{\max}, t_{\min}]$. The index i runs from 1 to the number of points in the point process. The points t_i are ordered such that $t_{i+1} > t_i$ which implies that all the time points t_i are unique, i.e. no two time points can be equal.

16. LPC analysis

16.1. Introduction

Most of the formant frequency determination algorithms that are used nowadays are based on linear predictive coding, i.e. linear predictive coding or LPC analysis is the algorithm that is behind formant frequency analysis. Linear predictive coding started its popularity in the speech analysis domain in the 1970's. It is based on the source-filter model of speech production which models the speech signal as a combination of a source signal and a filter. For vowels the vocal folds are associated with the source and the vocal tract with the filter. The source signal is filtered by the vocal tract filter to produce the sound.

The assumptions implicitly made in performing an LPC analysis are

- The speech signal is semi-stationary, i.e. during our analysis interval the signal characteristics don't change too much.
- The vocal folds are modeled as a pulse train or white noise.
- The source and the filter are independent of each other.
- The combined effects of the glottis, the vocal tract and the radiation of the lips can be modeled by an all pole filter.

This chapter will give a reasonably technical introduction of the LPC algorithm. For more advanced theory we refer to Markel and Gray [1976] and Makhoul [1975].

16.2. Linear prediction

In linear prediction the problem can be simply stated as: can we predict what the next sample value, s_n will be if we have the previous p sample values at our disposal. A model linear in its parameters is the simplest model for s_n and we write our prediction of s_n as

$$\hat{s}_n = -\sum_{k=1}^{p} a_k s_{n-k}, \tag{16.1}$$

where $\hat{s}_n$ indicates that it is the predicted value for s_n, and the p parameters a_k are the linear prediction coefficients. Convention says that we start with a minus sign. Equation (16.1) shows that we need at least $p + 1$ sample values, $s_1, s_2, ..., s_{p+1}$, to calculate the prediction coefficients. We will show how this calculation works and assume that sample values that occur before s_1 are all zero. We can use the following chain of equations to solve for the a_k: for the first sample we have no previous ones so we start with the second sample for which we only have one previous sample and model it as $\hat{s}_2 = -a_1 s_1$. We can solve for a_1 by substituting the real sample value s_n as the predicted one which results in $a_1 = -s_2/s_1$. For the prediction of the third sample value we use $\hat{s}_3 = -a_1 s_2 - a_2 s_1$. The value of a_1 is known and by substitution the value of s_3 as the predicted value we find that $a_2 = \frac{s_3 + a_1 s_2}{s_1}$. We proceed in this way to the fourth sample to get prediction coefficient $a_3 = \frac{s_4 + a_1 s_3 + a_2 s_2}{s_1}$. In this way we can solve for all the p coefficients.

If we have more sample values at our disposal than the bare minimum $p + 1$ values, say sample values s_1 to s_N where N is (preferably much) larger than $p + 1$, we would like to calculate only one set of a_k's that describe these N sample values best. We now have more than $p + 1$ values to choose from, how do we obtain the a_k values? The best choice would be to choose those a_k that minimize the difference between the real values s_n and the predicted values $\hat{s}_n$ from the model. The following notation formalizes this. We start by defining the sample-wise prediction error at each sample index n as

$$e_n = s_n - \hat{s}_n. \tag{16.2}$$

As the measure E, for the total error made in the prediction, we accumulate the squares of the sample-wise errors to obtain

$$E = \sum_{n=1}^{N} e_n^2. \tag{16.3}$$

The sum of *squares* is used because this makes the further handling mathematically more tractable[1]. The error E is a function of the parameters a_k, this could be indicated by writing it as $E(a_1, a_2, \ldots, a_p)$, but nobody does so. We can find the extrema of the E function by taking partial derivatives with respect to the a_i and assigning the result to zero. These partial derivatives are

$$\frac{\partial E}{\partial a_i} = \sum_{n=1}^{N} 2 e_n \frac{\partial e_n}{\partial a_i} = 0, \qquad \text{for} \qquad 1 \le i \le p. \tag{16.4}$$

To solve these p equations we rewrite the e_n using eq. (16.1) as

$$\begin{aligned} e_n &= s_n - \hat{s}_n \\ &= s_n + \sum_{k=1}^{p} a_k s_{n-k}. \end{aligned} \tag{16.5}$$

The partial derivative of e_n with respect to a_i is then

$$\frac{\partial e_n}{\partial a_i} = \frac{\partial}{\partial a_i}\left(s_n + \sum_{k=1}^{p} a_k s_{n-k} \right) = s_{n-i}.$$

[1] We could also accumulate the absolute values of the differences between the real value and the predicted value as $E = \sum_{i=1}^{N} |e_n|$. However, the absolute signs make it more difficult to use standard minimization techniques.

Using this together with eq. (16.5), eq. (16.4) becomes

$$\begin{aligned}\frac{\partial E}{\partial a_i} &= \sum_{n=1}^{N} 2\left(s_n + \sum_{k=1}^{p} a_k s_{n-k}\right) s_{n-i} \\ &= 2\sum_{n=1}^{N}\left(s_n s_{n-i} + \sum_{k=1}^{p} a_k s_{n-k} s_{n-i}\right) = 0.\end{aligned}$$

Leaving out the factor 2 and rearranging gives

$$\sum_{k=1}^{p} a_k s_{n-k} s_{n-i} = -\sum_{n=1}^{N} s_n s_{n-i}, \qquad \text{for} \qquad 1 \le i \le p \tag{16.6}$$

There are many ways to solve these equations and they depend on the what we assume what the sample values are outside the range $s_1 \dots s_N$. In the equation above there are products of sample values like $s_{n-k}s_{n-i}$ where for certain combinations of n and k and/or n and i the resulting index is smaller than 1. In the sequel our assumption will be that these sample values with an index smaller than one are all zero. This corresponds to what in the literature has become the *autocorrelation* method. Given that only $s_1$to s_N are different from zero we can define the (autocorrelation) coefficients R_i as

$$R_i = \sum s_n s_{n+|i|}. \tag{16.7}$$

We then rewrite eq. (16.6) as

$$\sum_{k=1}^{p} a_k R_{|i-k|} = -R_i$$

This can be written in matrix form as

$$\begin{pmatrix} R_0 & R_1 & \cdots & R_{p-1} \\ R_1 & R_0 & \cdots & R_{p-2} \\ \cdots & \cdots & \cdots & \cdots \\ R_{p-1} & R_{p-2} & \cdots & R_0 \end{pmatrix}\begin{pmatrix} a_1 \\ a_2 \\ \cdots \\ a_p \end{pmatrix} = -\begin{pmatrix} R_1 \\ R_2 \\ \cdots \\ R_p \end{pmatrix}.$$

In matrix-vector notation we write $\mathbf{Ra} = -\mathbf{r}$ which has the solution $\mathbf{a} = \mathbf{R}^{-1}\mathbf{r}$ if the inverse exists. Normally inverting a matrix is an $O(p^3)$ computation. However, it is possible to use the symmetry of $\mathbf{R}$ to find numerically more efficient algorithms. The matrix $\mathbf{R}$ has Toeplitz symmetry as first of all it is a symmetric matrix and all descending diagonals are constant. The Levinson algorithm uses this symmetry and needs $O(p^2)$ operations to find the solution. However, the number of operations involved in calculating the autocorrelation coefficients exceeds the number of calculations involved in the matrix inversion most of the times. Using the autocorrelation method for solving the linear prediction equations always guarantees a *stable* filter. What we mean by stable will be expressed later on in this chapter when we discuss the Z-transform, but simply said it guarantees that if we use the equations as a filter, its impulse response will gradually fade away and not blow up.

16.3. Linear prediction applied to speech

The speech sound can be modeled as the result of filtering a source sound by the vocal tract filter. This translates to linear prediction as the following

$$s_n = \sum_{k=1}^{p} a_k s_{n-k} + u_n. \tag{16.8}$$

Here u_n is the source and the $a_k$represent the characteristics of the vocal tract filter. This equation says that the output s_n is predicted from p previous outputs and a source u_n. Because we do not know the source signal u_n we proceed by first neglecting it and stating that our best estimate of s_n would be a linear combination of p previous values only, i.e.

$$\hat{s}_n = -\sum_{k=1}^{p} a_k s_{n-k}. \tag{16.9}$$

In doing so we make an error $e_n = s_n - \hat{s}_n$ for each sample. Using eq. (16.9) we write

$$e_n = s_n + \sum_{k=1}^{p} a_k s_{n-k}, \tag{16.10}$$

which can also be written as

$$s_n = \sum_{k=1}^{p} a_k s_{n-k} + e_n. \tag{16.11}$$

Now the error e_n has appeared at the position of the source u_n in eq. (16.8). This is nice, we can use the linear prediction method outlined in the previous section to estimate the filter coefficients a_k and then use eq. (16.10) to give us the estimate of the source. It seems that we get two for the price of one: we start with the sound and get the filter as well as the source. Of course this can not be the whole truth. What happens is that part of the sound is modeled by the filter and that what has not been modeled ends up in the source. So in the end we still don't know exactly what the source was, it just adds up to all what is in the sound that is not modeled by the filter. The better our model fits the sound, the better our estimates of the vocal tract filter and the source signal.

16.4. Intermezzo: Z-transform

For the analysis and description of sampled systems we can use a special mathematical device which is called the Z-transform (ZT). We use this technique because the equations that govern digital filters are *difference equations*. For example a formant filter is described as

$$y_n = ax_n + by_n + cy_{n-1}, \quad \text{for} \quad 1 \le n \le N.$$

Typically this equation is evaluated at discrete times nT which are multiples of the sampling period, i.e. the index n refers to units of the sampling period T. The Z-transform offers a technique to tackle this equation to arrive at a description of the filter characteristics in the frequency domain.[2]

The Z-transform for a sampled signal y_n is defined as

$$Y(z) = \sum_{k=0}^{\infty} y_k z^{-k}, \qquad z\in .$$

Applying this transformation to the sequence y_n results in the Z-transform $Y(z)$. The Z-transform is a function in the complex z-plane. The Z-transform is important because it is a generalization of the Discrete Fourier transformation (DFT). If we evaluate the Z-transform at discrete points on the unit circle, i.e. for values of $z = e^{2\pi i m/N}$, where $0 \leq m \leq N-1$, the Z-transform for a particular value of m is $\sum_{k=0}^{N-1} y_k e^{2\pi i k m/N}$, which equals the term in the DFT formula in section (7.7). This means that we can get the frequency response of any system described by a Z-transform by evaluating the Z-transform for values of z on the upper half of the unit circle, i.e. for values of $z = e^{2\pi i f T}$, where frequency f runs from zero to the Nyquist frequency $1/2T$.

An more mathematically refined introduction to the Z-transform can be found in for example Papoulis [1988], in here we will only use some of the properties of the Z-transform that are convenient to us. If we define the operation $ZT(y_n)$ as applying the Z-transform to all the samples of a signal y_n we can write $ZT(y_n) = Y(z)$, i.e. the result of applying the Z-transform to the signal y_n is $Y(z)$. The inverse operation can then be written as $ZT^{-1}(Y(z)) = y_n$. Although we will not use the inverse Z-transform, we will use the notation $y_n \Longleftrightarrow Y(z)$ to denote a Z-transform pair. The most important properties of the Z-transform are:

1. $ay_n \Longleftrightarrow aY(z)$. If $Y(z)$ is the Z-transform of y_n then to obtain the Z-transform of ay_n, i.e. the signal multiplied by a common scale factor a, multiply the Z-transform of y_n with this factor and vice versa. With the notation developed above we can also write this as $ZT(ay_n) = a \cdot ZT(y_n)$ and $ZT^{-1}(aY(z)) = a \cdot ZT^{-1}(Y(z))$.

2. $ax_n + by_n \Longleftrightarrow aX(z) + bY(z)$. The Z-transform is a linear operation.

3. $y_{n-1} \Longleftrightarrow z^{-1}Y(z)$. The notation y_{n-1}means a time shift of the signal one sample back, i.e. the signal is shifted one step to the left. We have to repeat this shifting to obtain the more general result $y_{n-k} \Longleftrightarrow z^{-k}Y(z)$: the Z-transform of a signal shifted backward in time over k sample times is obtained by multiplying its Z-transform by a factor z^{-k}.

4. $x_n \star y_n \Longleftrightarrow X(z)Y(z)$. Convolution in the sampled domain is like multiplication in the z-domain.

These three properties are all we need to describe the frequency characteristics of a digital filter. For example, applying the Z-transform on the filter equation $y_n = ax_n + by_{n-1} + cy_{n-2}$ results in

$$ZT(y_n) = ZT(ax_n) + ZT(by_{n-1}) + ZT(cy_{n-2}).$$

This translates to

$$Y(z) = aX(z) + bz^{-1}Y(z) + cz^{-2}Y(z).$$

[2]Analog systems, for example electrical circuits, are typically described by differential equations and there the Laplace transform is the preferred tool.

We gather terms and obtain

$$Y(z)(1 - bz^{-1} - cz^{-2}) = aX(z).$$

Now define the filter function $H(z)$ as

$$H(z) = \frac{Y(z)}{X(z)}.$$

This definition states that if you divide the output (representation) of a filter by its input (representation) you will get the characteristics of the filter. For the formant filter this turns out to be

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a}{1 - bz^{-1} - cz^{-2}}. \tag{16.12}$$

This equation can be put in another form by extracting the z^{-2} from the denominator which results in

$$H(z) = \frac{az^2}{z^2 - bz - c}.$$

The denominator of this quotient is a second degree polynomial in z and has real coefficients, i.e. 1, b and c. The zeros are located at values

$$z_{1,2} = \frac{b \pm \sqrt{b^2 + 4c}}{2}. \tag{16.13}$$

This shows that the zeros are either real, or complex if $b^2 + 4c < 0$. If they are complex, then always $z_1 = z_2^*$, i.e. they are complex conjugate. For now we only consider the complex zeros because it will turn out that they are more interesting than the real zeroes. Now $H(z)$ can be expressed as

$$H(z) = \frac{az^2}{(z - z_1)(z - z_1^*)} \tag{16.14}$$

16.4.1. Stability of the response in terms of poles

We first express the poles in polar coordinates: $z_{1,2} = re^{\pm i\phi}$. A Z-transform that has the form of (16.12), i.e. with two complex conjugate poles comes from a sampled signal that can be written as $y_n \sim e^{n \cdot \ln r/\pi} \sin(n\phi)$. This is an oscillating sine function multiplied by an exponential function. If $\ln r < 0$ the exponent becomes negative and we get an exponentially *damped* sine. For large enough sample indices n the sample values become zero. This only happens if r is less than one which is the case if the pole is located within the unit circle. For values of r greater than one, the $\ln r$ term in the exponent becomes positive and sample values keep growing towards infinity. The latter system is *unstable*. If r equals one there will be no damping at all and an oscillating signal of constant amplitude will result. It is possible to generalize this to a system with more poles than the two considered here: *stable systems have their poles located within the unit circle*.

16.4.2. Frequency response

The frequency response of $H(z)$ can be obtained by evaluation of $H(z)$ for values of z on the upper half of the unit circle. This means evaluation for values of $z = e^{2\pi i f T}$, where f runs from 0 to the Nyquist frequency $1/2T$. As an example let us try to evaluate the frequency response of the filter in equation (16.14). We are only interested in the amplitude response and leave phase out of the picture. We write

$$|H(z)| = \frac{|az^2|}{|(z-z_1)(z-z_1^*)|} = \frac{|a|}{|z-z_1|\cdot|z-z_1^*|}. \tag{16.15}$$

This lends itself to a geometrical interpretation as is shown in figure (16.1). On the left the

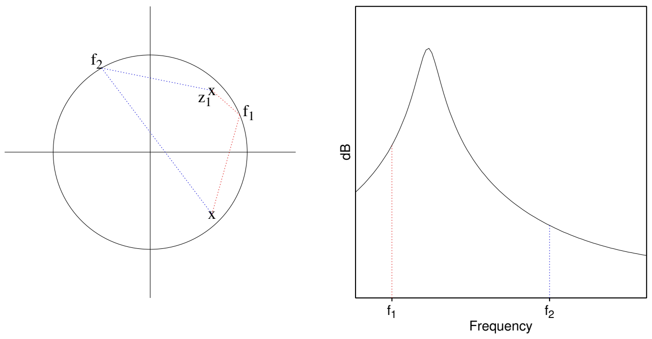


Figure 16.1.: The positions of the poles of a formant filter in the z-plane (left) determine the frequency response (right).

z-plane is drawn. The unit circle is shown and the two conjugate poles are indicated with a cross. On the right the frequency response is shown as the point $z = e^{2\pi i f T}$ moves on the unit circle. The point starts at $z = 0$ which corresponds to a frequency of 0 Hz, then travels on the top half of the unit circle, through $z = i$ which corresponds to a frequency of one half of the Nyquist and then ends at $z = -1$ which corresponds to the Nyquist frequency. The frequency response at any of these positions, for example, at the position labeled as f_1, is the inverse of the product of the distances of the point to both poles as equation (16.15) says. For a point f_1 these two distances are indicated with red dotted lines. In the right figure the (logarithm of the) inverse product of these distances is shown with a red dotted line. For another point f_2 these distances are shown in blue. The resulting response is smaller than for the point $f_1$because both distances are larger and their product is therefore also larger and the response being the inverse of this product will be smaller. If the point z is near the pole z_1 one of distances becomes very small and the product of the two distances will be small too. The inverse will be large then, hence a peak in the response. We can deuce from this that the closer a pole lies to the unit circle, the higher the peak will be and smaller its bandwidth.

What we showed here for a two pole system can be generalized to more poles. If z moves over the unit circle then every time it moves towards a pole the response goes up. In general the frequency response, i.e. the spectrum will then show as many peaks as there are poles in the upper half plane. Of course not all the poles in the upper half plane will always have a real peak in the spectrum associated with them because

1. the location of a pole might be so far away from the unit circle that the peak's bandwidth is so large that it will not be noticeable as a peak at all.
2. two poles are so close together that they merge together into one peak.
3. the pole is close to the real axis or might in fact be real. In the real part is positive the spectrum fill show a gradual fall off due to a peak near zero hertz, the impact of this real pole is like a low-pass effect on the spectrum . If the real part is negative a high-pass effect on the spectrum results.

16.5. LPC interpretation

Now with the Z-transform at our disposal we can try to bring the equations in section (16.3) in a more manageable form to obtain the transfer function $H(z)$. We started with equation

$$s_n = -\sum_{k=1}^{p} a_k s_{n-k} + u_n,$$

whose transfer function can be written as

$$H(z) = \frac{S(z)}{U(z)} = \frac{1}{1 + \sum_{k=1}^{p} a_k z^{-k}}.$$

This is the standard formulation if we write it as $S(z) = H(z)U(z)$, i.e. it says that in order to get the output $S(z)$ you apply the filter $H(z)$ on the input $U(z)$. In the z-domain the Z-transform of the output can simply be obtained by a multiplication of the Z-transforms of the filter and the input. However, another description exist which is called the *inverse filter* formulation. Taking the Z-transform of equation (16.11) results in

$$E(z) = A(z)S(z), \tag{16.16}$$

where $A(z)$ is called the analysis filter

$$A(z) = 1 + \sum_{k=1}^{p} a_k z^{-k}. \tag{16.17}$$

Because we don't know what the source signal $U(z)$ is we assumed $U(z) = E(z)$. The analysis filter being $A(z) = 1/H(z)$ as can be easily verified.

The denominator of $H(z)$ is a polynomial of degree p in z whose poles are either real or complex conjugate pairs. If the system $H(z)$ is stable and all poles are complex then $p/2$ poles are located in the upper half plane. The spectrum will show maximally $p/2$ peaks as was explained in section (16.4.2).

16.6. Performing LPC analysis

We start with a warning: if you only use LPC analysis to get formant frequency values it is advisable to first resample the sound to a sampling frequency of 10 kHz for a male voice, or to 11 kHz for a female voice. For a male voice there is approximately one formant in a 1000 Hz interval and this leaves room for five formants in the interval from zero to 5000 Hz. For a female voice we have one formant per 1100 Hz and to measure five formants we therefore need an upper limit of 5500 Hz. Most of the energy (and therefore also information) in a speech sound is, at least for male voiced sounds concentrated below 5000 Hz. This is because the combined effect of the sound source and the radiation at the lips makes the spectrum of a vowel sound decline with approximately 6 dB per octave. The spectrum of the source signal alone has a slope of approximately −12 dB/oct. However, when the sound leaves the mouth at the lips, the low frequencies in the sound immediately spread out in all directions while the high frequencies in the sound stay much more directed. The combined effect of low frequencies spreading out and high frequencies staying more directed is like the effect of a high-pass filter with a slope of +6 dB/oct. Taking the effect of the −12 dB/oct slope at the source together with the +6 dB/oct slope of the effects at the lips results in a spectrum with a slope of approximately −6 dB/oct.

A number of different LPC analyses have been implemented such as the autocorrelation method and the covariance method [Markel and Gray, 1976], the Burg method [Childers, 1978], the combination of forward and backward prediction of Markel and Gray [1976] or the robust method as described by Lee [1988]. The method that is commonly used in the automatic formant analysis is the Burg method.

16.6.1. Pre-emphasis

As we explained in the previous section, the spectrum of a vowel sound has a slope of approximately -6 dB/oct. In general this makes the amplitude of peaks at higher frequencies weaker than those at lower frequencies. This has a negative effect on the LPC analysis because these peaks would not get enough impact in the analysis procedure. The analysis automatically matches stronger peaks better than less stronger peaks. Perceptually the peaks at higher frequencies are important because the power at higher frequencies is more concentrated on the basilar membrane because of membrane's logarithmic frequency to place relationship. It would be nice if we could compensate for the negative slope of the spectrum to give the spectral peaks at higher frequencies the same impact as the ones at lower frequencies. By high-pass filtering the sound before performing the actual analysis we can compensate. The simplest high-pass filter that accomplishes can accomplish this task is of the form $y_n = x_n - ax_{n-1}$. The effect of this filter is a +6 dB/oct correction applied to the spectrum. The coefficient a is always near the value one and is calculated as

$$a = e^{-2\pi FT},$$

where T is the sampling period and F is chosen as the frequency from which we apply the +6 dB/oct correction. The default value for F is 50 Hz. For a male voice with 5000 Hz as the maximum formant and $F = 50$ Hz, the coefficient a turns out to be

$$a = e^{-2\pi 50/10000} \approx 0.969.$$

Applying such a high-pass filter on a sound is called *pre-emphasis*. In figure 16.2 the effect of

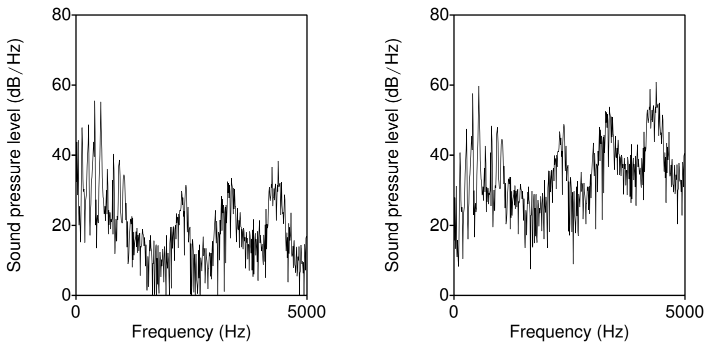
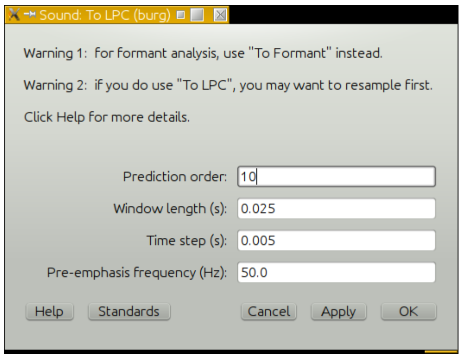


Figure 16.2.: The effect of pre-emphasis on the spectrum of the vowel /oː/ from a male speaker. The spectrum on the right was pre-emphasized.

applying default pre-emphasis is shown on the spectrum of a vowel /oː/ from a male speaker. The left spectrum is without pre-emphasis, the one on the right was pre-emphasized above 50 Hz. It is clear that the high frequency peaks have been emphasized. This will result in a better spectral modeling from a following LPC analysis.

16.6.2. The parameters of the LPC analysis

The minimum parameters that all LPC analyses need are shown in the form that appears after choosing **To LPC (burg)...**, as shown by figure 16.3. The form shows the settings for a



Warning 1: For formant analysis, use "To Formant" instead.

Warning 2: if you do use "To LPC", you may want to resample first.

Click Help for more details.

Prediction order:	10
Window length (s):	0.025
Time step (s):	0.005
Pre-emphasis frequency (Hz):	50.0

Figure 16.3.: The Sound: To LPC (burg)... form with parameter settings.

prediction order of 10 which means that we want to match 5 formant peaks in the spectrum.

A pre-emphasis is applied for frequencies of 50 Hz and above. The window length of 0.025 s assumes that the spectral characteristics are approximately stable within this time interval. For normally uttered speech this is most of the time a reasonable assumption.

16.6.3. The LPC object

The result of applying an LPC analysis on a sound will be an object of type LPC. This object incorporates the (local) filter characteristics of the vocal tract. The filter characteristics as a function of time are represented in the LPC object as an array of LPC frames. Each LPC frame contains the maximally p filter coefficients a_i that were the result of the analysis. Equation (16.16) shows that if we invert the filter and apply it to the sound we get the source signal.

17. Dynamic time warping

17.1. Introduction

Dynamic time warping (DTW) is a technique to align two time sequences, like for example, two sounds. If we produce the same sentence at two different speech rates, fast and slow, a detailed analysis of the two sounds shows that not all segments of the two sounds show the same time stretching or compression [Sakoe and Chiba, 1978]. Therefore techniques that try to align these two sentences by a linear strectching of time generally will not achieve their goal. To obtain better alignment we need a technique that can stretch or compress locally with different scale factors. This technique is called Dynamic Time Warping (DTW) and was introduced by Sakoe and Chiba [1978] into the speech signal processing community. The DTW algorithm is part of the domain of *dynamic programming* techniques. Dynamic programming is also used in Praat's pitch post-processing step.

17.2. The DTW algorithm

For the description of the algorithm we assume that for the two sounds Q and C that we want to align, an analysis has been performed. Trying to align two sounds on the basis of their sample values is generally a very bad idea because we know that sounds that look very dissimilar might be perceived as being similar and vice versa. In section 7.1.9 we showed that phase is not very important to how we hear things and that we can vary the phase, and as a consequence the sample values, without having any consequence on perception. It is therefore advisible to use a spectral representation of the speech signal. The most used spectral representation in this respect is with mel frequency cepstral coefficients often abbreviated as MFCC We will explore the MFCC representation in section 17.3. For the description of the DTW algorithm we only assume that we have an analysis of the sound at regularly spaced time intervals and that we can calculate the distance between any two analysis frames (or *feature vectors*). Let us assume that the sound Q has analysis frames $(q_1, q_2, \ldots, q_n)$ and the sound C has analysis frames $(c_1, c_2, \ldots, c_m)$. The first step in the algorithm is to construct a distance matrix $\mathbf{D}$ of dimension $n \times m$ in which each cell d_{ij} contains the distance between analysis frames i and j. An example of such a matrix is shown in figure 17.1. The distance values d_{ij} in the matrix are indicated with grey colouring. Black colour indicates large distances while white colour indicates small distances. The distance matrix is shown with the oscillogram of the Dutch sentence "Er was een een oud kasteel" as spoken by a female speaker on the left vertical scale, and the oscillogram of a slower resynthesized version below the horizontal axis. The horizontal sound is approximately 1.5 times as long as the vertical sound. The distance matrix for this particular example has 324×497 cells. In the distance matrix a warping path is shown, this is the line that starts in the lower ledt point of the matrix and ends in the upper right. The

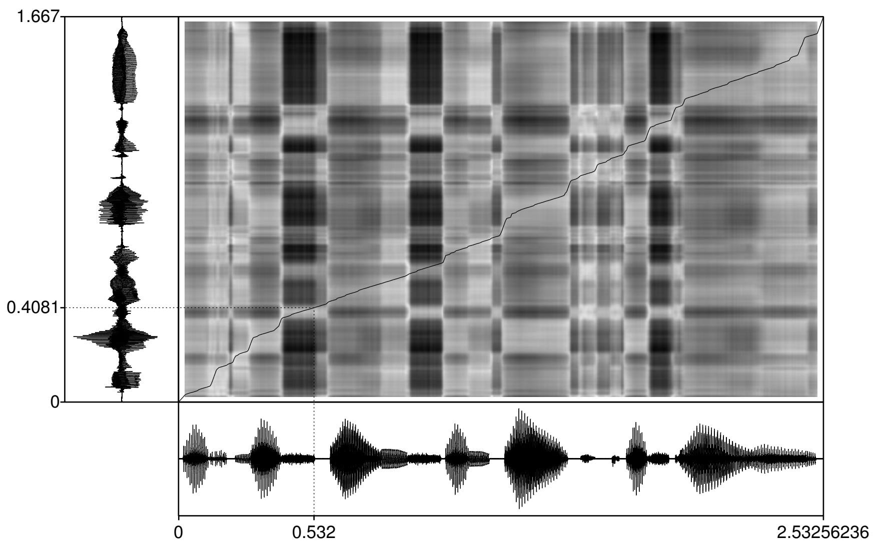


Figure 17.1.: The dynamic time warp of two versions of the Dutch sentence “Er was eens een oud kasteel” (*Eng.* Once upon a time there was an old castle).

warping path shows how time at the horizontal axis corresponds best with time on the vertical axis. For example, if we start at time $t_x = 0.532$ s in the horizontal sound and follow the dotted line to where it crosses the warping path and then follow the dotted horizontal line to the left, we arrive at time $t_y = 0.4081$ s in the vertical sound. This means that time 0.532 s at the horizontal time axis is warped to time 0.4081 s on the vertical time axis. And, of course, this goes the other way around too: the time 0.4081 s on the vertical axis is warped to the time 0.532 s on the horizontal axis. As the figure shows the warping path is not a straight line running from bottom left to top right. If it were so, both consonants and vowels in both sounds would be streched by an equal amount and this almost never happens if both sounds are naturally produced utterances. If people speak the same sentence twice, first slowly and then fast, not all parts in the fast sound will be shortened in the same proportion.[1] The DTW algorithm tries to calculate a warping path that is optimal in some sense.

17.3. Mel Frequency Cepstral Coefficients (MFCC)

Obtaining a representation of the speech sound in terms of mel frequency cepstral coefficients is a four step process.

1. The analysis windows are determined by windowing part of the sound as was described

[1]Speech rate is normally expressed as words per minute. However, several studies have indicated that if speakers change their speech rate, this has effects on various levels of the temporal structure of speech.

in section 2.4.

2. The sound in each analysis window is converted to a spectrum.
3. The spectrum is reduced by a weighted binning of neighbouring spectral values on a mel frequency scale.
4. A cosine transform is applied to the binned spectrum and only a limited number of the resulting coefficients are kept.

The calculation of the MFCC's for speech sounds is most of the time performed on sounds that are band-limited to 8000 Hz. This will reduce the number of computations involved without noticeable degradation of the sound. The 8000 Hz is sufficient to also capture spectral differences between fricatives which of all speech sounds extend highest in frequency. In the following sections we will discuss the individual steps separately. In figure 17.2 the necessary steps are outlined separately with the sound /kɑsteːl/, produced by a female speaker, as input.

1. *The analysis window.*
   As was outlined in section 2.4, the choice of the duration of the analysis window is directly related to the underlying analysis model that says that during this time interval we consider the statistics of a speech sound to be more or less stable. A reasonable choice for spectral analysis is a 25 ms window length. Because the windowing function is a Gaussian the actual window length is 50 ms. In figure 17.2.a one of these windows is indicated with a red colour. The effect of applying the window function to the part of the sound in the analysis window is show in figure 17.2.b. We can deduce from the latter figure that the fundamental frequency in this fragment is approximately 160 Hz as approximately eight periods fit into the 50 ms segment.
2. *Sound to spectrum.*
   The windowed sound of figure 17.2.b is transformed to a spectrum which is displayed in figure 17.2.c. The spectrum shows an harmonic structure because the analysed fragment is taken from the vowel /eː/ of the /kɑsteːl/ sound. The distance between the harmonics at the lower end of the spectrum is approximately 160 Hz, which, of course, conforms with the sound fragment in figure 17.2.b. We further notice the first formant of the /eː/ lying somewhere between the second and the third harmonic. This formant also is very noticeable in 17.2.b where somewhat more than two period of the first formant fit into on fundamental period. The second formant in the spectrum is also visible at near 2300 Hz.
3. *Spectrum to MelFilter.*
   To get rid of some of the "unnecessary details" in the spectral representation, the power in the spectrum is binned with special triangular weighting filters. The centre frequencies of these filters are at a constant distance from each other on a mel scale. This is a frequency scale developed by Stevens et al. [1937] to relate the judgements of pitch differences to the actual frequencies. The relation between mels and hertz is given by the following formula:

$$\text{mels} = 2595 \log(1 + \text{hertz}/700) = 1127 \ln(1 + \text{hertz}/700). \tag{17.1}$$

Its inverse is the following

$$\text{hertz} = 700\left(10^{\text{mel}/2595} - 1\right) = 700(e^{\text{mel}/1127} - 1). \tag{17.2}$$

The first formula expresses that a frequency in mels is a logarithmic function of the frequency in hertz. The numbers in the formula were chosen such that a frequency of 1000 Hz corresponds to a value of 1000 mels. Several other formulas exist in the literature that relate mels to hertz.[2]

[2]See for example the wikipedia entry on "Mel scale" for more definitions.

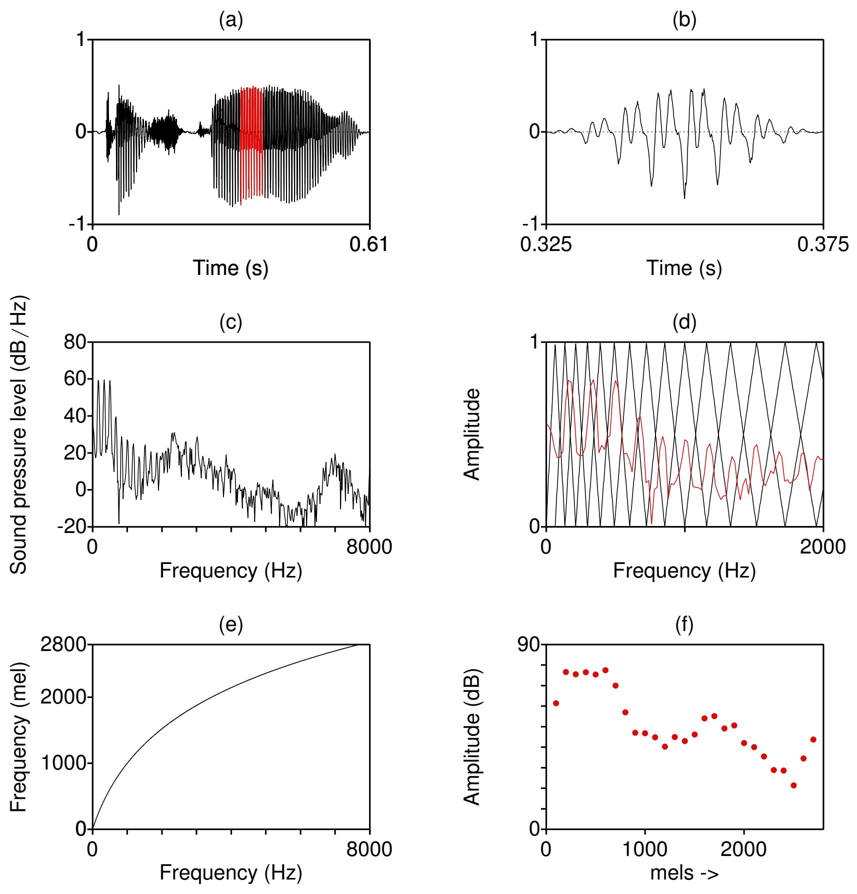


Figure 17.2.: From sound to mel filter coefficients. (a) Selected part of the sound in red colour, (b) after windowing, (c) the spectrum, (d) in red part of the spectrum with overlaid triangular filters in black, (e) the hertz to mel function, (f) the filter values.

18. Scripting simulations

18.1. Introduction

In this chapter we will discuss an extension of standard scripting: the demo window. In the scripting language we have used up till now we only had only limited user interaction during the execution of the script. The only user interaction possible was by filling out a form which might pop up during the execution of a script. Sometimes it is desirable to have more interaction during the execution of the script by letting subsequent actions depend on specific user input.

A. Mathematical Introduction

A.1. The sin and cos function

These functions originally were invented for characterizing triangles in trigonometry, the branch of mathematics that studies the relations of the sides and angles of triangles, the methods of deducing from certain given parts other required parts, and the general relations between the trigonometrical functions of arcs or angles. [1913 Webster]

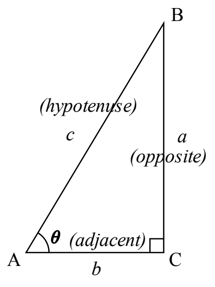


Figure A.1.: A triangle for defining the sine function.

The sine of the angle BAC ($\sin\theta$), for example, is defined as BC/AB, the ratio between the *opposite* side and the *hypotenuse* of a *right triangle*. The cosine (cos) of this angle is defined as the ratio between the *adjacent* side and the hypotenuse (AC/AB). It is clear that since the hypotenuse of a triangle is always the longest side, the maximum value these two functions can attain is one. Note that if we make the length of AB equal to one, the sine and cosine reduce to the length of the sides BC and AC, respectively.

Table A.1.: The triangle relations

	Denomination	Value
$\sin\theta$	$\frac{\text{opposite}}{\text{hypotenuse}}$	$\frac{a}{c}$
$\cos\theta$	$\frac{\text{adjacent}}{\text{hypotenuse}}$	$\frac{b}{c}$
$\tan\theta$	$\frac{\text{opposite}}{\text{adjacent}}$	$\frac{a}{b}$

In general we are not interested in the sine of one particular angle, we want to know how the value of the sine varies if the angle varies. We like to consider the sine as a *function* that varies with its one variable: its angle. As a consequence the definitions above needs to be extended: the triangle is placed in a circle with the point A at the centre and B on the circumference, i.e. we position the triangle in the unit circle, a circle with a radius of length one. The generalization is than simple: we now allow the point B to travel along the circle and define the sine as the length of BC and the cosine as the length of AC.

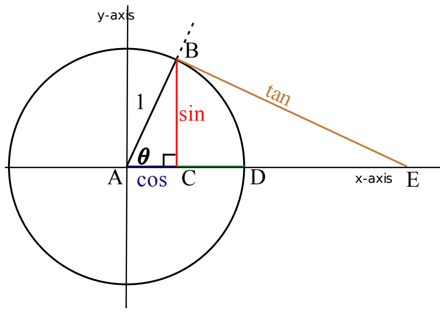


Figure A.2.: The unit circle for defining trigonometric functions.

Another way of viewing this is the cosine as the projection of the point B on the horizontal x-axis and the sine as its projection on the vertical y-axis. With respect to the centre A, we define the lengths to the right as positive, the lengths to the left as negative, the lengths above A as positive and the lengths below A as negative. We now allow the point B to move on the circle, and draw the cosine and the sine as a function of the distance that the point B travels. We start when B lies on the point D and make a left turn, i.e. we go upwards.

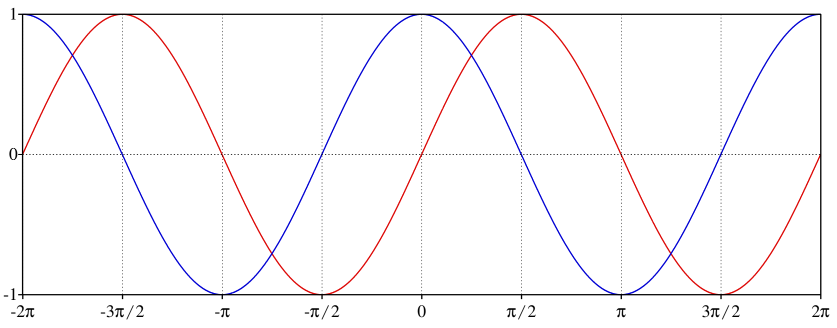


Figure A.3.: The sin and cos functions.

If we start at D, where AC=1 and BC=0, and make a quarter left turn, we have AC=0 and BC=1 and B has traveled a distance of one fourth of the circumference of the circle $2\pi/4$.[1] At

[1] To travel around a square with sides of length l, we have to travel a distance of $4l$. This means that the relation between the circumference and the diameter is exactly 4 for a square. The relation between the circumference and the diameter of a circle can not be calculated that easily. It is not an integer number, it is not a rational number (i.e. it can not be expressed as p/q, the quotient between two integer numbers p and q). The irrational number

the end of the next quarter turn AC=-1, BC=0 and B has traveled half the circumference and traveled a distance π. At three quarters AC=0 and BC=-1 and B is at $3\pi/2$. After a complete turn B has traveled a distance of 2π and the values of AV and BC are the same as when we started.

In the figure above, the sine and cosine trace the red solid and the blue dotted line from 0 to 2π. If the point B had taken the right turn then the red and blue curves from 0 to -2π would have been traced. The figure makes clear that sine and cosine have the same form and are merely displaced versions of each other. This displacement is $\pi/2$. We can write that as: $\sin\theta = \cos(\theta - \pi/2)$. This equation means that the value of the sine at a particular argument θ is equal to the value of the cosine function that is displaced $\pi/2$ to the left. We can also write: $\cos\theta = \sin(\theta + \pi/2)$, i.e. the cosine is equal to a sine function displaced to the right by $\pi/2$.

When the point B takes another turn around the circle, the lengths AC and BC behave in exactly the same way as during the first turn. We can extend the figure above to 4π by just copying the two functions between 0 and 2π and pasting it in the region between 2π and 4π. The fit at the joint is perfectly smooth and continuous. Now add a third turn, and a fourth... We end up with perfect repetitions of the first turn.

We started this story with a left turn but we could have taken a right turn as well. If we count the distance traveled as a negative number we can extend the figure on the left by just copying the stretch between 0 and 2π and pasting it on the left.

The stretch between 0 and 2π is one period and this shows that sine and cosine are *periodic* functions with period 2π. This periodicity means that if you take *any* stretch of length 2π it will be exactly equal to the following and to the previous stretch.

A.1.1. The symmetry of functions

The symmetry of a function is defined with respect to its behavior for positive and negative values of its argument.

- For a *symmetric* function: $f(-x) = f(x)$. The function values at equal positive and negative distances from the origin are the same. Figure A.3 shows that the cosine function is symmetric.

- For an *antisymmetric* function: $f(-x) = -f(x)$. The function values at equal positive and negative distances from the origin are each others opposite. The sine function is antisymmetric.

For two functions $f_1(x)$ and $f_2(x)$ table A.2 shows the symmetry of the product function $f_1(x)f_2(x)$ and the quotient function $f_1(x)/f_2(x)$. The first line at the third column reads that if $f_1(x)$ is a symmetric function (S) and $f_2(x)$ is an antisymmetric function then the product function $f_1(x)f_2(x)$ is an antisymmetric function (A). As we can see from the table the relations with respect to symmetry for the quotient function equal the relations for the product functions. This is because the function $f_2(x)$ and $1/f_2(x)$ have the same symmetry. For example, the tangent function (see section A.2) is antisymmetric because it is the quotient of the antisymmetric sine function and the symmetric cosine function.

π is involved: the circumference of a circle equals π times its diameter, or 2π times its radius. The number π is approximately 3.1415...

Table A.2.: Symmetry of the product and division of two functions.

$f_1(x)$	$f_2(x)$	$f_1(x)f_2(x)$	$f_1(x)/f_2(x)$
S	A	A	A
S	S	S	S
A	S	A	A
A	A	S	S

Every function $f(x)$ can be split in a symmetric part $f_s(x)$ and an antisymmetric part $f_a(x)$ such that $f(x) = f_s(x) + f_a(x)$ by:

$$\begin{aligned} f_s(x) &= (f(x) + f(-x))/2 \\ f_a(x) &= (f(x) - f(-x))/2 \end{aligned}$$

You can try it for example with the function $f(x) = x + x^2$ which has the symmetric part x^2 and the antisymmetric part x.

A.1.2. The sine and cosine and frequency notation

In the previous section we have shown that sine and cosine are periodic functions with period 2π. To make more explicit how many periods we want to "travel", we could write $\sin(2\pi x)$ instead of $\sin\theta$. This notation makes more explicit that when x varies between 0 and 1, the argument of the sine varies between 0 and 2π. Especially in physics where the notion of frequency is coupled to sines and cosines, we want a notation where it is immediately obvious how many periods occur within a certain time interval.

Consider the following notation for a signal $s(t)$: $s(t) = \sin(2\pi f t)$ for $0 \leq t \leq 1$. This function represents one second of a tone with frequency f. If we want to know how this function behaves in this interval we have to know what the value of f is.

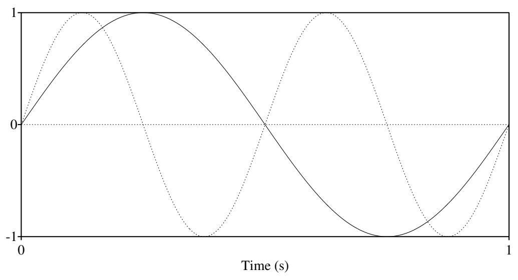


Figure A.4.: The function $sin(2\pi t)$, drawn with a solid line, and the function $\sin(2\pi 2t)$, drawn with a dotted line.

In the figure we have plotted with a solid line the function $s(t)$ for $f = 1$ and with a dotted line for $f = 2$. We see that the solid line traces exactly one period of the sine and the dotted line exactly two periods. The parameter f denotes how frequent a period occurs within a time interval of one second. Hence the term frequency for the parameter f. Frequency is expressed in hertz, abbreviated to Hz.

A sound $s(t)$ that varies in time like $s(t) = a \times sin(2\pi ft)$ is called a *pure tone*. In this formula a is called the *amplitude* and f is called the *frequency* of the tone. In the sequel we leave out the explicit multiplication sign and write $s(t) = a\sin(2\pi ft)$. A pure tone is an *elementary* sound. In chapter 4 on scripting, especially section 4.4, we explain how to create pure tones in Praat. What is important now is that if we increase the frequency f of a tone from a low value, say 100 Hz, to a higher value, say 1000 Hz, we can hear that the second sound is higher than the first. If we have a number of sounds that *only differ* in frequency, we are able to order these sounds on a scale from low to high. Pure tones are elementary because they are the only sounds that we can order on a scale from low to high *unambiguously*. The perceptual equivalent of the frequency of a pure tone is *pitch*, i.e. a pitch is something that is made in our heads. There is a *monotone relationship* between the frequency of a pure tone and pitch. A monotone relationship between pitch and frequency means that if the frequency increases the pitch increases. This does not necessarily mean that, for example, if we have two tones with frequencies f_1 and $f_2 = 2f_1$, the second pitch will be two times higher than the first one. The second pitch will be perceived as the higher pitch of the two.

When we vary the amplitude of a pure tone, and keep the frequency constant, we hear a difference in loudness. There is also a monotone relationship between the amplitude of a pure tone and its perceptual measure loudness. The larger the amplitude a the louder the sound.

A.1.3. The phase of a sine

We will first show that if we add a cosine and a sine with the same argument the resulting function can be written in an alternative way by using only a sine function. We start with $s(x) = a\cos(x) + b\sin(x)$, where initially we assume that a and b are some positive numbers. The function $s(x)$ is a *mixture* of a sine and a cosine, the outcome of the mixture is determined by the coefficients a and b. Two extreme cases are for $b = 0$, then $s(x)$ reduces to a cosine function $s(x) = a\cos(x)$, and for $a = 0$, then $s(x)$ reduces to the sine function $s(x) = b\sin(x)$. In figure A.5 we show $s(x)$ for three different mixtures (a, b). In the top row $a = 1$ and $b = 1/10$ and we show: on the left three periods of the function $\cos(x)$, in the middle three periods of the function $1/10\sin(x)$ and on the right the result of summing these two functions $s(x) = \cos(x) + 1/10\sin(x)$. In the middle row both a and b are equal to one and the function on the right is the sum of three periods of $\cos(x)$ and three periods of $\sin(x)$. For the bottom row $a = 1/10$ and $b = 1$.

A careful look at the functions in the right column shows that they all show *exactly three* periods. Each period is marked with a vertical dotted line and shows a complete sine function. Compared to the "normal" $\sin(x)$ function they are displaced and start with a value as if x was not zero at the start. Well, this is exactly how we can describe this function: as a displaced sine. We write $c\sin(x + \theta)$, where the displacement θ is called the *phase*, and c is the amplitude. As the figure makes clear, both c and θ have a relation with the coefficients a and b of the mixture.

We will now derive what this relation is. We start with the mixture

$$a\cos(x) + b\sin(x). \tag{A.1}$$

The trick we will use is to rewrite the a and the b. Let us assume first that a and b correspond to the sides of a right triangle as in figure A.1. From this triangle it follows that $\sin\theta = a/c$, $\cos\theta = b/c$ and $\tan\theta = b/a$. We rewrite the first two as $a = c\sin\theta$ and $b = c\cos\theta$ and the third as $\theta = \tan^{-1}(b/a) = \arctan(b/a)$. Since c is the hypotenuse of the triangle, its length is $\sqrt{a^2+b^2}$. We substitute these values for a and b in equation (A.1) to obtain

$$\begin{aligned} a\cos(x) + b\sin(x) &= c\sin\theta\cos(x) + c\cos\theta\sin(x) \\ &= c(\sin\theta\cos(x) + \cos\theta\sin(x)) \\ &= c\sin(x+\theta), \end{aligned} \tag{A.2}$$

where we used one of the "familiar" trigonometric relations between sines and cosines of different arguments. This equation says that any linear combination of a sine and a cosine results in a displaced sine function. Given a displaced sine function we can decompose it into a mixture of a sine and a cosine. If we start from the pair a and b we calculate $c = \sqrt{a^2+b^2}$ and $\theta = \arctan(b/a)$. If we start from c and θ we calculate $a = csin\theta$ and $b = c\cos\theta$.

These insights will be used in the next session where we will explore how to determine from a mixture the mixture coefficients a and b, or the alternative representation with c and θ.

A.1.4. Average value of products of sines and cosines

In this section we show a way to calculate the average value of products of sine and cosine functions. We need a way to calculate these averages because they turn up in many problems like for example Fourier analysis. We want to avoid the mathematics of integrals, instead we use the related concept of the average value of a function on an interval. The average values of the functions we use are easy to calculate: just sum the function values at "regularly spaced" x-values. To be able to calculate the averages of these compounds, we start with simple sines and cosines. In figure A.3 the sine and cosines functions are shown. Both functions have a period of 2π. It is easy to see that, loosely speaking, the average value of the sine curve in a stretch of one period equals zero. The sine curve in the interval from 0 to π is exactly equal but of opposite sign to the curve in the interval from π to 2π. It doesn't even matter where the period starts: for any stretch of length of one period, the average of the sine curve will be zero. Since the cosine curve equals a sine curve, but displaced over a distance of $\pi/2$, these results also apply to the cosine. As a consequence, *if an integer number of sines or cosines fit in an interval then the average values of these functions equal zero.* This result is important because the rest of this section will depend on it.

We will start with the investigation of the simple products as displayed in figure A.6. In this figure all functions are displayed in the interval from 0 to 2π. The first column shows one period of a sine or cosine function that is multiplied with the sine or cosine function from the second column. The third column shows the product function. For example in the middle row the first column shows one period of $\sin(x)$, the second column shows one period of $\cos(x)$ and the third shows the product $\sin(x)\cos(x)$. The third column in the first row shows the product of the sine function with itself, the $\sin^2(x)$ function. This is a periodic function but the period has halved and it shows two periods in the interval. This shows the familiar result of equation (A.32) that $\sin^2(x) = 1/2 - 1/2\cos(2x)$. From this representation it is easy to see that the average value of this function is 1/2 because the average of the $-1/2\cos(2x)$ term equals zero as we have shown before.

In the second row the third column displays the $\sin(x)\cos(x)$ function. Like in the previous row, this results in a periodic function with half the period. It expresses the familiar result of equation (A.33) that $\sin(x)\cos(x) = 1/2\sin(2x)$. The average value of this function is zero.

In the third row the last column graphically displays equation (A.34) which says $\cos^2(x) = 1/2 + 1/2\cos(2x)$. The average of this function equals 1/2 just like the average for the $\sin^2(x)$ function. These results show that the average of a $\sin(x)$ or $\cos(x)$ multiplied by itself is different from zero on the interval 0 to 2π.

We will now investigate how the average behaves for functions that are products of sines and/or cosines that are harmonically related, i.e. where an integer number of periods fit in the interval. Let us start with the product $p_{ss}(x) = \sin(mx)\sin(nx)$, where m and n are integer numbers. We will use equations (A.28) to rewrite the product in terms of sums.

$$\begin{aligned} p_{ss}(x) &= \sin(mx)\sin(nx) \\ &= (\cos(mx - nx) - \cos(mx + nx))/2 \\ &= 1/2\cos((m-n)x) - 1/2\cos((m+n)x). \end{aligned}$$

The equation above shows that if m is not equal to n, the average of $p_{ss}(x)$ is zero because it is the sum of two cosine functions each of which has average value zero. If m equals n, the situation is different and $p_{ss}(x) = 1/2 - 1/2\cos(2mx)$. The average of this function will be 1/2 and the result does not depend on the value of m. We conclude from this: on the interval $[0, 2\pi]$, the average of the product of two harmonically related sines is only different from zero when the sines are equal.

For the product of a harmonically related sine and cosine we use equation (A.30) to get

$$\begin{aligned} p_{sc}(x) &= \sin(mx)\cos(nx) \\ &= 1/2\sin((m-n)x) + 1/2\sin((m+n)x). \end{aligned}$$

The average of this equation is always zero. If m equals n the $\sin((m-n)x)$ term equals zero and the average of the other term equals zero too.

For the other possibility $p_{cc}(x) = \cos(mx)\cos(nx)$ we use equation (A.31), and the results are like the results obtained for two harmonically related sines: on the interval $[0, 2\pi]$, the average of the product of two harmonically related cosines is only different from zero when the cosines are equal.

The summary conclusion from these products is: *in the interval* $[0, 2\pi]$, *the average of products of harmonically related sines and/or cosines are only different from zero when both terms in the product are the same function.*

The results above can be used to show that when we have a mixture of a sine and a cosine function, then we can get the strength of each component by only determining specially formed averages. Suppose we have a function $s(x) = a\cos(x) + b\sin(x)$ in the interval from 0 to 2π, and the numbers a and b are unknown. How can we determine them? Let us multiply

$s(x)$ by the function $\cos(x)$. This results in $a\cos(x)\cos(x) + b\sin(x)\cos(x)$. If we calculate the average of this function the second term will not contribute because of the results obtained above. The first term will contribute the value $a/2$. Therefore the value for a will be twice the calculated average of $s(x)\cos(x)$. If we multiply $s(x)$ with a sine function, we obtain $a\cos(x)\sin(x) + b\sin(x)\sin(x)$ and now only the second term contributes to the average. The value of b is twice the calculated average of $s(x)\sin(x)$.

These results are so important that we will rephrase them again: if we have a function $s(x) = a\cos(x) + b\sin(x)$ that is a mixture of a cosine and a sine component whose strengths a and b are not known, we can calculate the strengths a and b by any of the following procedures:

1. Multiply $s(x)$ by the function $\cos(x)$ and determine the average value of the product function on the interval from 0 to 2π. The strength a will be two times this value. To determine b, we multiply $s(x)$ by the function $\sin(x)$ and determine the average value of this product function on the integral 0 to 2π. The value of the strength b is two times this average.

2. The alternative procedure starts at the alternative formulation of the mixture with the function $s(x) = c\sin(x + \theta)$. We multiply with $cos(x)$ and apply equation (A.30) we obtain

$$\begin{aligned} s(x)\cos(x) &= c\sin(x+\theta)\cos(x) \\ &= c/2(\sin(\theta) + \sin(2x+\theta)). \end{aligned}$$

   Now if we calculate the average of this function only the first term contributes and the result will be $c/2\sin(\theta)$. Multiplication of $s(x)$ with $\sin(x)$ and calculating the average results in a value that equals $c/2\cos(\theta)$. This can be shown by applying equation (A.28). The two averages can be used to solve for c and θ.

The results above are used in Fourier analysis where the individual frequency components are all harmonic frequencies. By multiplying with cosines and sines of the right frequency their strengths can be determined.

A.1.5. Fade-in and fade-out: the raised cosine window

When the amplitude at the start or the end of a sound changes abruptly, we often hear a click sound. Most of the time these click sounds can be avoided by selecting parts of the sound that start and end at zero crossings of the sound signal. Sometimes however, these clicks can not be avoided. For example if we want to create fixed duration stimuli for a listening experiment, it is almost impossible to guarantee that all these sound start and end nicely on zero crossings. We can arrange that the signal amplitude, instead of abruptly rising, slowly rises at the start of the sound. This is called “fade in”. By multiplying the first part of the sound with a function whose value gradually increases from zero to one over a predefined interval we can accomplish the fade in. A five millisecond rise time is sufficient to avoid clicks. The simplest function that accomplishes a linear rise time for a sound that starts at zero seconds is the function $x/0.005$. The following script modifies the first five milliseconds of a selected sound to fade in smoothly:

```
Formula: "if x <= 0.005 then self*x/0.005 else self fi"
```

To fade out the last five milliseconds of a sound the function needs to fall from one to zero within the interval. The following script modifies the last five milliseconds of a selected sound to fade out smoothly:

```
Formula: "if x >= xmax-0.005 then self*(xmax-x)/0.005 else self fi"
```

The disadvantage of the linear function defined above for the fading is that at the start and the end of the fading it is not continuous because the slope changes instantly at these points. We like the slope to change gradually at the start and end of the fade. Better functions to accomplish the fading can for example be based on a raised cosine. In the left plot of figure A.7 we show the first five milliseconds of the function $w_o(x) = (1 + \cos(2\pi 100x))/2$ with a solid line and of $w_i(x) = (1 - \cos(2\pi 100x))/2$ with a dotted line. The dotted line shows a curve smoothly rising from zero to one, while the solid line shows a smooth transition from one to zero. Moreover, the slopes of these curves start and end horizontally. This behavior makes these two function very suitable as a fade-in and a fade-out function, respectively. The following script fades in a selected sound:

```
Formula: "f x<=0.005 then self*(1-cos(2*pi*100*x))/2) else self fi"
```

To have the fade-in or the fade-out start at any defined point x_0 we have to translate these functions that show the desired behavior in the interval from 0 to 0.005, i.e. for $x_0 = 0$. For example, if we like to start the fade-in or the fade-out at say $x_0 = 0.13775$, the following script to fade in

```
x1 = x0+0.005
Formula: "if x>=x0 and x<=x1 then self*(1-cos(2*pi*100*x))/2 else self fi"
```

would produce the wrong result since the function $w_i(x)$ evaluated for values of x in the interval from x_0 to $x_0 + 0.005$ shows the values in the middle plot of figure A.7 indicated with a dotted line. The solid line shows the $w_o(x)$ function evaluated in the same interval. Both curves show that the fade functions used in this way do not produce the right results. There is an easy way out: arrange for the arguments of these functions that when they start at value $x = x_0$ they behave as if they start at $x = 0$. If we replace the x in both function definitions by $(x - x_0)$ then we have accomplished our goal: the evaluation for x running from x_0 to $x_0 + 0.005$ returns the same values as the evaluations for x between 0 and 0.005.

With the adaption for x_0, our fade-in and fade-out functions become $w(x) = (1-\cos(2\pi 100(x-x_0)))/2$ and $w_o(x) = (1 + \cos(2\pi 100(x - x_0)))/2$. The correct fade-in that reproduces the right plot in figure A.7 at time x_0 now reads

```
x1 = x0+0.005
Formula:
 ... "if x>=x0 and x<=x1 then self*(1-cos(2*pi*100*(x-x0)))/2 else self fi"
```

As one might guess, there is a relation between the 0.005 s duration of the fade and the number 100 that appears in the functions w_i and w_o. The duration of the fade corresponds exactly to one half period of the cosine in these functions. We then calculate a full period duration as 0.01 s. This corresponds to 100 periods in a second, i.e. a frequency of 100 Hz for the cosine. In the following script a noise signal of 50 ms duration is faded in a 10 ms duration interval and faded out with the same duration interval. The script shows the fade-in and the fade-out rather explicit:

```
xmax = 0.05
d = 0.01
f = 1 /(2*d)
Create Sound from formula: "noise", "Mono", 0, xmax, 44100, "0"
Formula: "randomUniform(-1,1)"
x0 = 0
x1 = x0+d
Formula:
 ... "if x>=x0 and x<=x1 then self*(1-cos(2*pi*f*(x-x0)))/2 else self fi"
x0 = xmax - d
x1 = x0+d
Formula:
 ... "if x>=x0 and x<=x1 then self*(1+cos(2*pi*f*(x-x0)))/2 else self fi"
```

In figure A.8 we see the result of applying a 10 ms fade-in at the start of a noise and a 10 ms fade-out at the end of the noise. The combined effect of the fade-in and the fade-out on this 30 ms long noise can also be visualized as the multiplication of the noise with a window signal. This window signal has value one everywhere except at the start and the end where the faders work. The middle pane shows the window signal, the right panel the result of multiplying the left noise with the middle window signal.

A.2. The tan function

The tangent function originally was defined as the quotient of the opposite side and the adjacent side of a right triangle (BC/AC in figure A.1). This turns out to be equal to the quotient of the sine and the cosine, i.e. $\tan\theta = \text{BC}/\text{AC} = (\text{BC}/\text{AB})(\text{AB}/\text{AC}) = (\text{BC}/\text{AB})/(\text{AC}/\text{AB}) = \sin\theta/\cos\theta$. In figure A.1 this is equal to the length of BE, i.e. the length of the line from the intersection of the *tangent* line in B with the horizontal axis. If the length of AC becomes smaller and smaller the tan function becomes larger and larger. The tan function is unbounded.

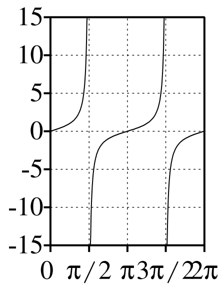


Figure A.9.: The $\tan(x)$ function for $0 \leq x \leq 2\pi$.

In figure A.9 we show the $\tan(x)$ function for values of x between 0 and 2π and for amplitudes limited between $-15$and $+15$. At $x = \pi/2$ and $x = 3\pi/2$ there are two discontinuities:

on the left side of these x values the function goes to positive infinity and on the right side the function comes from minus infinity. Closer inspection reveals that the part between 0 and π and the part between π and 2π are exactly equal. In contrast to the functions $\sin(x)$ en $\cos(x)$ that have periodicity 2π, the $\tan(x)$ has periodicity π.

A.3. The sinc(x) function

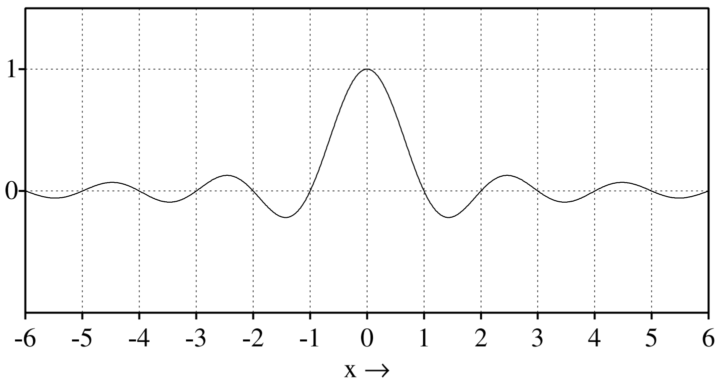


Figure A.10.: The function $\frac{\sin(\pi x)}{\pi x}$.

The sinc function is important in digital signal analysis because it is used among others for interpolation of band-limited signals. For example, if we want to represent a sound with a different sampling frequency, the new sample values have to be interpolated from the old values and the sinc function is used to do this.

The sinc function, where sinc stands for *sinus cardinalis*, is defined as

$$\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}. \tag{A.3}$$

In figure A.10 we show this function. The function is symmetric because it is the product of two antisymmetric functions, the sine function $\sin(\pi x)$, and the slowly decaying function $1/(\pi x)$. Because $\sin(\pi x)$ equals zero if x has integer value, the sinc function equals zero at these points too. A shorthand notation for the former notion is

$$sin(k\pi) = 0, \quad for \quad k = 0, \pm 1, \pm 2, \ldots$$

At $x = 0$ both the numerator and the denominator in equation A.3 are zero and this signals a potential problem because the result of such a division is not defined. However, it turns out that the sinc function behaves nicely at $x = 0$ and is equal to one. The reason for the nice behavior of equation A.3 for $x = 0$ stems from the fact that for very small values of x, the $\sin(x)$ is approximately equal to its argument x, i.e. $\sin(x) \approx x$ for small x. The smaller x is, the better this approximation will be. The consequence is that near $x = 0$ the functions $f(x) = x$ and $f(x) = \sin(x)$ become nearly indistinguishable. Therefore, for very small x we may substitute for the $\sin(\pi x)$ function its argument πx and we may write $\text{sinc}(x) \approx \pi x/\pi x = 1$.

The square of this function turns up in power spectra where amplitudes are expressed in dB's. In figure A.11 we show the function $10\log(\sin(\pi x)/(\pi x))^2$. It shows the typical main lobe and side lobes. The main lobe of this function is at 0 dB for $x = 0$, since the argument of the log equals one there. The amplitude of the first side lobe is indicated in the figure with a horizontal dotted line at −13.3 dB below the amplitude of the main lobe. Because the amplitude of the sync function decreases as $1/x$, the amplitudes of the side lobes in the squared function decrease at a rate of −6 dB/oct.

A.4. The log function and the decibel

A simplified definition for the logarithm of a given number is the power to which you need to raise 10 in order to get the number. For example, the logarithm of 1000 is 3, because 10 raised to a power of 3 is 1000:

$$y = \log x \Longleftrightarrow x = 10^y \tag{A.4}$$

The logarithm function is only defined if its argument x is a positive number. If $x \geq 1$ the logarithm acts as a compressor as can be seen in the following figure.

We see that if x varies from 1 to 10, y varies from 0 to 1. If x varies from 10 to 100, y varies from 1 to 2. If x varies from 100 to 1000, y varies from 2 to 3. The logarithm compresses the interval 1-10, 10-100 and 100-1000 into intervals of the same length 1.

A.4.1. Some rules for logarithms

To make calculations with logarithms, we first recapitulate some elementary rules of arithmetic. We will not be mathematically rigorous in the formulations but try to state these rules in popular terms. The first rule we need concerns multiplication: in multiplication with numbers, the order of the terms is not important, i.e. $a \times b = b \times a$. For example $2 \times 3 = 3 \times 2 = 6$. Another elementary rules of arithmetic states that if we calculate a quotient like a/b, with numerator a and denominator b, this expression can also be calculated as the product of the numerator with the inverse of the denominator, i.e. $a/b = a \times (1/b)$. Some examples: $4/2 = 4 \times (1/2) = 2$ and $(7/4)/(7/8) = (7/4) \times (8/7) = 2$. With these rules we can master logarithms, giving the following rules for logarithms.

- $\log 1 = 0$, the logarithm of the number one equals zero.
- $\log 10 = 1$, the logarithm of 10 equals 1.
- $\log 2 \approx 0.3$, the logarithm of the number two is approximately equal to the number 0.3. This is probably the most often used logarithm in this book.
- $\log(a \times b) = \log a + \log b$, the multiplication rule: the logarithm of a product of two terms equals the sum of the logarithms of each term. Some examples: $\log 4 = \log(2 \times$

2) $= \log 2 + \log 2 \approx 0.3 + 0.3 = 0.6$, $\log 100 = \log(10\times 10) = \log 10 + \log 10 = 1+1 = 2$. A generalization is that the logarithm of a product of any number of terms equals the sum of the logarithms of the individual terms. For example , if we need the logarithm of a product of three terms, $\log(a \times b \times c)$, we can rewrite the product of the three terms $a \times b \times c$ as a product of two terms $(a \times b)$ and c as $(a \times b) \times c$. The logarithm of this two-term expression can then be rewritten as the sum of two logarithms $\log(a \times b)$ and $\log c$. In the final step we simplify the $\log(a \times b)$ expression with the rule above. The following calculation summarizes:

$$\log(a \times b \times c) = \log((a \times b) \times c) = \log(a \times b) + \log c = \log a + \log b + \log c$$

In a multiplication, the order of the terms is not important, i.e. $a \times b \times c = (a \times b) \times c = a \times (b \times c) = b \times (a \times c)$. We could have started with any of these terms to find the same result.
Another example: $\log 20 = \log(2 \times 10) = \log 2 + \log 10 \approx 0.3 + 1 = 1.3$.
From the multiplication rule we can also deduce that the logarithm of the inverse of a number equals the negative of the logarithm of that number, i.e. $\log(1/b) = -\log b$. To show this, we start with the fact that if we divide a number by itself the result is one, i.e. $b/b = 1$. If we take the logarithm of both sides we get $\log(b/b) = \log 1 = 0$. Now we use the quotient rule and rewrite the first part as $\log(b/b) = \log(b \times (1/b)) = \log b + \log(1/b)$. The sum of the last two terms must equal zero and that can only happen if the two terms cancel each other. In other words: $\log(1/b) = -\log b$.

- $\log(a/b) = \log a - \log b$, the quotient rule: the logarithm of a quotient is the logarithm of the numerator minus the logarithm of the denominator. This rule follows from the multiplication rule because we know that since $a/b = a(1/b)$, we can rewrite $\log(a/b)$ as $\log(a \times (1/b))$ which can be rewritten as $\log a + \log(1/b) = \log a - \log b$. Some examples: $\log(1/10) = \log 1 - \log 10 = 0 - 1 = -1$, $\log 0.5 = \log(1/2) = \log 1 - \log 2 \approx 0 - 0.3 = -0.3$, $\log 5 = \log(10/2) = \log 10 - \log 2 \approx 1 - 0.3 = 0.7$.

- $\log a^x = x \log a$, the power rule: the logarithm of a number raised to a power equals the exponent times the logarithm of this number. This rule is also an extension of the multiplication rule. For example $\log(2^3) = \log(2\times 2\times 2) = \log 2 + \log 2 + \log 2 = 3\log 2$. Examples: $\log 10000000000000 = \log 10^{13} = 13$, $\log 16 = \log 2^4 = 4\log 2 \approx 4 \times 0.3 = 1.2$.

A.4.2. The decibel (dB)

The decibel is a logarithmic unit of measurement that expresses the magnitude of a physical quantity (for example power or intensity) relative to a specified or implied reference level. Since it expresses a ratio of two quantities, it is a dimensionless unit:

$$\text{Power(dB)} = 10 \log(P/P_{ref}) \tag{A.5}$$

Because power in a signal is related to the square of the amplitude, dB's can also be stated in terms of amplitude:

$$\text{Amplitude(dB)} = 20\log(A/A_{ref}) \tag{A.6}$$

Human perception experiments have shown (refs??) the smallest just noticeable difference (jnd) in sound intensity is approximately 1 dB. This jnd is the smallest difference in intensity that is detectable by a human being. The jnd is a statistical, rather than an exact quantity: from trial to trial, the difference that a given person notices will vary somewhat, and it is therefore necessary to conduct many trials in order to determine the threshold. The jnd usually reported is the difference that a person notices on 50% of trials. We can now simply calculate how much two tones have to differ in *amplitude* to have a 1 dB difference in *intensity*. If we have two sounds with intensities I_1 and I_2 their intensity difference, call it y, in dB's is $10\log(I_1/I_2)$. We want to use amplitudes instead of intensities so we use the scale factor 20 for amplitudes to get $y = 20\log(A_1/A_2)$. In the last equation we divide both sides by 20 to get $\log(A_1/A_2) = y/20$, from which we get by using equation (A.4) $A_1/A_2 = 10^{y/20}$. For two pure tones, where the first is 1 dB less intense than the first we calculate $A_1/A_2 = 10^{-1/20} \approx 0.89$ and for two tones where the first is 1 dB louder we calculate $A_1/A_2 = 10^{+1/20} \approx 1.12$.

We can create a series of 1000 Hz pure tones that differ by a specified amount of dB's with the following command:

```
Create Sound from formula: "s", 1, 0, 0.05, 44100, "scale*0.5*sin(2*pi*1000*x)"
```

The following table (A.3) gives some values for *scale*. The table shows that the following

Table A.3.: The scale factor for intensity differences in dB's

dB's	formula	scale
0.0	$10^{0/20}$	1.00
+0.5	$10^{0.5/20}$	1.06
+1.0	$10^{1/20}$	1.12
−0.5	$10^{-0.5/20}$	0.94
−1.0	$10^{-1/20}$	0.89
−3.0	$10^{-3/20}$	0.71
−6.0	$10^{-6/20}$	0.50

two commands result in two tones with a frequency of 1000 Hz that differ by 0.5 dB in intensity.

```
Create Sound from formula: "s", "Mono", 0, 0.05, 44100,
    ... "0.5*sin(2*pi*1000*x)"
Create Sound from formula: "s", "Mono", 0, 0.05, 44100,
    ... "0.94*0.5*sin(2*pi*1000*x)"
```

A.4.3. Other logarithms

We now have worked with the simplified definition for the logarithm of a given number as the power to which you need to raise 10 in order to get the number. We repeat once again equation

(A.4): $y = \log x \Longleftrightarrow x = 10^y$. As mathematicians like to generalize, they ask: what is so special about the number 10? Why can't we define a logarithm with any positive number b instead of 10? Of course, we can. We define the logarithm with base b as

$$y = {}^{b}\log x \Longleftrightarrow x = b^y \tag{A.7}$$

In the notation the base b is made explicit. If the base $b = 10$ and we write $\log x$ this base is implicit. The multiplication rule, the quotient rule and the power rule apply to all logarithms, and do not depend on the base b.

In working with the computer we often encounter the base 2 logarithm: the power needed to raise 2 in order to get the number:

$$y = {}^{2}\log x \Longleftrightarrow x = 2^y \tag{A.8}$$

Some numeric examples for base 2: ${}^{2}\log 1 = 0$, ${}^{2}\log 2 = 1$, ${}^{2}\log 4 = {}^{2}\log 2^2 = 2$, ${}^{2}\log 4 = {}^{2}\log(2 \times 2) = {}^{2}\log 2 + {}^{2}\log 2 = 1 + 1 = 2$, ${}^{2}\log 8 = {}^{2}\log 2^3 = 3, \ldots, {}^{2}\log 2^n = n$, ${}^{2}\log \frac{1}{2} = -1$, ${}^{2}\log \frac{1}{8} = -3$.

Another base that deserves special attention is the one with base e.[2] The logarithm with this base has a special *notation* and a special *name*. We write "the *natural logarithm* of x is y" as $\ln x = y$ instead of ${}^{e}\log x = y$:

$$y = \ln x \Longleftrightarrow x = e^y. \tag{A.9}$$

A.5. The exponential function

There are two notations for this function, $exp(x)$ or e^x, which both signify the *same* function: the number e raised to the power x. In figure A.13 the solid line shows the function $\exp(x)$ when x varies between -3 and $+3$. For x values smaller than zero e^x *decreases* very rapidly; for values greater than zero the exponential function *increases* very rapidly. The value for $\exp(x)$ increases without bound if x increases. Table A.4 show the values of the functions $\exp(x)$ and $\exp(-x)$ for some values of its argument x. The table shows that for positive

[2]Like the number π, the number e is an irrational number. Jacob Bernoulli discovered the number while working on the following financial problem of compound interest. Suppose you start with a sum of euro 1 and you receive an interest of p percent per period. To calculate how much money you own after one period, you can do the following. The interest paid equals the start sum multiplied by $p/100$. You add this value to the money you started with and now you know how much you own. If you had multiplied your start sum with $(1 + p/100)$ you would have obtained the same result: it says that the money you have after any interest period can be obtained from the money you had at the start of that period multiplied by $(1 + p/100)$. If you start with 1 euro then after the first interest period you will own $(1 + p/100)$. After the second interest period you will own $(1 + p/100)$ times what you owned at the start of the second interest period, i.e. $(1 + p/100) \times (1 + p/100) = (1 + p/100)^2$. After interest period three you will own $(1 + p/100)$ times what you owned at the start of the third interest period, i.e. $(1 + p/100) \times (1 + p/100)^2 = (1 + p/100)^3$. After n periods you own $(1 + p/100)^n$. Now we start calculating with real numbers. Start with 1 euro and an interest rate of 100%. If the interest is paid yearly then after one year you own $1 \times (1 + 100/100) = 2$ euros. If you had received the interest half-yearly, i.e. in two 50% terms, then after a year you would have owned $1 \times (1 + 50/100)^2 = (1 + 1/2)^2 = 2.25$. After a year, receiving interest quarterly at 25%, you would own $1 \times (1 + 25/100)^4 = (1 + 1/4)^4 \approx 2.44$. We continue: monthly interest at 8.25% after one year would yield $1 \times (1 + 8.25/100)^{12} = (1 + 1/12)^{12} \approx 2.61$. Interest periods of one week would result in $1 \times (1 + 1/52)^{52} \approx 2.69$. Daily interest results in $1 \times (1 + 1/365)^{365} \approx 2.714$. Bernoulli asked what number would result if the interest period went infinitely small, i.e. what is the value of $(1 + 1/n)^n$ for n going to infinity. Well this number is e.

Table A.4.: Approximate values for the exponential functions e^x and e^{-x}.

x	e^x	e^{-x}
0	1	1
1	2.71828	0.36788
2	7.38906	0.36788
3	20.0855	0.04979
4	54.5982	0.01832
5	148.413	0.00674
...		
10	22026.5	0.00005
...		
20	$4.8\ 10^8$	$2.1\ 10^{-9}$
...		
100	$2.7\ 10^{43}$	$3.7\ 10^{-44}$

numbers x the size of $\exp(x)$ grows very fast indeed. The exponential rise behavior of the function for x greater than zero is mirrored by the exponential decline in the function $s(t) = \exp(-x)$, for values of x greater than zero, i.e. for arguments smaller than zero. If $x = 0$ both the exponential rising and declining functions are equal to one.

A.6. The damped sinusoid

A damped sinusoid is used in speech processing where we use it to model formants as a function of time. The formula of a damped sinusoid is

$$s(t) = e^{-\alpha t}\sin(2\pi F t). \tag{A.10}$$

Here the parameter α is called the damping constant and the parameter F is called the frequency. Another possible way to write this function is by using the damping time τ as

$$s(t) = e^{-t/\tau} sin(2\pi F t). \tag{A.11}$$

The damping time τ is the time the amplitude of the sine wave falls by a factor of e ($\approx$2.71828). Another description is in terms of formant frequency and formant bandwidth:

$$s(t) = e^{-\pi B t}\sin(2\pi F t). \tag{A.12}$$

Now F and B denote the formant frequency and the formant bandwidth respectively. Each formant is modeled as a tone, the $\sin(2\pi F t)$ part, whose amplitude falls off exponentially, the $e^{-\pi B t}$ part. The frequency of the sine is called the formant frequency and the damping, or fall-off, is related to bandwidth. For example, the time function of a formant with frequency 500 Hz and bandwidth 50 Hz is $s(t) = \exp(-\pi 50 t)\sin(2\pi 500 t)$. What does this function look like? The following script creates figure A.14 where we see the first 100 ms of the formant.

```
Create Sound from formula: "f500", "Mono", 0, 0.01, 44100,
    ... "exp(-pi*50*x)*sin(2*pi*500*x)"
Solid line
Draw: 0, 0.01, -1, 1, "yes", "Curve"
Dotted line
Draw function: 0, 0.01, 1000, "exp(-pi*50*x)"
One mark right: exp(-pi*50*0.01), 1, 1, 0, ""
```

the first line in the script creates the formant as a sound from a formula. Its duration was chosen as 0.01 s, i.e. 100 ms. In the fourth line we draw the first 0.01 s of the formant. The sixth line draws the exponentially decaying part of the formant with a dotted line. Finally we draw a marker with the value of the exponential function $\exp(-\pi 50 t)$ at time 0.01 at the right of the figure.

A.7. The 1/x function

The function $p(x) = 1/x$ has the property that the size of the area under the curve between the positive values $x = a$ and $x = b$ equals the size of the area between two other values $x = c$ and $x = d$, as long as $b/a = d/c$.[3] For example, the size of the shaded area between $x = 1$ and $x = 2$ in figure A.15 equals the size of the shaded area between $x = 4$ and $x = 8$. This function is important for the following reason. Suppose that the power spectrum $P(f)$ of a signal has a $1/f$ shape. We can write this as $P(f) = a/f$, where the number a is is some scale factor whose size is not important here. The plot of power versus frequency would look like figure A.15.

When we divide the frequency axis in intervals, say in octave frequency bands, and we let the first band start at 100 Hz and end at 200 Hz, notation [100-200), the next frequency bands will be [200-400), [400-800), [800-1600), et cetera. Because of the $1/f$ function, all these frequency bands contain the same amount of power. We know that on the basilar membrane the frequency to place mapping follows approximately a logarithmic curve. This means that on the basilar membrane the interval from 100 to 200 Hz has the same length as the interval from 200 to 400 Hz, and the same length as the interval from 400 to 800 Hz, et cetera. Therefore, a $1/f$ power spectrum spreads the power evenly over the basilar membrane, i.e. the *equal power per frequency band* translates to *equal power per unit of length*. This is exactly what makes the $1/f$ power spectrum so special.

The power spectrum of a signal is normally not shown on a linear scale but on a logarithmic (dB) scale. The left plot in figure A.16 shows the power spectrum on a dB scale for the five octaves between 100 Hz and 3200 Hz. The number a was chosen to result in a 0 dB power at 100 Hz.[4] The power of the $1/f$ power spectrum in dB's then equals $-10 \log f$. From this formula we deduce that for two frequencies f_1 and $f_2 = 2f_1$ that differ by an octave, the spectrum falls of as $-10 \log(f_2/f_1) = -10 \log 2 \approx -3$ dB. One says, a $1/f$ *power* spectrum falls off as minus three dB per octave: -3 dB/oct. If f_1 and f_2 differ by a factor 10, one says

[3]The size of the area between $x = a$ and $x = b$, where $a < b$, is given by the value $A = \ln(b/a)$. If we choose another interval with a starting point at $x = c$, and we then choose the endpoint d, not at random, but at $d = \frac{b}{a}c$, the new area size is $A' = \ln(d/c) = \ln((\frac{b}{a}c)/c) = \ln(b/a) = A$.

[4]Because the power in dB's is $10 \log P(f)/P_{ref}$. With $P(f) = a/x$, this results in $10 \log((a/x)/P_{ref}) = -10 \log x + 10 \log(a/P_{ref})$. If we choose $a = P_{ref}$, the second term equals zero and we are left with $-10 \log x$.

the spectrum falls off as -10 dB/decade. In the left figure we can check that the five octaves result in a -15 dB drop.

Because the power P of a tone and the amplitude A of a tone are related as $P \sim A^2$, amplitude is related to power as $A \sim \sqrt{P}$. This means that if the power spectrum falls of as $1/f$, the amplitude spectrum falls off as $1/\sqrt{f}$. The amplitude spectrum in dB's then equals $-20 \log \sqrt{f} = -10 \log x$.

A.8. Division and modulo

If we divide two numbers like 5 and 3 and we are only interested in its integer result we could do this in several ways:

- floor(5/3) = 1, round down, highest integer not larger than 5/3
- ceiling(5/3) = 2, smallest integer not smaller than 5/3
- round(5/3) = 2, nearest integer, where round(1.5)=2

The div integer division operator behaves like the floor function, i.e. 5 div 3 = floor (5/3). Normally 5/3 would result in 1.666...

If we are interested in the remainder after division we can use the modulo (mod) operator. Some examples:
5 mod 3 = 2, 1 mod 3 = 1, 2 mod 3 = 2, 3 mod 3 = 0, 4 mod 3 = 1, 5 mod 3 = 2, 6 mod 3 = 0, 1 mod 2 = 1, 2 mod 2 = 0, 3 mod 2 = 1, 4 mod 2 = 0, 100 mod 50 = 0, 99 mod 50 = 49.
Note that modulo 2 tests whether a number is even or odd (uneven).

An example where we use the mod and div functions is in the stimuli preparation for the following experiment. Suppose we want to investigate the perceptual restoration of time-reversed speech, like Saberi and Perrot [1999] have done. We start with recording a number of sentences and making them available in a digital form in our computer. To time-reverse all the intervals in a sound, i.e. to reverse the order of all the samples in each interval, we proceed as follows. We choose the length of the interval beforehand. We make a new sound that has exactly the same duration and number of samples as the originating sound (this may be a perfect copy of the original). For every sample in the newly created sound we follow the same procedure: we determine in which interval it is located, we then determine its relative position *from the start* of the interval and finally we copy from the original sound the sample that has the same relative position but now *with respect to the end* of this interval. This is where mod and div come in handy: if you divide the current position of a sample by the interval length and round down to an integer number then you know in what interval this sample lies. This is exactly what the div operator does. The upper and lower limit of the interval can now simply be found: multiply the interval number with the interval length and you get the lower limit, add one interval length to obtain the upper limit. In figure A.17 the duration of the original sound (O) and the new sound (N) have been visualized. The current position is x. The lower and upper interval limits are indicated with small vertical marks on the lines. The relative position r of x within the interval can be calculated by subtracting from x the lower interval limit. This is exactly what the mod operator can calculate. If we know r we can do the actual time-reversal: this is indicated with the arrow, it shows that the value at relative position *r* from the end of the interval in the original sound is copied to the current position x. After doing so for all samples, we have completely destroyed the local spectro-temporal fine structure in each interval, but the global structure larger than the interval has been preserved.

The script on the current page has in lines 10-12 the code to time-reverse a sound.[5] This code looks complicated because it has to work for all possible values of the start time, xmin, and the end time, xmax, of a sound (the only condition is that xmin is smaller than xmax). For example the start time could be 10.12 and the end time 11.45 and the script should still behave correctly, i.e. for a chosen interval of 0.17 s duration, it should time reverse the first 0.17 s of this sound, then the next 0.17 s, etc.

Script A.1 Time reverse intervals.

```
1  form Time reverse intervals
2    positive Interval_(s) 0.17
3  endform
4
5  Create Sound from formula: "test", "Mono", 0, 0.777, 44100, "x"
6  s = selected ("Sound")
7  ims = 1000*interval
8  sc = Copy: "_"+ string$(ims)
9
10 Formula: "if ((((x-xmin) div interval)+1)*interval) <= (xmax-xmin) then
11 ... Object_'s'(xmin+((((x-xmin) div interval)+1)*interval -(x-xmin) mod interval))
12 ... else Object_'s'(xmin+((xmax-xmin) -(x-xmin) mod interval)) endif"
13
14 # Visual check
15 Erase all
16 Select outer viewport: 0, 6, 0, 3.5
17 selectObject: s
18 Solid line
19 Draw: 0, 0, 0, 1, "yes", "Curve"
20 selectObject: sc
21 Dotted line
22 Draw: 0, 0, 0, 1, "yes", "Curve"
23 Marks bottom every: 1, 0.17, "yes", "yes", "no"
```

If you run the script a form pops up in which you fill out a field labeled “Interval (s)”. After pushing the OK-button, a test signal is time-reversed and figure A.18 shows up in the Picture Window. This figure shows with a solid line the amplitude of the test signal and with a dotted line its time-reversed version for an interval duration of 0.17 seconds. As we can see, from the dotted line, time reversion was successful.[6]

Let us step slowly through this example. The first three lines define the form that pops up. In line 5, a sound named “test” with a duration of 0.777 seconds is created with a linearly rising amplitude (the x in the formula part guarantees this). This is a good sound to test the

[5] A script is text that consists of Praat commands. If you *run* the script, the commands are executed. In appendix 4 you will find more information about scripting Praat.

[6] We have deliberately chosen an interval duration that is not an integer divisor of the test signal’s length to test the correctness of the time reversal code for the last part of the sound.

time-reversal algorithm in lines 10, 11 and 12. To be able to refer to the name of this sound in a more general way, we assign the name of the created sound to a variable s$ in line 6. Because the code is simpler and because we like to keep the original signal intact, we create a copy of the original sound in line 8. To give the new sound a meaningful name we copy the name from the original sound and append the interval duration, rounded to an integer number of milliseconds to it. Line 7 assigns the number of milliseconds to a variable "ims" while the 'ims:0' part in the next line substitutes this number rounded down to an integer. Therefore, if a 0.17 s interval duration is chosen, the newly created sound will be named "test_170". All the magic of this script is in the next three lines. In fact, it is only one line that, for reasons of readability, has been laid out into three lines (in a Praat script a line that starts with three dots signals a continuation of the previous line). This powerful one-liner starts with `Formula...` which always implies that the code in the rest of the line is interpreted for each sample in the sound afresh. See section 4.7.1.2 for more info on this command. Line 11 is the meat of the formula, line 10 tests if the duration of the sound is an integer multiple of the chosen interval length and if not then line 12 will treat the last incomplete interval of the sound. The value of x is evaluated to a number between the start time of the sound, xmin, and the end time, xmax. In the formula the "div" operator and the "mod" operator are used in specific ways: $(x-xmin)\ \mathrm{div}\ \ interval$ and $(x-xmin)\ \mathrm{mod}\ interval$. The former equation essentially counts in which interval the x value lies. If we multiply this number with the duration of an interval we arrive at the starting position of the interval. First adding the number 1, before the multiplication, returns the end position of the interval. The latter equation calculates the relative position of x in the interval. In line 11 a sound sample with a certain offset from the *end position* of an interval in the originating sound is copied to a same offset but now with respect to the *start position* of the interval in the new sound.

To use this script on your own Sounds, you modify the script as follows: remove line 5, where the test sound is created, and remove the last part starting at line 14. The resulting script will time reverse the intervals in any sound selected in Praat's Object window. If the interval length is chosen larger than the duration of the sound, the sound will be completely time reversed! Applying the script again, now on the time-reversed sound, will result in another sound that is the time-reversed version of the time-reversed version, i.e. a sound that is identical to the sound you started with.

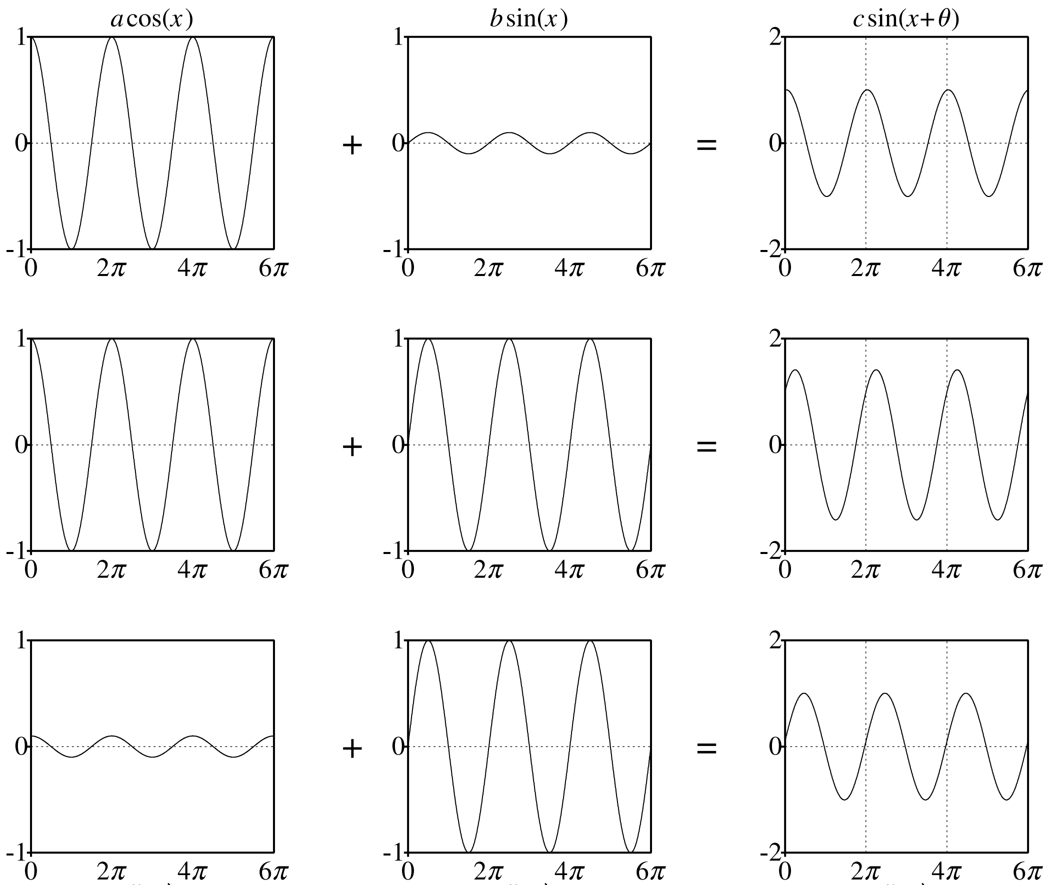


Figure A.5.: The mixture $a\cos(x) + b\sin(x)$. From top row to bottom row the values for the coefficients (a, b) are $(1, 1/10)$, $(1, 1)$ and $(1/10, 1)$, respectively. For each row the function at the right is the result of adding the two functions on the left.

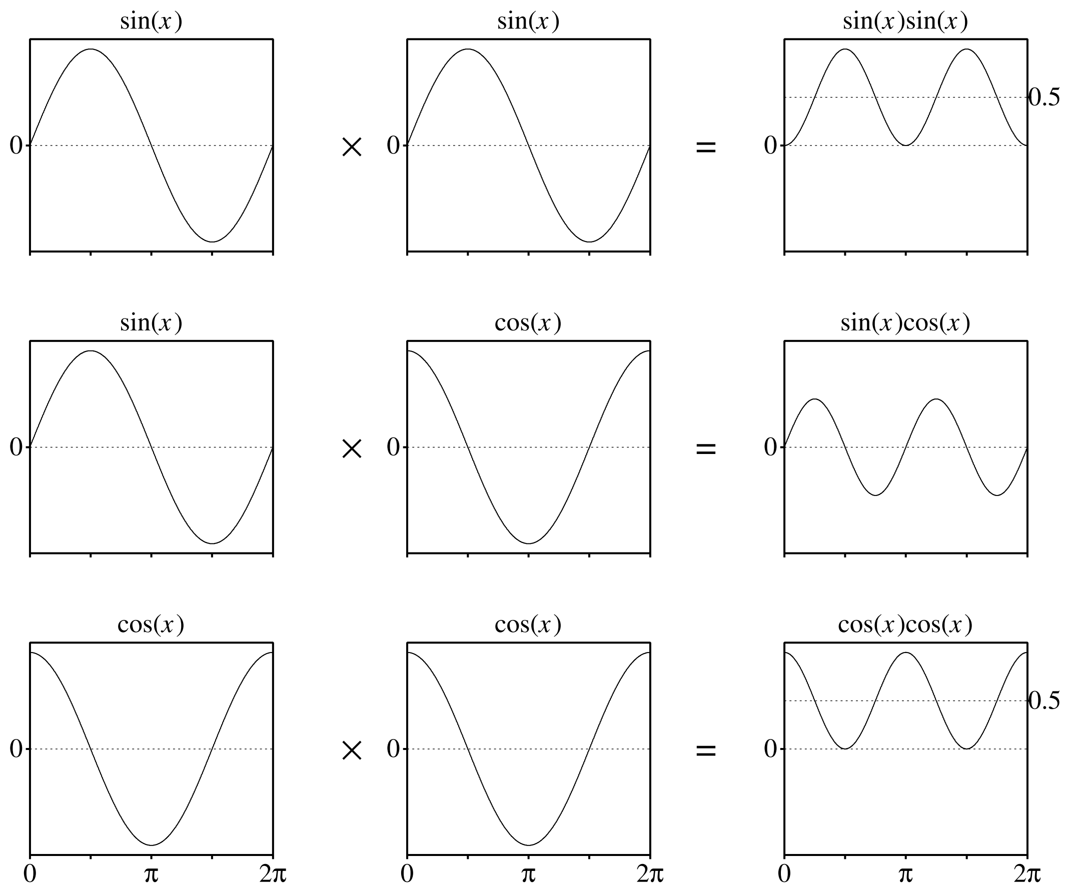


Figure A.6.: Products of $\sin(x)$ and $\cos(x)$.

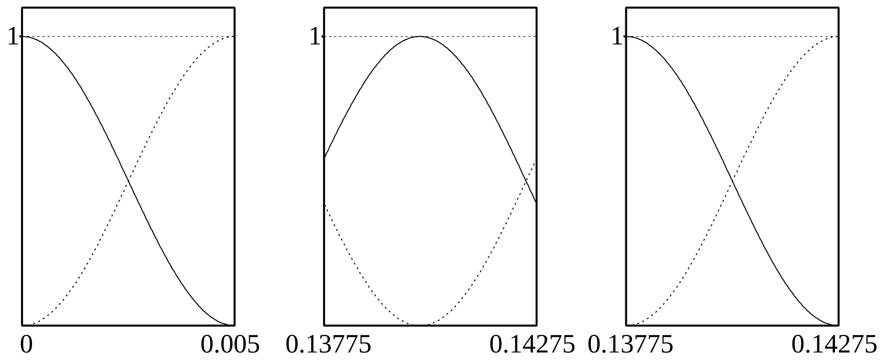


Figure A.7.: The raised cosine windows.

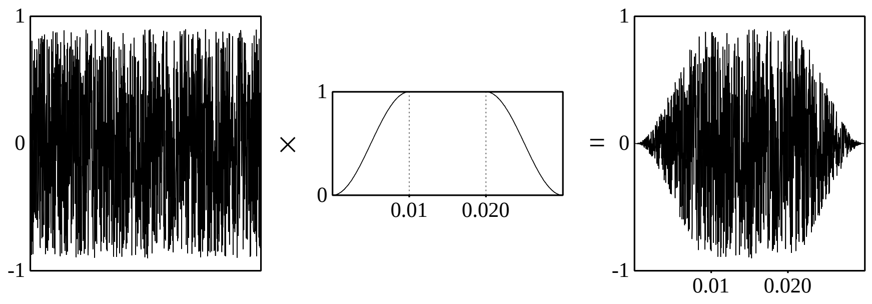


Figure A.8.: Applying a 10 ms fade-in and fade-out to a noise signal.

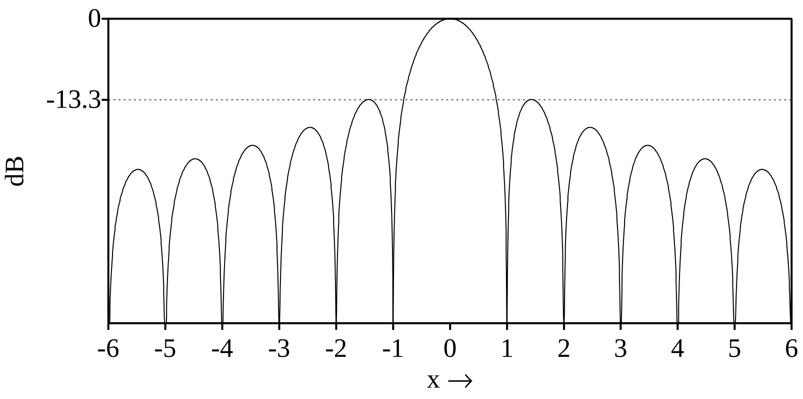


Figure A.11.: The square of the sinc(x) function on a decibel scale.

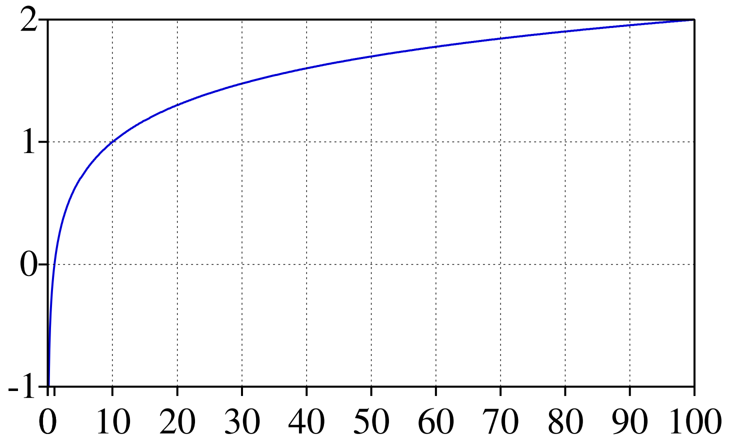


Figure A.12.: The function $\log x$.

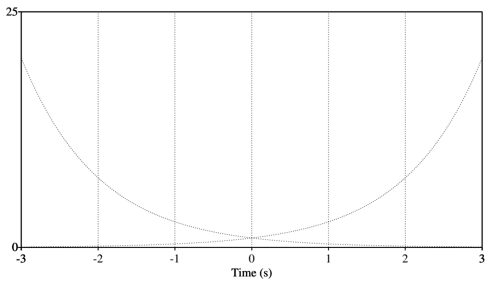


Figure A.13.: The exponential functions $\exp(x)$, drawn with a solid line, and $\exp(-x)$, drawn with a dotted line.

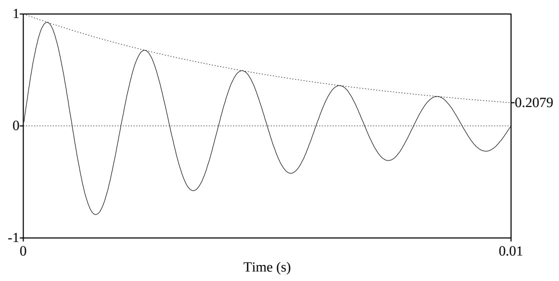


Figure A.14.: A 500 Hz formant with a bandwidth of 50 Hz drawn with a solid line, showing the exponentially declining part with a dotted line.

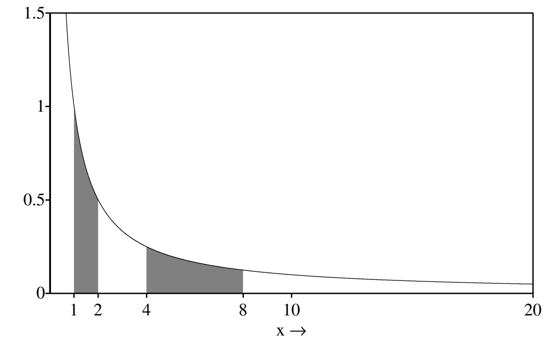


Figure A.15.: The function $1/x$.

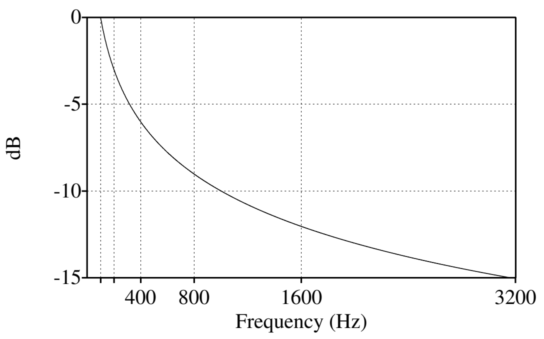


Figure A.16.: The $1/f$ power spectrum.

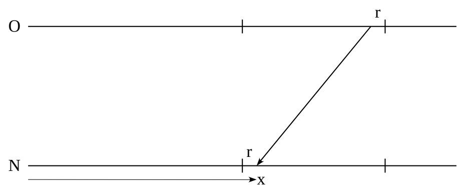


Figure A.17.: Time-reverse a sound: copying of a sound value, located a distance *r* from the end of the interval in the original sound (O), to a position at distance *r* from the start of the interval in the new sound (N).

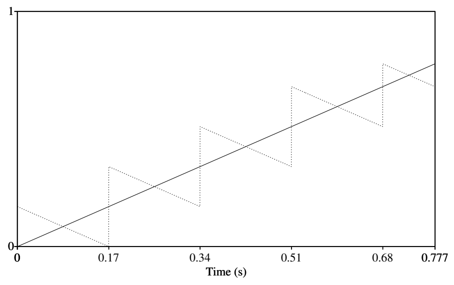


Figure A.18.: The result of the time-reversal script.

A.9. Integration of sampled functions

The object of integration is to determine the "area" underneath a curve, where we note that area can have a positive or a negative sign. Given a function $f(t)$ the area under the curve between a and b is indicated as follows

$$Area = \int_a^b f(t)dt. \tag{A.13}$$

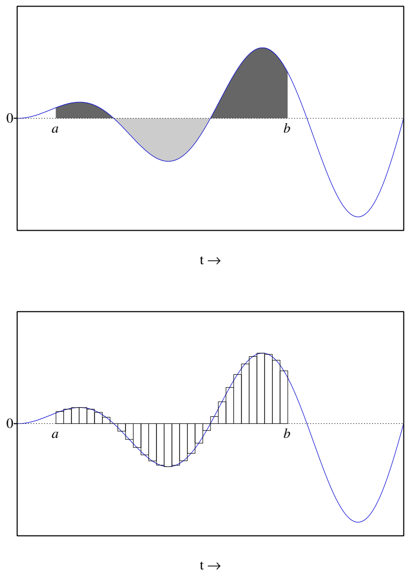


Figure A.19.: Integrating a function from $t = a$ to $t = b$ is the determination of the areas above and below the horizontal axis. The top panel shows the area between the function curve and the horizontal axis with shades of grey. The bottom panel shows an approximation of these areas with a sum of rectangular areas.

We can determine this area with a mathematical technique called *integration*. In figure A.19 we show an example. If the function of equation (A.13) is given by the blue curve then the result of this equation is equivalent to the determination of the area enclosed by the horizontal axis and the function on the interval between the points a and b. The areas above and below the horizontal axis are shown with different shades of grey. For the calculation of the integral the area below the horizontal axis is subtracted from the area above this axis. We will not go into the mathematical techniques that are available to solve equations like (A.13) for different functions $f(t)$, but instead show a simple numerical approximation of this integral for sampled signals or sounds. The bottom panel shows the sampled representation of the function in the top panel, however instead of a representation of the sample values as speckles or poles it shows each sample value as a rectangle. The height of the rectangle equals the amplitude and the width equals the sampling time Δt. The area of a rectangle equals its heigth times its width and is therefore $f(t_i)\Delta t$, where t_i is the time at the middle of the rectangle and $f(t_i)$ the amplitude at t_i. As we can see now the sum of the areas of these rectangles approximates the grey area in the top panel quite well. If there are N samples between a and b then the sum of the areas of the rectangles will be $\sum_{i=1}^{N} f(t_i)\Delta t = \Delta t \sum_{i=1}^{N} f(t_i) = \Delta t N \overline{f}$. Here $\overline{f}$ is the average value of $f(t)$ over the interval and $\Delta t N$ is the duration of the interval. This is a nice result because it shows that

for a sampled sound the integral equals the sound's average value times its duration.

In Praat we have the Sound > Query > Get mean... command to access a sound's mean value and it is therefore easy to derive the value of the area under a sound curve.

A.10. Interpolation and extrapolation

Given some pairs of numbers (x_i, y_i) on an domain $[x_{\min}, x_{\max}]$. Interpolation is the estimation of values y_j from any x_j on the domain.

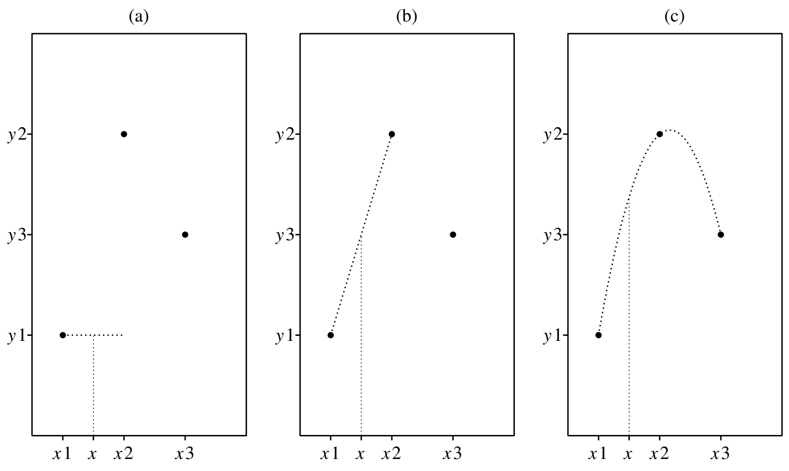


Figure A.20.: Interpolation examples. (a) constant, (b) linear, (c) quadratic or parabolic.

Constant interpolation The amplitude is assumed to be constant. In pane (a) of figure A.20 we show an example. The y-value at a point x that lies between x_1 and x_2 can be found by extending the y-value of x_1. Therefore $y = y_1$.

Linear interpolation Given two points (x_1, y_1) and (x_2, y_2) we assume a straight line through the two points. The equation of a straight line is of the form $y = ax + b$, where a and

b are parameters that can be calculated such that the line passes through the two points. The parameters a and b can be calculated as:

$$\begin{aligned} a &= \frac{y_1 - y_2}{x_1 - x_2} \\ b &= y_1 - a x_1 \end{aligned} \tag{A.14}$$

Now for any x the corresponding y-value can be calculated as $y = ax + b$. In pane (b) of figure A.20 we show an example of linear interpolation. The interpolated y-value always lies between the y-values of the two surrounding points, i.e. $\min(y_1, y_2) \leq y \leq \max(y_1, y_2)$.

Quadratic or parabolic interpolation Given three points (x_1, y_1), (x_2, y_2) and (x_3, y_3), the parameters a, b, and c of the quadratic curve (parabola) $y = ax^2 + bx + c$ that touches all three points can be calculated as:

$$\begin{aligned} a &= \frac{(x_2 - x_3)(y_1 - y_2) - (x_1 - x_2)(y_2 - y_3)}{(x_1^2 - x_2^2)(x_2 - x_3) - (x_2^2 - x_3^2)(x_1 - x_2)} \\ b &= \frac{(y_1 - y_2) - a(x_1^2 - x_2^2)}{x_1 - x_2} \\ c &= y_1 - a x_1^2 - b x_1 \end{aligned} \tag{A.15}$$

In pane (c) of figure A.20 we show an example. The figure already makes clear that an interpolated y-value may be larger (or smaller) than any of the y_i of the three points.

Sinc interpolation For sampled signals we can use sinc interpolation as it perfectly reconstructs the original signal from the sample values. This interpolation is used very often in Praat because of the famous sampling theorem which states that for a correctly sampled signal[7] $s(t)$ the original can be reconstructed from the sample values s_k as

$$s(t) = \sum_{k=-\infty}^{+\infty} s_k \mathrm{sinc}\pi(t - t_k), \tag{A.16}$$

where the s_k represent the sample value at time $t_k = t_0 + kT$ and T is the sampling time whose inverse $1/T$ is called the sampling frequency. The formula states that given sample values s_k at regularly spaced intervals, we can calculate the value at any value of the time t, by first weighing sample values according to a sinc function and then adding them. In contrats with the interpolations defined above, the sync interpolation is laborous because the determination of the amplitude at any only *one instance in time* involves *all* sample values.

[7] A correctly sampled signal can be derived from a band-limited analog signal by sampling with a sampling frequency of at least twice the bandwidth. For speech sounds that have frequencies that start at 0 Hz, the bandwidth equals the highest frequency present. The sampling frequency than needs to be at least twice the highest frequency in the sound.

A.11. Random numbers

Numbers that are chosen at random can be useful under many different circumstances. Knuth [1998] in his standard work on random numbers mentions the following situations:

Simulation. When a computer is being used to simulate human learning, random numbers are required to make things realistic.

Sampling. When the number of possible cases is impractically large, a smaller random sample of cases can be investigated.

Numerical analysis. Sometimes very complicated numerical problems can be solved by using random numbers.

Computer programming. Testing a computer program with all kinds of possible and impossible inputs drawn randomly.

Decision making. In a football game flipping a coin by the referee decides which team plays on either side.

Aesthetics. Adding a certain kind of randomness to regularly spaced objects in a graphics can make it more pleasing.

Recreation. Rolling dice, shuffling cards and playing roulette are favorite pastimes. The roulette wheel inspired the term "Monte Carlo method" to describe algorithms that use random numbers.

The definition of what exactly a random number is, will not be treated here. We will speak of a *sequence of independent random numbers* with a specified *distribution*: each number was obtained by chance alone, each number in the sequence has nothing to do with the other numbers in the sequence, and each number has a specified chance of falling in any given range of values.

In a *uniform* distribution of numbers on a finite set each number is equally probable of being drawn. In a fair die each of the eyes one to six is equally probable. Despite the fact that on the long run the number of ones, two's etc. will approximately be equal, the probability of finding an exactly equal number of occurrences for each of the six possibilities is small. The probability that in a sequence of six throws you will exactly find one one, one two, one three, etc is only1.5 %.[8]

On a computer we can generate random sequences of numbers by an algorithm called a *random number generator*. The generator that is used in Praat is based on Knuth [1998, p.187, 602] and described as a lagged Fibonacci on real numbers in $[0, 1)$. Based on this one random number generator all other sequences of random numbers with a specified distribution can be generated.

[8]This probability is $6!/6^6 \approx 0.0154$, i.e. a chance of only 1.5 %.

A.11.1. How are random numbers used in Praat?

One of the most frequent uses of random numbers in Praat is to create noise sounds. For example, if we create a sound with amplitude values drawn from a random uniform distribution we have created a white noise variant. The following line shows how to create a white noise whose amplitude is randomly distributed in the interval $(-0.5, +0.5)$.

```
Create Sound from formula: "wn", 2, 0, 1, 44100, "randomUniform (-0.5, 0.5)"
```

The spectrum of this white noise sound is flat as figure A.21 shows.[9]

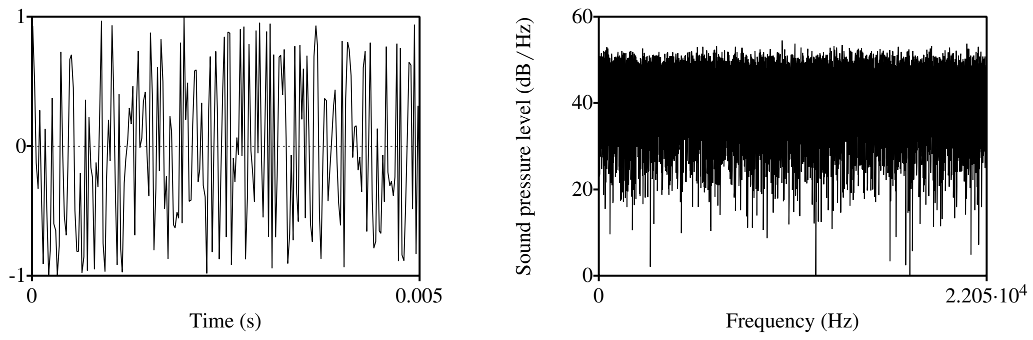


Figure A.21.: Part of a random uniform noise sound and its spectrum.

On the left we show the first 5 ms of the noise sound. The amplitude varies widely within this interval. The spectrum on the right shows that the amplitude spectrum is flat. If you would use the script above again, the resulting noise sound will be different from the one in the figure. You can execute the script above as often as you like and never will there be two noises exactly equal! However, they will all have approximately the same flat spectrum and you cannot hear any difference between all these noise sounds.

In the next example we will again generate random numbers but now they will be distributed according to a normal distribution with mean zero and standard deviation one. The script is somewhat more explicit than strictly necessary to make the relation clear between the total number of random numbers, the number of bins in the drawing interval of the distribution and the scaling of the normal curve.

```
npoints = 10000
nbins = 100
xleft = -5
xright = 5
mu = 0
s = 1
Create simple Matrix: "rg", 1, npoints, "randomGauss(mu, s)"
Erase all
```

[9]The randomUniform function that is used in the formula part will be re-evaluated for every sample in the sound, and each time it will return another random value from the (-0.5, 0.5) interval. More information about the "Create Sound from formula..." command is given in section 4.7.1.2.

```
Select outer viewport: 0, 5, 0, 5
Draw distribution:  0, 0, 0, 0, xleft, xright, nbins, 0, 500, "yes"
Line width: 2
Draw function: xleft, xright, 1000, "npoints / (nbins / (xright - xleft)) *
  ... 1 / (s*sqrt(2*pi))*exp(-x^2/(2*s^2))"
```

Figure A.22 is the plot that results from this script above. The distribution of the generated numbers follows the normal curve rather nicely.

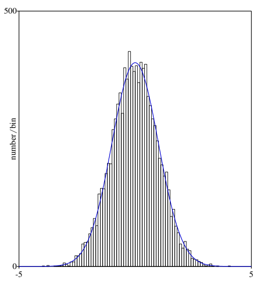


Figure A.22.: The distribution of random numbers generated according to a Gaussian distribution with mean zero and standard deviation one.

A.12. Correlations between Sounds

Because correlations involves shifting functions in the time domain we start with the concept of applying a time lag to a function.

A.12.1. Applying a time lag to a function

Applying a time lag means shifting the function in time. In figure A.23 we show for three different values of *time lag* τ how $g(t+\tau)$ and $g(t-\tau)$ behave if we know the function $g(t)$. For this example we have chosen a function $g(t)$ that increases linearly from zero at time $t=0$ to 0.5 at time $t=T_0$. Outside this interval the function equals zero. In the figure the following lag time were used: $\tau = 0$, $\tau = 0.3T_0$ and $\tau = 0.6T_0$ as indicated in the first column. In the middle and right column we consider the effect of the sign of a lag τ on $g(t)$, the middle column shows $g(t+\tau)$ and the right column shows $g(t-\tau)$. In the top panel where there is no lag, i.e. $\tau = 0$, $g(t+\tau)$ and $g(t-\tau)$ show no difference: both equal $g(t)$. The second and third row show that $g(t+\tau)$ has the same form as $g(t)$ but displaced to the left over a distance τ. In an analog way $g(t-\tau)$ is also a displaced version of $g(t)$ but now to the right. In the figure we have only considered values of τ that were greater than or equal to zero. It is not difficult to see that if the lag time becomes negative $g(t+\tau)$ behaves as $g(t-\tau)$ for positive lags. It will therefore shift to the right. We resume:

For positive values of the lag τ, we can get the function $g(t+\tau)/g(t-\tau)$ by displacing $g(t)$ to the left/right, for negative values of τ we can get the function $g(t+\tau)/g(t+\tau)$ by displacing $g(t)$ to the right/left.

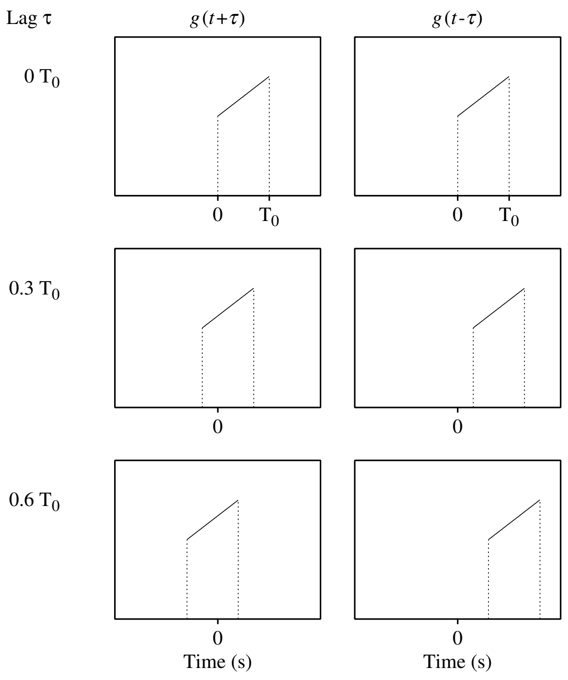


Figure A.23.: The effect of a lag τ on a function.

A.12.2. The cross-correlation function of two sounds

Informally spoken, the cross-correlation function is a measure of the similarity of two wave forms as a function of the time lag of one of them. Given two functions $f(t)$ and $g(t)$, the cross-correlation $R_{fg}(\tau)$ is defined as

$$R_{fg}(\tau) = \int_{-\infty}^{+\infty} f(t)g(t+\tau)d\tau, \tag{A.17}$$

where τ is a parameter called the *lag time*. This looks like a complicated formula: it says that to determine the cross-correlation of $f(t)$ and $g(t)$ for one particular value of the lag time τ you have to

1. displace the function $g(t)$ over the distance τ to obtain $g(t+\tau)$,
2. multiply $g(t+\tau)$ with the function $f(t)$ to obtain the product function $h(t) = f(t)g(t+\tau)$,
3. determine the integral of the product function $h(t)$.

To obtain the cross-correlation function you have to repeat the steps above for many different values of the lag time τ. We are interested in the outcome of equation (A.17) when the two functions $f(t)$ and $g(t)$ are sounds and in the sequel we will use the words *function* and *sound* interchangeably. Now let us show how to perform the calculation above in somewhat more detail.

First of all the integral sign means that we have to determine an “area”. In section A.9 we show that an integral of a sampled function can be easily calculated by multiplying the function’s average value by its duration. Therefor, once we know the sample values of a function we can easily determine its area. Further we have to calculate the product of a function $f(t)$ with a lagged version of another function $g(t)$ as $f(t)g(t+\tau)$ for different values of the lag τ. Applying a time lag is explained in section A.12.1. In figure A.24 we show step by step how to calculate the cross-correlation of two simple functions $f(t)$ and $g(t)$.

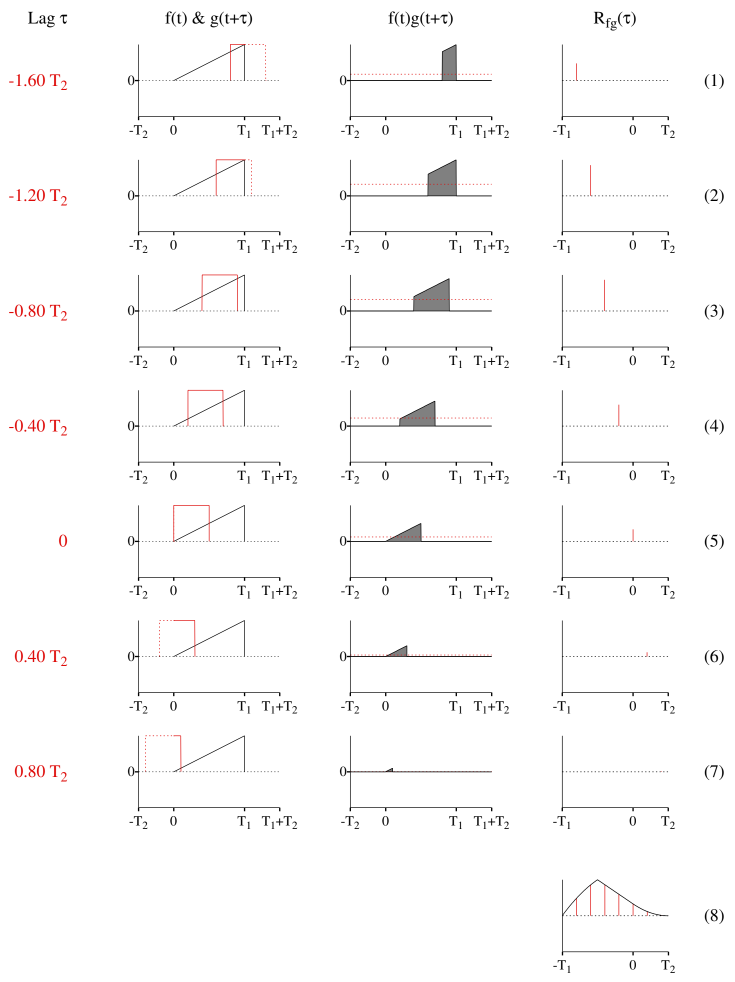


Figure A.24.: Calculation of the cross-correlation of two functions $f(t)$ and $g(t)$ for different values of lag τ. Column 1: lag time τ. Column 2: $f(t)$ in black and $g(t+\tau)$ in red. Column 3: product function $h(t) = f(t)g(t+\tau)$ with red dotted line showing the average value. Column 4: the cross-correlation value at the lag position. The bottom panel shows the accumulation of the seven cross-correlation values in red together with the complete cross-correlation function in black.

The first function $f(t)$ shows an amplitude that slowly rises from 0 to 1 between times 0 and T_1 and which is zero otherwise. The second function $g(t)$ has a constant value of 1 between times 0 and T_2 and is zero outside this domain. We have chosen these very simple functions because the steps in the algorithm can now easily be checked by eye. Figure A.24 shows in each row, from left to right, the steps involved in the calculation of one value of $R_{fg}(\tau)$ i.e. for one particular value of the lag time τ.

1. The first column, labeled "Lag τ" shows values for τ expressed as a fraction of the duration of $g(f)$. We start with an some arbitrary negative lag of $-1.6T_2$ at the top panel, increase the lag with a fixed value of $0.4T_2$ going to the next row, and end with a $0.8T_2$ positive lag at the bottom panel. In this way the seven lags almost cover the complete lag domain.

2. The second column, labeled "f(t) & g(t+τ)" shows the two functions. The $f(t)$ is drawn with a black solid line and $g(t+\tau)$ is drawn with a red line; the part of $g(t+\tau)$ that does not overlap with the domain of $f(t)$ is drawn with a dotted red line to emphasize that this part does not contribute to the cross-correlation. This column shows that the function g moves in fixed steps from the right position of the time axis in the top panel to the left position of the time axis in the bottom panel. It is easy to see that the maximum interval where $f(t)$ and $g(t+\tau)$ show overlap runs from $-T_2$ to T_1+T_2. The canonical forms of both functions $f(t)$ and $g(t)$ are shown in the fifth row where the lag time happens to be zero.

3. The third column is labeled "f(t)g(t+τ)" and shows the product function $h(t)$ of the two functions $f(t)$ and $g(t+\tau)$ for the corresponding value of τ. The vertical scale here is the same as in the previous column. As equation (A.17) shows, the cross-correlation at an instance of τ is given by the "area" of the product function. This area is indicated with grey colour. As we already know, this areas relates to the average value of this product function. The red dotted line in this column marks this average value over the domain from $-T_2$ to T_1+T_2. The time axis here extends, like in the previous column, from $-T_2$ to T_1+T_2. Outside this interval these functions do not overlap and consequently the product function will always equal zero and therefore will give no contribution to the cross-correlation. The larger the area of the product function $h(t)$, the larger the cross-correlation. The area is largest when the two functions maximally overlap.

4. Finally, the column labeled $R_{fg}(\tau)$ shows the value of the cross-correlation at the corresponding lag time. This value was obtained by multiplying the average value of $h(t)$, as shown in the previous column, with the duration of the domain of $h(t)$, i.e. T_1+2T_2. The area enclosed by $h(t)$ and the horizontal axis therefore equals the area of the rectangular area between the red dotted line and the horizontal axis. Because positive lags in $g(t+\tau)$ result in a shift of $g(t)$ to the left, i.e. to negative times, and negative lags result in shifts of $g(t)$ to positive times as we also demonstrate in section A.12.1, it is now easy to see that the *domain of the cross-correlation function* is from lag $-T_1$ to lag T_2. For displaying purposes only, the vertical scale in this column differs from the previous ones.

In the bottom panel the cross-correlation data of the last column are accumulated. For comparison the complete cross-correlation function has also been drawn in the panel which shows that our calculations of the cross-correlations at the lag times were correct. [10]

[10]The complete cross-correlation as follows:

```
s1 = Create Sound from formula:  "x", "Mono", 0, 1, 1000, "x"
s2 = Create Sound from formula:  "1", "Mono", 0, 0.5, 1000, "1"
plusObject:  s1
```

Equation (A.17) shows that the order of the two functions $f(t)$ and $g(t)$ matter. It is easy to show that $R_{fg}(\tau) = R_{gf}(-\tau)$, i.e. interchanging argument functions $f(t)$ and $g(t)$ only time-reverts the cross-correlation.

The calculation scheme in the figure shows that a lot of computation will be involved in the calculation of the cross-correlation function for all values of the lag time. Luckily the calculation can be performed efficiently with the help of the Fourier transform.[11]

Now that we know how the cross-correlation can be calculated for any value of the lag τ, we can show some examples of its use. We note that that if the two functions $f(t)$ and $g(t+\tau)$ maximally overlap, the area under the product $h(t)$ is a maximum too. A trivial example is when for some τ the functions $f(t)$ and $g(t+\tau)$ happen to be "equal". This can be used to find the location of a small duration signal within a signal of much larger duration.

A.12.2.1. Cross-correlating sines

To show you how we use the cross-correlation function to find a small fragment within a larger signal we present the following example. Let us start with one of the simplest possible signals: a sine wave. In the first two columns of figure A.25 we show the two sounds, f and g, whose cross-correlation, R_{fg}, is in the third column column. The first row shows one period of a sine, three periods of a sine/tone and their cross-correlation, respectively. R_{fg} shows a peak at lag time zero and other peaks at multiples of the period. We can easily see why: at zero lag time the signal in the first column completely overlaps the sound in the second column thus giving rise to a maximum area under the product function and consequently a peak in the cross-correlation function. Shifting g over one period again results in maximum overlap and hence another peak in the cross-correlation function. The periodicity repeats itself. The second row shows the cross-correlation of a sine but this time with a phase of $pi/2$, i.e. a cosine. Now the peaks in the cross-correlation occurs a quarter of a period later as indicated by the t_2 marks on the horizontal axes. In the third row the sine has a phase of π. Now the peak occurs one half of a period later as is indicated by the t_3 markers. There is a valley at lag time zero, here the two signals are in anti phase. The first three rows show that the cross-correlation function is able to show exactly the place of occurrence of a small sine fragment within a larger part. In the last two rows of the figure we have made the task somewhat more realistic: we paste the sound of the left panel in the middle of another sound, the position of pasting was at time t_4. The cross-correlation at the right nicely displays a peak at time t_4 which means that it has "found" the correct starting point of the hidden sound. In contrast with the panels in the first three rows, the cross-correlation here does not show a number of peaks with all nearly at the same amplitude, but only one peak with a substantial amplitude. The other peaks all are of much smaller amplitude. This shows that the cross-correlation is able to find the best match between the two signals. Even if a fair amount of noise is present, like in the bottom panel, the peak occurs at $t_5 \simeq t_4$ and the cross-correlation is able to find the right location.

```
Cross-correlate:  "integral", "zero"
Draw:  <etc>
```

[11]If $f(t) \Leftrightarrow F(\omega)$ and $g(t) \Leftrightarrow G(\omega)$ are Fourier transform pairs then $R_{fg}(\tau) \Leftrightarrow F^{*}(\omega)G(\omega)$ is a Fourier transform pair too.

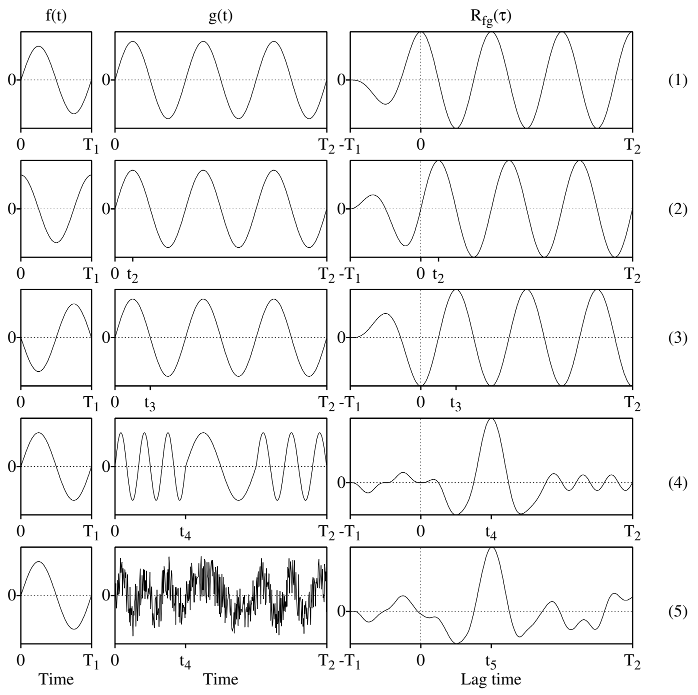
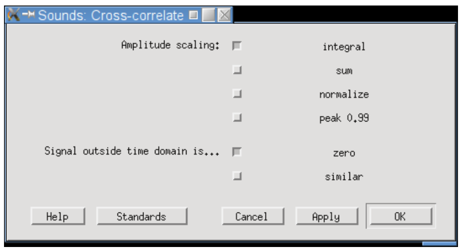


Figure A.25.: The cross-correlation of the sound in the column labeled $f(t)$ with the sound in the column labeled $g(t)$ results in the sound in the column labeled $R_{fg}(t)$. The sounds f have exact duration T_1 while the sounds $g(t)$ actually are of longer duration than the T_2 seconds shown.

A.12.2.2. Praat's cross-correlation


Sounds: Cross-correlate

Amplitude scaling: [✓ integral] sum normalize peak 0.99

Signal outside time domain is... [✓ zero] similar

Figure A.26.: The Sounds: Cross-correlate... form.

A cross-correlation is performed by first selecting the two sounds that you want to cross-correlate and then choosing the Cross-correlate... command. Next the form of figure A.26 appears. The amplitude scaling options of the resulting sound represent:

integral. Equation (A.17) is numerically evaluated as $\sum_{i=1}^{N} f(t_i)g(t_i+\tau)\Delta t$, where Δt is the sampling period and t_i are the sampling times. The resulting object has dimensions of Pa^{2}s.

sum. Equation (A.17) is now evaluated as $\sum_{i=1}^{N} f(t_i)g(t_i+\tau)$ which differs from the integral evaluation only by a factor of Δt. The resulting object now has dimensions of Pa^2.

normalize. We calculate the normalized cross-correlation, defined as

$$R'_{fg}(\tau) = \frac{\int_{-\infty}^{+\infty} f(t)g(t+\tau)d\tau}{\sqrt{\int_{-\infty}^{+\infty} f^2(t)dt \int_{-\infty}^{+\infty} g^2(t)dt}}.$$

Effectively we divide the cross-correlatation by the square root of the product of the energies of the two sounds which will result in a dimensionless object.

peak 0.99. This is a pragmatic scaling to be able to play the resulting sound without audible distortion.

The result of a cross-correlation is a new sound object in the list of objects. The dimensions of the result of the correlation with each of the first three scalings differ and none really conforms to dimension of a sound. A sound's amplitude is expressed in Pa and equation (A.17) makes clear that the dimension of the cross-correlation is in Pa^{2}s since it is obtained by multiplying two sounds together followed by a multiplication with a time unit. Therefore, the dimension of Pa^2 for the "sum" scaling and the dimensionless object from the "normalize" scaling are also not in compliance with the dimension of a real sound. Nevertheless, once we have a sound object, its history may turn out to be completely irrelevant.

A.12.3. The autocorrelation

Informally spoken, the autocorrelation function of a sound shows the similarity of the sound with displaced versions of itself. The autocorrelation function is often written as $r(\tau)$, where τ is the displacement time or *lag time*. The autocorrelation of a sound equals the cross-correlation of a sound with itself. Given a function $f(t)$, it is defined as

$$r(\tau) = \int_{-\infty}^{+\infty} f(t)f(t+\tau)d\tau. \tag{A.18}$$

It can be used as a tool to find repeating patterns in a sound. The pitch algorithm in Praat uses the autocorrelation function. For a periodic sound with period T, the autocorrelation function will show maxima at *lag* times that are multiples of the period T, as we will demonstrate shortly.

Some properties of the autocorrelation function:

- The autocorrelation function is a symmetric function, i.e. $r(\tau) = r(-\tau)$.

- The maximum peak is always at the origin, i.e. for $\tau = 0$. We can state this mathematically as $|r(\tau)| \leq r(0)$. Sometimes the autocorrelation function is divided by the maximum $r(0)$ to obtain the normalized autocorrelation function $r\prime(\tau) = r(\tau)/r(0)$. The maximum of the normalized autocorrelation function always equals one.
- The autocorrelation function of a periodic function has peaks at intervals of the period.
- The autocorrelation of white noise sound will show a peak at $\tau = 0$ and will be almost zero for all other values of τ. This can be used to reveal hidden periodicity of a sound buried in noise.
- The autocorrelation of the sum of two uncorrelated functions is the sum of the autocorrelations of these functions.

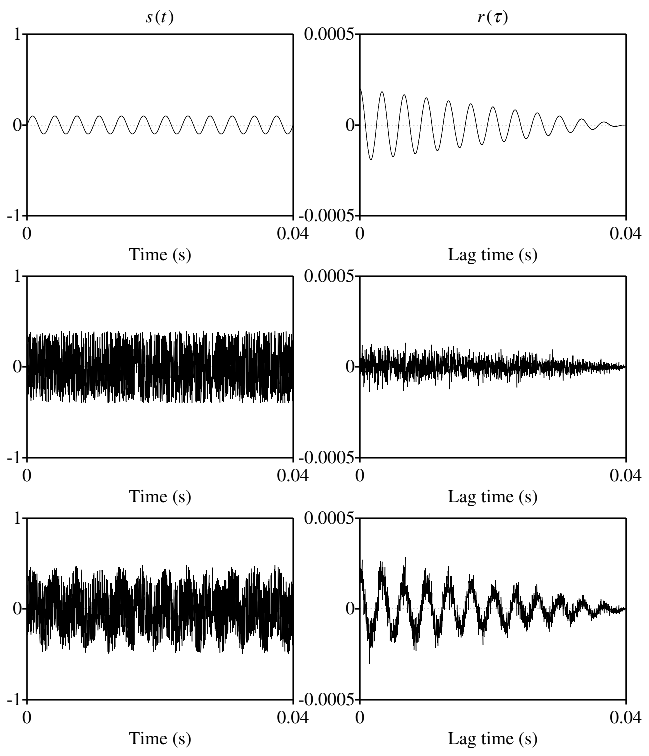


Figure A.27.: The left column shows three different sounds of 40 ms duration. In the top panel the sound is a tone of 300 Hz with amplitude 0.1, in the middle panel it is random uniform noise with amplitudes between −0.4 and 0.4 and in the bottom panel it is the sum of these two sounds. The right column shows the autocorrelation of the corresponding sound in the left column for positive lag times only.

We now show how the noise cancellation by the autocorrelation function can help find hidden periodicity. For this example we have generated three sounds of 40 ms duration. The first sound is a tone of 300 Hz with an amplitude of 0.1. The second sound is a random uniform noise with noise amplitudes between −0.4 and 0.4. The third sound in the sum of these sounds, i.e. it is the periodic sound buried in the random uniform noise. In the left column of figure A.27 these sound are displayed. The time scale permits seeing the nice periodic tone in the top panel. In the bottom panel in the first column no clear structure appears in the sound: the periodicity of the tone is hardly visible. This, however, changes for the autocorrelation of this sound as is shown in the panel in the column on the right. Because of the symmetry of the autocorrelation, only the parts for positive lag times are displayed. We also limited the vertical scale of the autocorrelation to be able to compare the meaningful parts. In the middle and bottom panel the values at $\tau = 0$ are therefore clipped because they are much larger than the vertical scale indicates. We see from the bottom panel that despite the noise, the autocorrelation reveals the periodicity. This can happen because the noise samples are not correlated and so the autocorrelation function of a noise is almost zero everywhere except at $\tau = 0$.

A.12.3.1. The autocorrelation of a periodic sound

Figure shows the autocorrelation of a two different perfectly periodic signals both with a period of 0.01 s. The first sound is a tone of 100 Hz and the second sound is a spike train with spikes 0.01 s apart.

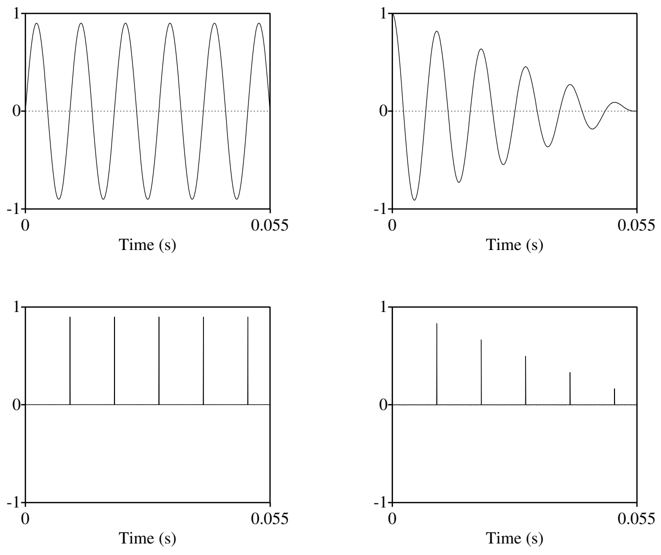
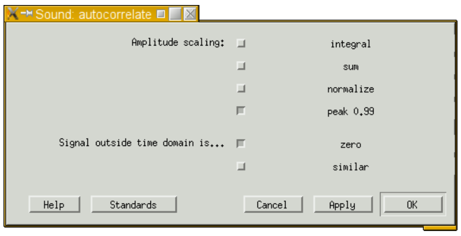


Figure A.28.: The autocorrelation of two periodic sounds.

A.12.3.2. Praat's autocorrelation



Amplitude scaling: integral / sum / normalize / [✓ peak 0.99]

Signal outside time domain is... [✓ zero] / similar

Figure A.29.: The Sound:autocorrelate... form.

An autocorrelation is performed by first selecting a sound and then choosing the Autocorrelate... command. Next the form in figure A.29. The amplitude scaling options of the resulting sound represent:

integral. Equation (A.18) is numerically evaluated as $\sum_{i=1}^{N} f(t_i)f(t_i+\tau)\Delta t$, where Δt is the sampling period and t_i are the sampling times. The resulting object has dimensions of $\mathrm{Pa}^2\mathrm{s}$.

sum. Equation (A.18) is now evaluated as $\sum_{i=1}^{N} f(t_i)f(t_i+\tau)$ which differs from the integral evaluation only by a factor of Δt. The resulting object now has dimensions of Pa^2.

normalize. We calculate the normalized autocorrelation, defined as

$$r'_f(\tau) = \frac{\int_{-\infty}^{+\infty} f(t)f(t+\tau)d\tau}{\int_{-\infty}^{+\infty} f^2(t)dt}.$$

We divide the autocorrelatation by the energy of the sound which will result in a dimensionless object. The maximum amplitude will occur at $\tau = 0$ and be equal to one.

peak 0.99. This is a pragmatic scaling to be able to play the resulting sound without audible distortion.

The result of the autocorrelation is a new sound object in the list of objects. Because the autocorrelation of a sound is a symmetric function, only the values for lag times greater equal than zero are given.

The arguments used at the end of section (A.12.2.2) about apply here too, the dimensions of the result of the autocorrelation with these scalings differ and none really conforms to the dimensions of a sound. Nevertheless, once we have a sound object, its history may turn out to be completely irrelevant.

A.13. The Σ summation sign

The sigma sign, Σ, is just a notational shorthand for compactly writing a *sum of terms*. Instead of writing for example $1 + 2 + 3 + 4 + 5$ we can write $\sum_{k=1}^{5} k$. For this 5-term sum this is not really much shorter. However, if we want to extend this sum to the first 20 integers then $\sum_{k=1}^{20} k$ is really shorter than 1 + 2 + ... + 19 + 20, and less error prone.

Other examples:

$\sum_{k=1}^{25} 1/k$ Sum the inverses of the first 25 integers $(1/1 + 1/2 + \ldots + 1/25)$.

$\sum_{k=1}^{20} k^2$ Sum the first 20 squares: $(1^2 + 2^2 + \ldots + 20^2)$.

If we want to compose a signal with a number of harmonically related frequency components as happens in Fourier synthesis we may write something complicated as $s(t) = \sum_{k=0}^{N}\{a_k \sin(2\pi k f_o t) + b_k \cos(2\pi k f_o t)\}$, where the a_k,the $b_k$and the f_0 are known numbers. This expression states that $s(t)$ is the sum of sines and cosines whose frequencies are multiples of some fundamental frequency f_0.

An example where we use this symbol is in the Shannon-Nyquist theorem. . .

A.14. Abouts bits and bytes

In order to understand the binary representation of information in a computer, we need to know some elementary facts about numbers and how they can be represented. We therefore start with an excursion into *number systems.*

The first aspect we discuss is how the numbers are written. We normally use the *Arab system* and write numbers as 1 or 15 or 4567890, etc. The Romans, for example, used a completely different system to write numbers, their numbers were written as I, or IV or MCLXIII. We conclude from this that a number in its written form consists of a sequence of symbols and that to get the value of this sequence, certain rules have to be applied on this symbol sequence.

A.14.1. The Roman number system

The Romans used the symbols I (value 1), V (value 5), X (value 10), L (value 50), C (value 100), D (value 500) and M (value 1000) to represent numbers. The rules to attain the value represented by a sequence of these symbols, are fairly complicated. If the symbols, from left to right, are such that the values of the symbols are not increasing, these values can be added directly. For example, in the number CXII the ordering is non-increasing: the C on the left represents the highest value, 100, the C is followed by the X with the lower value 10. The X is followed by the I with the lower value 1 and this I is followed by another I. This makes the sequence non-increasing. Its value can be calculated as C+X+I+I = 100+10+1+1=112.

Because the Romans did not allow to write more than three symbols with the same value next to each other, they needed a way to write for example 4 ($\neq$ IIII) or 9 ($\neq$ VIIII) or 14($\neq$ XIIII) or 40($\neq$ XXXX). They allowed a lower symbol to be on the left of the next higher symbol and the value of the lower symbol had to be subtracted from the higher symbol. In the representation MCMLXXIX there are two lower symbols before a higher symbol: the C (=100) is before the second M (=1000) and the I (=1) is before the third X. The value of this number is therefore 1000+(−100+1000)+50+10+10+(−1+10) = 1000+900+50+10+10+9 = 1979. The first 15 numbers in the Roman system are I=1, II=2, III=3, IV=4, V=5, VI=6, VII=7, VIII=8, IX=9, X = 10, XI = 11, XII=12, XIII=13, XIV=14, XV=15. In this system performing arithmetic, like adding numbers or multiplying them, is a horror.

A.14.2. The decimal system

In the decimal system that we are used to, the rules to obtain the value of written numbers are much simpler than in the Roman system discussed above. The Romans had a system where the value of a symbol was fixed. We use a system where the value of a symbol also depends on its *position* in the sequence. For example, the numbers 12 (twelve) and 1234 (one thousand three hundred and thirty four) both have the digit 1 in the first position from the left. However, the value of this digit in both numbers is very different. In 12 the 1 represents the value ten while it represents the value thousand in 1234. The position of the digit in the sequence determines its value: the more to the left in the sequence the higher its value, the more to the right the lower its value. This is a very elegant and efficient representation of numbers. It allows us to represent an infinite number of values with only 10 symbols, the digits from 0 to 9.

Now we will try to make the way in which the value of a sequence of symbols is constructed, more explicit. The rule to obtain the value of a sequence of digits in a positional system like our decimal system, is as follows: multiply the value of each symbol in the sequence with the value corresponding with its position in the sequence and sum the results. For example, the value of a sequence of four digits, the number 1234, can be calculated as $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$. From this representation we can see that the first position on the right is worth one, the second position from the right is worth ten, the third hundred, and the fourth position from the right thousand. We rewrite the values associated with position as powers of 10, $1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$. The system becomes clear: the values of neighboring positions differ by a factor of 10. Mathematicians like to generalize this and write that the value of position p is 10^{p-1}, if we count the positions going from right to left with the rightmost position at $p = 1$. Check: in the number 456712 the value of the fifth position from the right is $10000 = 10^{5-1} = 10^4$, and the symbol 5 at this position adds 50000 to the value of this number. In these numbers a value of 10 plays a crucial role, it is called the *base* of the decimal system.

A.14.3. The general number system

Mathematicians like to formalize the rules to obtain the value of a sequence of n symbols. They write such a sequence as $s_n s_{n-1} \cdots s_3 s_2 s_1$, where each of the s_p is one of the symbols from the number system. We start with the decimal system where we know that the value of this sequence of digits can be calculated as $s_n \times 10^{n-1} + s_{n-1} \times 10^{n-2} + \cdots + s_4 \times 10^3 + s_3 \times 10^2 + s_2 \times 10^1 + s_1 \times 10^0$, where each of the symbols appearing before a power of ten represents one of the digits $0, 1, \cdots, 9$. Lets define a symbol b that has the value 10. We can then write the value of the sequence as

$$s_n s_{n-1} \cdots s_3 s_2 s_1 = s_n \times b^{n-1} + s_{n-1} b^{n-2} + \cdots + s_3 \times b^2 + s_2 \times b^1 + s_1 \times b^0, \qquad \text{(A.19)}$$

in which the value of a digit s_p at a position p from the right is $s_p \times b^{p-1}$. This is the desired result. Check: in the number 143298 the digit at $p = 5$, i.e. 4, represents the value $4 \times 10^{5-1} = 40000$.

The last step. Up till now we have used the decimal system: base $b = 10$ with 10 symbols, the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We now leave the familiar decimal system and allow the base b to be any number. A number system with base b needs b different symbols to write numbers with that base. We could use any symbol set but traditionally certain conventions are followed. If we choose a base smaller than 10 the digits 0, 1,$\cdots$, b are used. For example, in the octal system, the base b is 8 and the symbols are 0, 1, 2, 3, 4, 5, 6, and 7. To distinguish numbers with different bases the base is often used as a subscript. Some examples of octal numbers and their values in the decimal system are: $10_8 = 1 \times 8^1 + 0 \times 8^0 = 8 + 0 = 8_{10}$ and $457_8 = 4 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 4 \times 64 + 5 \times 8 + 7 = 303_{10}$.

Although the decimal system is dominant we use other number systems as well. We count 60 seconds in a minute and 60 minutes in an hour, 24 hours in a day, 7 days in a week, 52 weeks make one year. With time it gets even more weird: we don't count in units of 1/60 s but in units of 1/100 s. We divide the circle in 360°. Even in counting remnants of other number system show up: the French swap base 10 and base 20 systems for numbers from 80 to 99. They say *quatre-vingts* for 80, which means four times twenty, they say 81 is $4 \times 20 + 1$, they say 93 is *quatre-vingts-treize*, $4 \times 20 + 13$.[12]

A.14.4. Number systems in the computer

In the computer world three number systems are dominant, the hexadecimal system, the octal system and the binary system. Of these three the octal system is used less frequently than the other two. The hexadecimal system uses base $b = 16$ and its symbols are the digits 0 to 9 which make up the first 10 symbols, the addition six are the A ($= 10_{10}$), B ($=11_{10}$), C ($= 12_{10}$), D (13_{10}), E ($= 14_{10}$) and F ($= 15_{10}$). Examples of hexadecimal numbers are $10_{16} = 1 \times 16 + 0 \times 16^0 = 16$ and $FF_{16} = 15 \times 16^1 + 15 \times 16^0 = 255_{10}$.

In the binary system the base $b = 2$, the two symbols used are 0 and 1. Some examples of binary numbers are $10_2 = 1 \times 2^1 + 0 = 2_{10}$, $11101 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 = 16 + 8 + 4 + 0 + 1 = 29_{10}$, $111_8 = 2^2 + 2 + 1 = 7_{10}$.

Now the basics of number systems have been explained, we know how to calculate the decimal value of a number in a certain number system with equation A.19. We have not shown yet how to do simple arithmetic like add and subtract in a particular number system, or how to transform numbers from one number system to any other.

Table A.5.: Conversion from decimal to binary by long division.

2	213_{10}	Remainder
	106	1
	53	0
	26	1
	13	0
	6	1
	3	0
	1	1
	0	1

An easy algorithm to convert a number from decimal to binary notation is by long division

[12]For numbers between 60 and 70 they mix base 10 and base 20. They say 71 is *soixante et onze*, 60 + 11.

as exemplified by the first three columns in table A.5. We start with 213_{10} and divide by 2. The result, 106, is written on the next line in the same column, the remainder, 1, in the next column. Check: $213 = 2 \times 106 + 1$, 213 is an uneven number and uneven numbers always have a remainder of 1 after division by 2. We continue the process and now divide the 106 by 2 and write the result and remainder on the next line. We continue $53 = 2 \times 26 + 1$, $26 = 2 \times 13 + 0$, $13 = 2 \times 6 + 1$, $6 = 2 \times 3 + 0$, $3 = 1 \times 2 + 1$, $1 = 0 \times 2 + 1$. The binary representation can now be formed by arranging the numbers in the Remainder column in the following way. Get the top number, 1, and write it down, get the number in the next row, 0, and write it to the left of the previous number, continue this procedure until there are no numbers left. You now have the sequence 11010101 which forms the binary representation of 213_{10}. Check: $11010101 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^2 + 1 = 128 + 64 + 16 + 4 + 1 = 213_{10}$. This procedure by long division works in any conversion from decimal to another number system as the following table shows.

Table A.6.: Conversions from decimal.

16	213_{10}	Remainder	10	213_{10}	Remainder
	13	5		21	3
	0	D		2	1
				0	2

A.15. Matrices

A matrix is a rectangular table of elements. Most of the time these elements are numbers. The horizontal and vertical lines in a matrix are called *rows* and *columns*. For example, the following matrix A has 3 rows and 5 columns.

$$A = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 \\ 2 & 5 & 8 & 11 & 14 \\ 3 & 6 & 9 & 12 & 15 \end{bmatrix}$$

Any element in a matrix can be indexed by a pair of numbers as (row number, column number). For example, the element with value 8 in the matrix A has row and column indices (2, 3). The convention is that a matrix is indicated with an alphabetic character in uppercase, like A in our example. There are various ways to indicate a matrix element. The only thing they have in common is that the matrix element is written in lowercase. For example, the element of matrix A with value 8 can be referred to as $a_{2,3}$ or $a(2,3)$ or $a[2][3]$ or $a[2,3]$.

A matrix with only one row is called a row vector, a matrix with only one column is called a column vector. A matrix with equal number of rows and columns is called a *square* matrix.

Matrices are used in Praat all over the place. For example, to represent sounds we use matrices. The samples of a mono sound are a matrix with only one row; the samples of a stereo sound are a matrix with two rows, each channel is represented as a separate row.[13] The only difference with a matrix like the one above is that a matrix with a sound normally has much more columns than 5. For example, a mono sound with a duration of 1 s and a sampling frequency of 44100 Hz has 1 row and 44100 columns. A stereo sound of the same duration and sampling frequency has two rows and 44100 columns.

A.16. Complex numbers

A complex number is a pair of real numbers with some special properties. We write the complex number z as $z = a + bi$, where a and b are real numbers and the i is special because $i^2 = -1$. The number a is called the *real part* of the complex number and b is called the *imaginary part*.

For two complex numbers $z_1 = a + bi$ and $z_2 = c + di$ we define

- Addition: $z_1 + z_2 = (a + c) + (b + d)i$.
- Subtraction: $z_1 - z_2 = (a - c) + (b - d)i$.
- Multiplication: $z_1 z_2 = (a+bi)(c+di) = ac+adi+bci+bdi^2 = (ac-bd)+(ad+bc)i$.

We define the complex conjugate of z as $z^* = a - bi$, i.e. the complex conjugate has the imaginary part of opposite sign. It follows that $zz^* = (a+bi)(a-bi) = a^2-abi+abi-b^2i^2 = a^2 + b^2$. We use this trick to define

- Division: $z_1/z_2 = \frac{a+bi}{c+di} = \frac{a+bi}{c+di} \cdot \frac{c-di}{c-di} = \frac{ac+bd+(bc-ad)i}{c^2+d^2}$.

A.16.1. Some rules of sines and cosines

All the trigonometric relations between sums and products of sines and cosines can easily be derived once we know about complex numbers. We start from the famous relation of Euler:

$$e^{i\phi} = \cos\phi + i\sin\phi. \tag{A.20}$$

Substitution of $-\phi$ for ϕ results in

$$e^{-i\phi} = \cos\phi - i\sin\phi. \tag{A.21}$$

In the latter formula we have used the fact that a cosine is a symmetric function, i.e. $\cos(-\phi) = cos\phi$ and the sine is antisymmetric, i.e. $\sin(-\phi) = -\sin\phi$. Subtraction and addition of these two equations results in expressions for the sine and cosine:

$$\sin\phi = \frac{e^{i\phi} - e^{-i\phi}}{2i} \tag{A.22}$$

$$\cos\phi = \frac{e^{i\phi} + e^{-i\phi}}{2}. \tag{A.23}$$

[13]To complicate things: the Matrix type in Praat also consists of a table of numbers. However, these numbers represent sampled values on a domain.

These equations are all we need to derive rules for the arguments of the form $\alpha + \beta$. We substitute $\phi = \alpha + \beta$ in the formula (A.20) above and obtain

$$e^{i(\alpha+\beta)} = \cos(\alpha+\beta) + i\sin(\alpha+\beta). \tag{A.24}$$

We can also write

$$\begin{aligned} e^{i(\alpha+\beta)} &= e^{i\alpha}e^{i\beta} \\ &= (\cos\alpha + i\sin\alpha)(\cos\beta + i\sin\beta) \\ &= \cos\alpha\cos\beta - \sin\alpha\sin\beta + i(\cos\alpha\sin\beta + \sin\alpha\cos\beta). \end{aligned} \tag{A.25}$$

For the right-hand sides of equations (A.24) and (A.25) to be equal, the real parts have to be equal and the imaginary parts have to be equal. We obtain

$$\begin{aligned} \cos(\alpha+\beta) &= \cos\alpha\cos\beta - \sin\alpha\sin\beta \\ \sin(\alpha+\beta) &= \cos\alpha\sin\beta + \sin\alpha\cos\beta. \end{aligned} \tag{A.26}$$

For β negative we obtain:

$$\begin{aligned} \cos(\alpha-\beta) &= \cos\alpha\cos\beta + \sin\alpha\sin\beta \\ \sin(\alpha-\beta) &= -\cos\alpha\sin\beta + \sin\alpha\cos\beta. \end{aligned} \tag{A.27}$$

By combining terms we easily get

$$\sin\alpha\sin\beta = \left(\cos(\alpha-\beta) - \cos(\alpha+\beta)\right)/2 \tag{A.28}$$
$$\cos\alpha\sin\beta = \left(-\sin(\alpha-\beta) + \sin(\alpha+\beta)\right)/2 \tag{A.29}$$
$$\sin\alpha\cos\beta = \left(\sin(\alpha-\beta) + \sin(\alpha+\beta)\right)/2 \tag{A.30}$$
$$\cos\alpha\cos\beta = \left(\cos(\alpha-\beta) + \cos(\alpha+\beta)\right)/2. \tag{A.31}$$

If the arguments α and β are equal we obtain

$$\sin^2\alpha = 1/2 - 1/2\cos(2\alpha) \tag{A.32}$$
$$\sin\alpha\cos\alpha = 1/2\sin(2\alpha) \tag{A.33}$$
$$\cos^2\alpha = 1/2 + 1/2\cos(2\alpha) \tag{A.34}$$

Adding equations A.32 and A.34 we obtain the familiar result

$$\sin^2\alpha + \cos^2\alpha = 1. \tag{A.35}$$

For the sum of two cosines with different arguments we obtain

$$\begin{aligned} \cos\alpha + cos\beta &= (e^{i\alpha} + e^{-i\alpha})/2 + (e^{i\beta} + e^{-i\beta})/2 \\ &= (e^{i\alpha} + e^{i\beta} + e^{-i\alpha} + e^{-i\beta})/2 \\ &= \left(e^{i(\alpha+\beta)/2}(e^{i(\alpha-\beta)/2} + e^{-i(\alpha-\beta)/2}) + e^{-i(\alpha+\beta)/2}(e^{i(\alpha-\beta)/2} + e^{-i(\alpha-\beta)/2})\right)/2 \\ &= (e^{i(\alpha+\beta)/2} + e^{-i(\alpha+\beta)/2})(e^{i(\alpha-\beta)/2} + e^{-i(\alpha-\beta)/2})/2 \\ &= 2\cos\left((\alpha+\beta)/2\right)\cos\left((\alpha-\beta)/2\right) \end{aligned} \tag{A.36}$$

and for the sum of two sines

$$\begin{aligned}
\sin\alpha + \sin\beta &= (e^{i\alpha} - e^{-i\alpha})/(2i) + (e^{i\beta} - e^{-i\beta})/(2i) \\
&= \left(e^{i\alpha} + e^{i\beta} - (e^{-i\alpha} + e^{-i\beta})\right)/(2i) \\
&= \left(e^{i(\alpha+\beta)/2}(e^{i(\alpha-\beta)/2} + e^{-i(\alpha-\beta)/2}) - e^{-i(\alpha+\beta)/2}(\ldots)\right)/(2i) \\
&= \left((e^{i(\alpha+\beta)/2} - e^{-i(\alpha+\beta)/2})(e^{i(\alpha-\beta)/2} + e^{-i(\alpha-\beta)/2})\right)/(2i) \\
&= 2\sin\left((\alpha+\beta)/2\right)\cos\left((\alpha-\beta)/2\right). \qquad \text{(A.37)}
\end{aligned}$$

A.16.2. Complex spectrum of simple functions

A.16.2.1. The block function

The block function is defined as

$$b(x) = \begin{cases} 1 & 0 \leq x \leq T_0 \\ 0 & \text{elsewhere} \end{cases} \qquad \text{(A.38)}$$

Its Fourier transform is defined as

$$\begin{aligned}
B(f) &= \int_{-\infty}^{\infty} b(x)e^{-2\pi i f t}dt \\
&= \int_{0}^{T_0} 1e^{-2\pi i f t}dt \\
&= \left[\frac{e^{-2\pi i f t}}{-2\pi i f}\right]_0^{T_0} \\
&= \frac{e^{-2\pi i f T_0} - 1}{-2\pi f} \\
&= T_0 e^{-i\pi f T_0}\left(\frac{e^{-i\pi f T_0} - e^{i\pi f T_0}}{-2\pi f T_0}\right) \\
&= T_0 e^{-i\pi f T_0}\frac{\sin(\pi f T_0)}{\pi f T_0} \\
&= T_0 e^{-i\pi f T_0}\text{sinc}(f T_0).
\end{aligned}$$

In the derivation of the transform we used identity (A.22) for the sine and (A.3) for the definition of the sinc function.

A.16.2.2. The time-limited tone

The time-limited tone function is defined as

$$t(x) = \begin{cases} \sin(2\pi f_1 x) & 0 \leq x \leq T_0 \\ 0 & \text{elsewhere} \end{cases} \qquad \text{(A.39)}$$

Its Fourier transform is

$$\begin{aligned}
T(f) &= \int_{-\infty}^{\infty} t(x)e^{-i2\pi ft}dt \\
&= \int_{0}^{T_0} \sin(2\pi f_1 t)e^{-i2\pi ft}dt \\
&= 1/(2i)\int_{0}^{T_0} (e^{i2\pi f_1 t} - e^{-i2\pi f_1 t})e^{-i2\pi ft}dt \\
&= 1/(2i)\int_{0}^{T_0} \left(e^{-i2\pi(f-f_1)t} - e^{-i2\pi(f+f_1)t}\right)dt \\
&= 1/(2i)\left[\frac{e^{-i2\pi(f-f_1)t}}{-i2\pi(f-f_1)} - \frac{e^{-i2\pi(f+f_1)t}}{-i2\pi(f+f_1)}\right]_0^{T_0} \\
&= 1/(2i)\left(\frac{e^{-i2\pi(f-f_1)T_0}-1}{-i2\pi(f-f_1)} - \frac{e^{-i2\pi(f+f_1)T_0}-1}{-i2\pi(f+f_1)}\right) \\
&= 1/(2i)\left(\frac{e^{-i\pi(f-f_1)T_0}\left(e^{-i\pi(f-f_1)T_0}-e^{i\pi(f-f_1)T_0}\right)}{-i2\pi(f-f_1)} - \frac{e^{-i\pi(f+f_1)T_0}\left(e^{-i\pi(f+f_1)T_0}-e^{i\pi(f+f_1)T_0}\right)}{-i2\pi(f+f_1)}\right) \\
&= T_0/(2i)\left(e^{-i\pi(f-f_1)T_0}\text{sinc}\left((f-f_1)T_0\right) - e^{-i\pi(f+f_1)T_0}\text{sinc}\left((f+f_1)T_0\right)\right).
\end{aligned}$$

For the power spectrum we have to calculate $T(f)T^{\star}(f)$. To simplify expressions we first write $\alpha = (f - f_1)T_0$ and $\beta = (f + f_1)T_0$.

$$\begin{aligned}
T(f)T^{\star}(f) &= T_0^2/4\left(e^{-i\pi\alpha}\text{sinc}(\alpha) - e^{-i\pi\beta}\text{sinc}(\beta)\right)\left(e^{i\pi\alpha}\text{sinc}(\alpha) - e^{i\pi\beta}\text{sinc}(\beta)\right) \\
&= T_0^2/4\left(\text{sinc}^2(\alpha) - \left(e^{i\pi(\alpha-\beta)} + e^{-i\pi(\alpha-\beta)}\right)\text{sinc}(\alpha)\text{sinc}(\beta) + \text{sinc}^2(\text{fi})\right) \\
&= T_0^2/4\left(\text{sinc}^2(\alpha) - 2\cos\pi(\alpha-\beta)\text{sinc}(\alpha)\text{sinc}(\beta) + \text{sinc}^2(\text{fi})\right).
\end{aligned}$$

We substitute the frequencies back and get the amplitude spectrum of a time-limited tone:

$$\begin{aligned}
T(f)T^{\star}(f) = T_0^2/4\Big(&\text{sinc}^2\left((f-f_1)T_0\right) - 2\cos(2\pi f_1 T_0)\text{sinc}\left((f-f_1)T_0\right)\text{sinc}\left((f+f_1)T_0\right) \\
&+\text{sinc}^2\left((f+f_1)T_0\right)\Big) \qquad \text{(A.40)}
\end{aligned}$$

This shows that the spectrum of a tone is the sum of three terms, all involving sinc functions. The first and the last term are the contributions of the frequencies at $\pm f_1$, while the middle term is the interference term between the positive and the negative frequency. We have plotted this equation and its constituent parts in figure A.30.

All plots on the left show the amplitudes on a linear scale while the plots on the right show the amplitudes on a logarithmic scale from 0 to −100 dB. The spectrum was determined from a 1 s duration sound where a tone of $f_1 = 1118.5$ Hz in the first $T_0 = 0.117313$ s was followed by silence. In all plots the horizontal scale runs from 1050 Hz to 1250 Hz for better visual display. The top panel on the left shows the $\mathrm{sinc}^2\left((f - f_1)T_0\right)$ function of equation (A.40), the contribution of the positive frequency f_1. The $10\log(\mathrm{sinc}^2\left((f - f_1)T_0\right))$ is on the right. The dotted horizontal lines are at 20 dB distances. In the second row on the left, the term $-2\cos(2\pi f_1 T_0)\mathrm{sinc}\left((f - f_1)T_0\right)\mathrm{sinc}\left((f + f_1)T_0\right)$ and on the right the logarithm of its absolute value $10\log|-2\cos(2\pi f_1 T_0)\mathrm{sinc}\left((f - f_1)T_0\right)\mathrm{sinc}\left((f + f_1)T_0\right)|$. To draw the logarithm we had to take the absolute value to avoid this term becoming negative. This term is very small, i.e. more than −30 dB lower than the previous term. The last term, $\mathrm{sinc}^2\left((f + f_1)T_0\right)$ is even smaller, more than −60 dB below the first term. These plots make clear that the amplitude spectrum is dominated by the first term alone, the $\mathrm{sinc}^2\left((f - f_1)T_0\right)$.

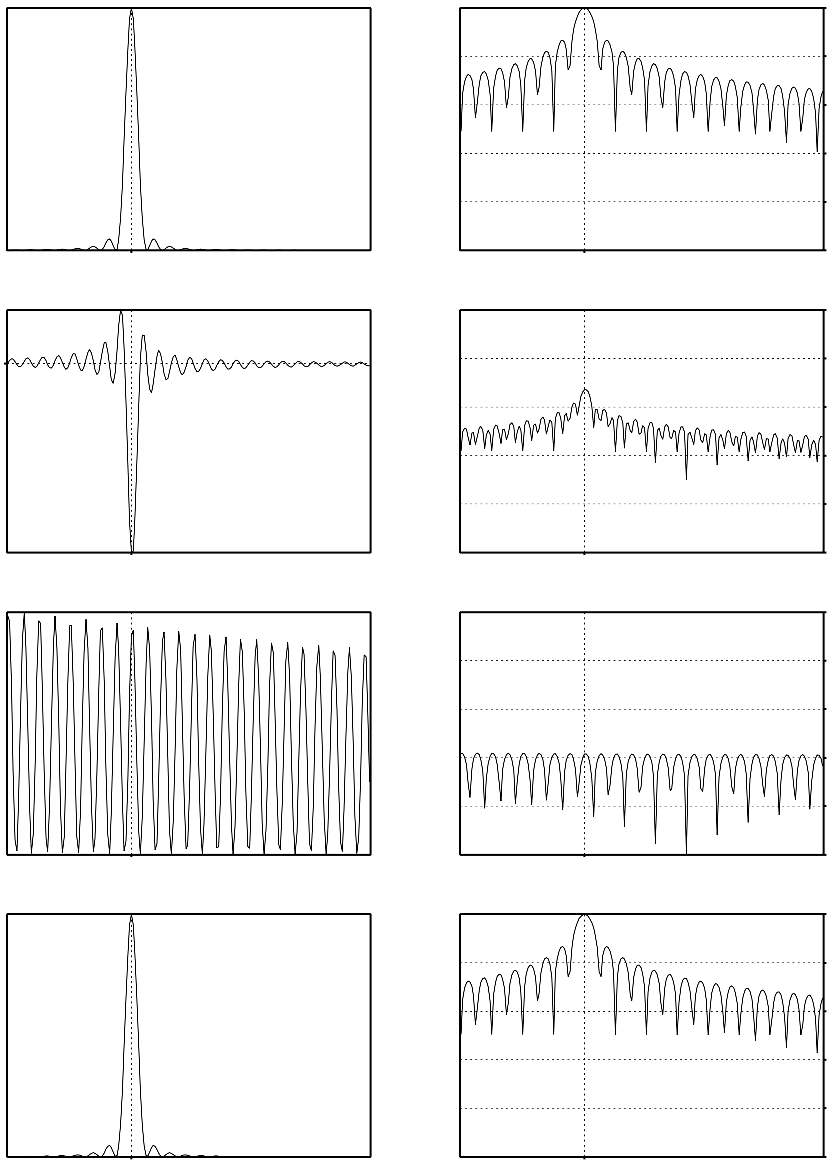


Figure A.30.: The spectral parts of a time-limited tone. For details see text.

B. Advanced scripting

B.1. Procedures

In writing larger scripts you will notice that certain blocks of lines are repeated many times at different locations in the script. For example, if you make a series of tones but the frequencies are not related in such a way that a simple “for loop” could do the job. One way to do this in a “for loop” is by defining arrays of variables like we did in the last part of section 4.7.1. In this section we describe another way, *procedures,* and we introduce *local* variables.

A procedure is a reusable part of a script. Unlike loops, which also contain reusable code, the place where a procedure is defined and the place from which a procedure is *called* differ. A simple example will explain. If you run the following script you will first hear a 500 Hz tone being played, followed by a 600 Hz tone and finally a 750 Hz tone. The first line in the

Script B.1 Use of procedures in scripting.

```
1 @playTone: 500
2 @playTone: 600
3 @playTone; 750
4
5 procedure playTone: frequency
6   Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, frequency, 0.5, 0.01, 0.01
7   Play
8   Remove
9 endproc
```

script *calls* the procedure *named* play_tone with an *argument* of 500.[1] This results in a tone of 500 Hz being played.

In detail: Line 1 directs that the code be continued at line 5 where the procedure play_tone starts. The *variable* `frequency` will be assigned the value 500 and the first line of the procedure, line 6, will be executed. This results in a tone of 500 Hz being created. Lines 7 and 8 will first play and after playing remove the sound. When the endproc line is reached, the execution of the script will return to the end of line 1. The execution of the following line 2 will result in the same sequence of code: the variable `frequency` will be assigned the value 600 and execution of the procedure continues until the endproc line is reached. Then execution will continue at line 3, and the whole cycle starts again. The effect of this script is identical to the following script.

[1] Indeed before praat version 5.3.46 calling a procedure was really preceeded by `call`, i.e. the first line of the script had to be like: `call play_tone 500`. From praat version 5.3.46 this way of calling is deprecated as the new calling syntax guarantees superior argument handling.

```
Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, 500, 0.5, 0.01, 0.01
Play
Remove
Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, 600, 0.5, 0.01, 0.01
Play
Remove
Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, 700, 0.5, 0.01, 0.01
Play
Remove
```

Although this is only a small script, it may be clear that defining a procedure can save a lot of typing: *less typing means less possibility for errors to creep in*. A procedure also makes a nice way to isolate certain portions of the code. You can than test that part more easily and thoroughly. In script B.1 you can test the play_tone procedure,[2] for a couple of frequencies and then it will work for all the other argument frequencies.

In order to use procedures safely we have to make them somewhat more self contained then they are now. In the next section we show how we can do that by the introduction of local variables.

B.1.1. Local variables

Consider the following script:

Script B.2 Interference because of global variables.

```
frequency = 220
@playOctave: frequency
frequency = 2 * frequency
writeInfoLine: "Frequency = ", frequency

procedure playOctave: frequency
  freq = 2 * freq
  Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, frequency, 0.5, 0.01, 0.01
  Play
  Remove
endproc
```

The procedure `play_octave` plays a tone twice as high as its argument. However, in doing so it also changes the variable `frequency`. The consequence is that in line 4 of the script the

[2]For example, one way of testing script B.1 could be that you don't remove the created sound immediately after playing it. Instead, remove line 8, or even better: put a "#" sign in front of line 8 to make it a comment line. Next you call the procedure with two or three different frequency arguments. Each time the procedure is called, a new sound is added to the list of objects. Open the sounds in the sound editor and measure the duration of one period. If you zoom in you will notice that the sound editor shows both the duration of a selected segment as well as its inverse, i.e. frequency. For example the rectangle above the sound that indicates your selection might show "0.002734 (365.784 / s)" which indicates that the selection duration is 0.002734 s and if this were one period of a sound it would correspond to 365.784 periods per second, i.e. a frequency of 365.784 Hz. You can improve the accuracy of your measurement by not taking one period but say ten periods. The advantage of taking ten periods is that you don't have to zoom in as often and that your measurement is more accurate. To obtain the frequency of one period you have to multiply the frequency that is shown in the selection window by ten. Check if this is in agreement with the given frequencies.

variable `frequency` will have the value 880 instead of the expected 440. Consequently this script will print the line `Frequency = 880` in the info window. The variable `frequency` is defined *outside* the procedure but is modified also *inside* the procedure. We say the variable `frequency` has *global scope* in the procedure fragment above. This is not always something desirable. In general we want variables to be confined to the procedure where they are used. This can be arranged by defining special variables that start with a dot. These variables are called *local variables*. Variables without the starting dot are *global variables*. The following script corrects the confusing behavior of the previous script.

Script B.3 Interference variables removed.

```
frequency = 220
@playOctave: frequency
frequency = 2 * frequency
writeInfoLine: "Frequency = ", frequency

procedure playOctave: .frequency
  .frequency = 2 * .frequency
  Create Sound as pure tone: "tone", 1, 0, 0.2, 44100, .frequency, 0.5, 0.01, 0.01
  Play
  Remove
endproc
```

A local variable is, as the naming already suggests, only locally known within the procedure. If you don't use the dot in front of the name, the variable's scope is global and its value may be changed outside the script, or, your script modifies an outside variable. This may create very undesired side effects as we have seen.[3] The modified script will now correctly print `Frequency = 440` because the global variable `frequency` will not interfere with the local variable `.frequency`.

B.2. Selecting objects in a script

An important aspect of working with Praat is its object-oriented interface. Object oriented means that first you have to select the object and then you can do something with it, i.e. *selection precedes action*. Making a selection involves a movement of the mouse to the right object followed by a click. In section 1.5 various ways for the selection of objects were shown. All the various ways of making selection involve action with a mouse and sometimes also the keyboard. From within a script we cannot use a mouse, neither can we use a keyboard. So how do we make a selection from within a script? To solve this problem we need new scripting functions: `selectObject`, `plusObject` and `minusObject`.[4] Equipped with these three new functions we can now refer to objects from within the script itself. In script B.4 we show a number of ways the selection of objects might take place. There are three formats for selecting

[3]As another example of nasty side effects, consider the following situation. A procedure is called from within a `for ifreq to 10` loop. In the procedure the variable `ifreq` is assigned the number 4. In this case the script would never stop because the loop variable will never reach the value 10. . .

[4]These new functions exist since Praat 5.3.48.

Script B.4 Examples of object selection in a script.

```
 1  Create Sound from formula: "s", "Mono", 0, 1, 44100, "0"
 2  selectObject: "Sound s"
 3  Create Sound from formula: "s", "Mono", 0, 1, 44100, "0.5"
 4  selectObject: "Sound s"
 5
 6  s1 = Create Sound from formula: "s", "Mono", 0, 1, 44100, "0"
 7  s1 = selected ("Sound")
 8  s1$ = selected$ ("Sound")
 9  s2 = To Spectrum: "no"
10
11  selectObject: s1
12  plusObject: s2
13  Remove
14
15  # or
16  selectObject: s1, s2
17  Remove
18
19  # or
20  removeObject: s1, s2
```

objects with the `selectObject` function.

1. `selectObject`: "`<Object_type> <object_name>`": selects the object of a given type with a given name. For example in line 2 in script B.4 we use this selection method.
2. `selectObject`: `object_id`: selects an object by its unique number. This is the preferred way to select objects. Lines 11 and 12 show examples.
3. `selectObject ()`: deselects all objects.

Line 1 creates a sound object which bears the name "s", this object is selected in the next line according to the first selection format where `object_type` equals "Sound" and `object_name` equals "s" (in fact this line is not really needed here because, when a new object has been created, it is automatically selected). In the next line another sound object is created which also bears the name "s"; the list of objects will now show two sounds with the same name "s". Line 4 selects a sound with the name "s". But which one of the two sounds will selected? Well, in case of a *name conflict* as we have here, Praat always selects the *most recent one*, i.e. the one at the bottom of the list of objects.

To avoid potential ambiguities in object selections by name, it is always safe to use the second format and select objects by their unique id. Each time Praat creates a new object, it will reserve a unique identification number for the object. This number is visible in the list of objects as for example figure 1.3 shows where a number of objects and their id's are visible. Line 7 in script B.4 shows one way to assign the id of a selected object to a variable. Line 6 shows a much faster way to achieve the same result. Assigning a variable to the create command like we do in line 6, or to the "To Spectrum" command in line 9 saves a lot of typing and is less error prone.

Lines 7 and 8 show how to select the object *id* and object *name* by means of the `selected` the `selected$` functions, respectively. In lines 11, 12 and 13 we show a way to first select an object by id, then extending the selection and finaly removing both objects. In lines 16 and 17 we obtain the same result by using the `selectObject` function with two arguments. Finally the last line shows that we can remove multiple objects together.

B.3. Special words in scripts

The following words are special in a Praat script. Preferably do not use them as the name of a variable.

- `if`, `else`, `elsif`, `endif`, `fi`, `then, or, and, not`
- `for`, `endfor`, `repeat`, `until`, `while`, `endwhile`, `from`
- `form`, `endform`
  - `beginPause, real, positive, integer, natural,`
  - `word, sentence, text, boolean, choice, optionMenu, option`
  - `comment, endPause`
- `procecure`, `endproc`, `call, execute`,
- `selectObject, removeObject, plusobject, minusObject, select`, `plus`, `minus`
- `selected, selected$, numberOfSelected, do, do$`
- `exit, exitScript, runScript, pause`
- `e`, `pi, undefined`
- `appendInFo, writeInfo, appendInfoLine, writeInfoLine`
- `clearInfo, echo`
- `writeFile, appendFile, writeFileLine, appendFileLine`
- `demo, goto, label`
- Within a formula the following have special meaning:
  - `self, self$, xmin, xmax, nx, dx, x1, ymin, ymax, ny, dy, y1`
  - `row, col, row$, col$, nrow, ncol, y, x,`

B.4. Communication outside the script

B.5. File and directory traversals

For applying analysis on a large number of files we need functions to access files and directories, specifically we need functions that can *find* these files for us: in general we do not want to enumerate all the files in a directory by ourselves, we want Praat to find them for us. The following two actions from the New menu can be used to accomplish this: `Create Strings as directory list...` and `Create Strings as file list...`. The first one makes a list of all subdirectories within a certain directory. The second one lists all files within a directory. Both actions return a Strings object that contains the required list. Their syntax is:

1. `Create Strings as directory list: name$, path$`, returns a list of subdirectories in the current `path$` directory.

2. `Create Strings as file list: name$, file_globber$`, returns a list of files that match the specified `file_globber$`. For example, the file globber `/home/david/*.wav` matches all files with extension `.wav` in the directory `/home/david`. Note that a file globber is not the same as a regular expression! In a file globber the "`*`" means any string of characters while in regular expression syntax the "*" character is a quantifier.

In the next section we show how we can use these two actions in a script to recursively descend a directory tree and make a list of all its audio files.

B.5.1. Directory tree traversal in TIMIT

The directory tree traversal script example can be used for the TIMIT acoustic phonetic continuous speech corpus [Lamel et al., 1986]. The TIMIT speech corpus is a speech corpus in which all speech sounds have been labeled with text. All spoken sentences have been labeled, all the words in a sentence have been labeled and all phonemes in all the words have been labeled. This makes it a very valuable research tool because we have access to all the label information. The TIMIT corpus resulted from the joint efforts of several American speech research sites. The text corpus design was done by the Massachusetts Institute of Technology (MIT), Stanford Research Institute and Texas Instruments (TI). The speech was recorded at TI, transcribed at MIT, and has been maintained, verified and prepared for CDROM production by the American National Institute of Standards and Technology (NIST), and was made available via the Linguistic Data Consortium.

The TIMIT corpus contains a total of 6300 sentences. Each of 630 speakers from eight major dialect regions of the United States of America spoke ten sentences. Approximately 70% of the speakers were male and 30% were female. The speaker's dialect region was defined as that geographical area were s/he had lived during childhood years. Dialect number 8 (Army Brat) was assigned to people who had moved around a lot during their childhood and to whom no particular dialect could be attributed.

The ten sentences produced by each speaker consisted of two so-called SA-type, five SX-type and three SI-type sentences.

The *SA sentences* are dialect sentences and were meant to expose the dialectal variants of the speakers. Only two different SA-type sentences were designed, `sa1` and `sa2`, and they were spoken by all 630 speakers. The `sa1` sentence is "She had your dark suit in greasy wash water all year" and the `sa2` sentence is "Don't ask me to carry an oily rag like that".

The *SX sentences* are phonetically-compact sentences and were designed to provide a good coverage of pairs of phones, with extra occurrences of phonetic contexts thought to be either difficult or of particular interest. There were 450 different phonetically compact SX-type sentences, and, given that each speaker produced five of these sentences, each SX sentences was reproduced by seven speakers (= 630 × 5/450). An example of an SX-type sentence is `sx217`, "How permanent are their records?".

The phonetically-diverse *SI sentences* were selected so as to add diversity in sentence types and phonetic contexts. The selection criteria maximized the variety of allophonic contexts found in the texts. Each speaker reproduced three unique utterances out of the 1890 different phonetically diverse SI-type sentences. An example of such an SI-sentence is `si1027`, "Even then, if she took one step forward he could catch her". Taken together there are 2342 (= 2 + 450 + 1890) different sentences in the database.

The file naming and directory structure convention in TIMIT is:

```
<timitroot>/<USAGE>/<DIALECT>/<SEX><SPEAKER>/<SENTENCE>.<FILETYPE>,
```

where `<timitroot>` is the root of the timit directory tree and the other symbols in the path may obtain the following values:

```
<USAGE> := (train|test)
<DIALECT> := dr(1|2|3|4|5|6|7|8)
<SEX> := (m|f) male or female
<SPEAKER> := xyzd (three character followed by digit)
<SENTENCE> := (sa|si|sx)n (sentence number n)
<FILETYPE> := (wav|txt|wrd|phn)
```

For example, if TIMIT lives in `/data/TIMIT`, then the file `/data/TIMIT/test/dr1/fpaz0/si2223.wav` refers to the audio file for sentence `si2223` from the female speaker with initials `paz` from the North Midland dialect `dr3` in the `train` part of the database. Every audio file, with extension `.wav`, is accompanied by three text files with extensions `.txt`, `.wrd` and `.phn`, respectively. These three files are so-called label files, they have segmentation information at the sentence level, the word level and the phone level, respectively. All these files can be read from disk by Praat and displayed. The audio files with extension .wav will, of course, appear as a Sound in the list of objects. The label files when read from disk will be transformed into a TextGrid. As may be clear from now, the number of files in this data base is too large to analyse all by hand and some automatization, i.e. scripting is called for.

Whenever you have to analyze a large number of files like this it is preferable to perform the analysis in two steps. In the first step you select all the files you want to analyze. In the second step you use the selected files to perform the analysis.

In the following two sections we show two different ways to obtain this list files in a directory tree: the explicit way and the system-command way.

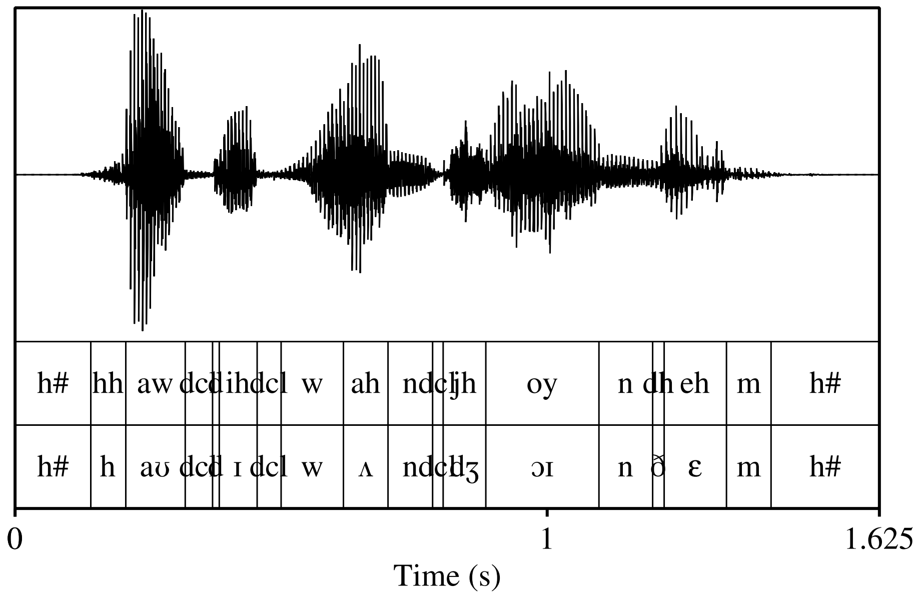


Figure B.1.: The audio and phoneme labels for sentence `si1640` from speaker `mjw0` of dialect region `dr1` in the `test` part of TIMIT.

B.5.1.1. Step 1, method 1: explicit directory traversal

In the following script we show how to traverse the timit directory tree and how to construct a table with all the 6300 file names in it. We use a table to collect file names because a table offers a nice possibility to have the file hierarchy split out and stored into separate columns. It offers the additional advantage that we can sort its contents or make file selections afterwards. The following script B.5 shows a procedure to traverse the TIMIT acoustic-phonetic database directory tree and save the base names of all audio files in a table. Although the following script is rather long, its structure is very simple and straightforward.

The procedure needs only one argument, `.timitroot$`, the location of the TIMIT directory tree. To begin, the procedure creates a summary table with five columns and 6300 entries, i.e. 630 speakers with 10 sentences per speaker.[5] As may be clear from the column names, the first four columns have to be filled with the corresponding part of the file name. The column labeled "usage" will be filled with either "train" or "test", the column labeled "dialect" will contain one of "dr1", "dr2", ..., dr8. The column labeled "speaker" will contain the 630

[5]In case you don't know the number of entries beforehand you have to create a table that is large enough. When you're done filling the table you can scale it down to the right size. For example if you had created a table with 10000 entries, you can use the command `Extract rows where column (text)..."`, `"usage"`, `"matches (regex)"`, `".+"`. This command creates a new table by only copying those rows where the element in the column labeled "usage" is non-empty.

different speaker initials. The fourth "sentence" column will contain one of the 2342 different sentence identifications. Finally, the fifth column, labeled "basepath" is for convenience only: we will fill it with the complete path name minus the file extension. It is constructed by glueing together `.timitroot$` and the contents of the first four columns.[6]

The TIMIT root directory contains three sub directories named train, test and doc of which we only need to process the first two because the doc directory contains documentation files and no audio or label files. We define two array variables to use in the processing of the train and the test directories. We use a for-loop that extends from line 8 to line 53. At the first pass of this loop, when the loop variable `.idir1` equals 1, the variable `.dir1$` attains the string value "train" (line 10). If for example `.timitroot$` equals "/data/TIMIT", then in the next line the variable `.path1$` will be set to the full path name of this directory: "/data/TIMIT/train". This directory is queried for its sub directories (line 12) and this list becomes available as a Strings object. The number of strings in this object will be equal to the number of sub directories in the train directory (the eight dialect sub directories dr1, dr2, … dr8). The next for-loop (line 15) now queries all these sub directories for their sub directories (a variable number of speaker directories, i.e. there will be more directories in the train part than in the test part). We are now at the lowest directory level in the directory tree and we query each directory for all files that end with ".wav". In the most inner loop (line 32 to line 44), we process all the audio files in one directory. In the first two lines we get a file name from the Strings object. We then remove the final ".wav" part of the name. Line 36 constructs the path name of the file. We now have all the information to fill the corresponding row in the table. We are done with this file now, we increase the row number and process the next file in the current directory. This goes on until we have processed all (10) wav-files in this speaker directory. We then proceed with a new speaker until we are done with all the speakers in the dialect directory. We then process another dialect, etc. Finally all directories and all files have been processed and we and up with a table where for all 6300 rows all five columns have the correct information.

B.5.1.2. Step 1, method 2: system command directory traversal

Sometimes shell commands from the computer's operating system can be used to get things done. Especially on Unix-based systems the shell has very powerful commands available. On MacOS X and Linux systems we can use the "find" command to recursively list all files in a directory tree. For example, the shell command "`find /data/TIMIT/ -name \*.wav > timit_file_list`" produces a file named "timit_file_list" with the complete file names of all 6300 audio files. This is much simpler than an explicit recursive descend of all the directories like we did in the previous section.

Windows also has the possiblity to recursively descent a directory tree in its command shell window. If TIMIT lives in directory E:\TIMIT, for example, we use `dir E:\TIMIT\*.wav /B /S > timit_file_list` to obtain the complete file list. To get access to these shell commands in a Praat script we have to precede these shell commands with the `system` command. The following script B.6 uses the shell command to arrive at the same table as in the previous section.

[6]We want to use the table also to be able to read all the label files. In this case we only need to append the specific extension to "basepath". This is a lot simpler than first removing the .wav extension and then appending the new extension.

Script B.6 Using a system command to traverse a directory tree.

```
procedure table_from_filelist_timit2 (.root$)
  .fsep$ = "/"
  .find$ = "find " + .root$ + " -name \*.wav"
  if windows
    .fsep$ = "\\"
    .find$ = "dir " + .root$ + "\*.wav /B /S"
  endif
  system '.find$' > timit_file_list
  .strings = Read Strings from raw text file: "timit_file_list"
  .nstrings= Get number of strings
  .table = Create Table with column names: "timit", 6300,
    ... "usage dialect speaker sentence basepath"
  .rootlen = length (.root$)
  for .is to .nstrings
    selectObject: .strings
    .path$ = Get string: .is
     # <root>/(test|train)/dr(1..8)/xxxx/*.wav
    .pathlen = length (.path$)
    .basepath$ = left$ (.path$, .pathlen - 4)
    .pathlen -= .rootlen
    .path$ = right$ (.path$, .pathlen)
    for .ip to 3
      .sep = index (.path$, .fsep$)
      .part$[.ip] = left$ (.path$, .sep - 1)
      .pathlen -= .sep
      .path$ = right$ (.path$, .pathlen)
    endfor
    # wav
    .part$[4] = left$ (.path$, .pathlen - 4)
    selectObject: .table
    Set string value: .is, "usage", .part$[1]
    Set string value: .is, "dialect", .part$[2]
    Set string value: .is, "speaker", .part$[3]
    Set string value: .is, "sentence", .part$[4]
    Set string value: .is, "basepath", .basepath$
  endfor
  removeObject: .strings
  selectObject: .table
endproc
```

We start this script by defining the file separator as the slash '/' character. For Windows the file separator is the backslash '\'. Next we use the find system command to get a file with all the file names. We read this file as a Strings object: each line in the file becomes a separate item in the Strings, i.e. a file name. We strip of the first part of each string, the root part. In the inner loop we successively strip the parts before a file separator character. Finally, we strip the last four characters ".wav" from the remaining part.

The script explicitly uses that audio files only exist at the lowest level of the tree.

B.5.1.3. Step 2: analysis script global structure

Once we have a list with filenames we can proceed with the second step: the analysis of these files. Maybe we do not want to analyse all the files at once but make a selection. For example, we might be interested in the data from the male speakers only and use "`do ("Extract rows where column (text)...", "speaker", "starts with", "m`") to get a new smaller table. Or we might be interested in the sheboleth sentences only "`do ("Extract rows where column (text)...", "sentence", "starts with"`, "sa"). The most elementary analysis script will process the items in the table successively. For example, suppose your TIMIT database lives in "/data/TIMIT/", then you may use the directory-traversal procedure as is shown in one the skeleton scripts B.7 and B.6.

The first line calls the directory-traversal procedure in script B.6. This results in the well known table with 6300 file name entries. Its number of rows are queried and then we loop over all rows from the table. We extract a string with the base part of the filename from the table cell at the row with index "ifile" and the column with label "basepath" and appends the ".wav" extension to this string. The complete filename is known and the sound file can be read from disk. We then can do some processing on this sound and, finally, when we are done with the sound, we remove it.

B.5.1.4. Step 2: formant analysis

Now, suppose we want to measure formant frequencies for all the vowels in the TIMIT database. In section B.5.1.1 we created a script to traverse the directory tree. We will use it here and provide a skeleton script to perform a rudimentary formant frequency analysis on all the vowels in the database. We will measure the first three formant frequencies at three positions in the vowel, at 20%, 50% and 80% of its duration. In this way we will have information for monophthongs and diphthongs as well. The following script will perform the analysis.

Script B.8 Formant analysis in TIMIT.

```
procedure timit_formant_analysis (.table)
    .vowels$ = " iy ih eh ey ae aa aw ay ah ao oy ow uh uw ux er ax ix axr ax-h "
    .results = Create Table with column names: "results", 100000,
        ... "usage dialect speaker sentence vowel tb te
        ... f11 f21 f31 f12 f22 f32 f13 f23 f33"
    selectObject: .table
    .nfiles = Get number of rows
    .irow = 1
    for .ifile to .nfiles
        .usage$ = Object_'.table'$[.ifile, "usage"]
        .dialect$ = Object_'.table'$[.ifile, "dialect"]
        .speaker$ = Object_'.table'$[.ifile, "speaker"]
        .sentence$ = Object_'.table'$[.ifile, "sentence"]
        .wav$ = Object_'.table'$[.ifile, "basepath"] + ".wav"
        .phn$ = Object_'.table'$[.ifile, "basepath"] + ".phn"
        nowarn Read from file: .phn$
        .textgrid = selected ("TextGrid")
        .sound = Read from file: .wav$
        .gender$ = left$ (.speaker$, 1)
        .max_formant = 5500
        if .gender$ = "m"
            .max_formant = 5000
        endif
        noprogress To Formant (burg): 0.05, 5, .max_formant, 0.025, 50
        .formant = selected ("Formant")
        selectObject: .textgrid
        .ninter = Get number of intervals: 1
        for .inter to .ninter
            selectObject: .textgrid
            .label$ = Get label of interval: 1, .inter
            .index = index (.vowels$, " " + .label$ + " ")
            if .index > 0
                .tb = Get start point: 1, .inter
                .te = Get end point: 1, .inter
                .dur = .te - .tb
                .t[1] = .tb + 0.2 * .dur
                .t[2] = .tb + 0.5 * .dur
                .t[3] = .tb + 0.8 * .dur
                selectObject: .formant
                for .it to 3
                    for .ifor to 3
                        .f[.ifor,.it] = Get value at time: .ifor, .t[.it], "Hertz", "Lin
                    endfor
                endfor
                # results
                selectObject: .results
                Set string value: .irow, "usage", .usage$
                Set string value: .irow, "dialect", .dialect$
                Set string value: .irow, "speaker", .speaker$
                Set string value: .irow, "sentence", .sentence$
                Set string value: .irow, "vowel", .label$
                Set numeric value: .irow, "tb", .tb
                Set numeric value: .irow, "te", .te
                for .it to 3
                    for .ifor to 3
                        .flabel$ = "f" + string$ (.ifor) + string$ (.it)
                        Set numeric value: .irow, .flabel$, .f[.ifor,.it]
                    endfor
                endfor
                .irow += 1
            endif
        endfor
        removeObject: .textgrid, .sound, .formant
    endfor
    selectObject: .results
    .trimmed = Extract rows where column (text): "usage", "matches (regex)", ".+"
    removeObject: .results
    Rename: "results"
endproc
```

The analysis procedure will have one argument, the table with file names. We start with defining the vowels that we like to analyze. We have selected all twenty as defined in the database. We might as well analyze them all and perhaps later from the analysis results select the ones we are interested in.[7] Each vowel in this string is surrounded by white space. This makes it easy to test for its occurrence as happens later in the script at line 31. Next we create a table to store the results of the analysis. We hopefully supply enough rows by creating 100000 of them initially (and later trimming it down at line 66). Because we want to know where the vowel came from, we supply some the "usage", "dialect", "speaker" and "sentence" columns. The identity of the vowel is stored in the "vowel" column while in the "tb" and "te" columns the starting and ending times of the vowel interval in the sound will be stored. The next nine columns are reserved for the first, second and third formant frequencies at respectively at 20%, 50% and 80% of its duration (for example, "f12" is the first formant frequency at the middle of the vowel).

Now we come to the first loop (line 9) in which we analyze each file from the table. In the next lines we read out the directory info which will also be stored in the results table later. Lines 14 and 15 construct the complete file paths for the sound and the label file. We read the label file with the `nowarn` keyword at the start of the line. This keyword suppresses warnings about inconsistencies in some of the label files that would otherwise pop up. The phonetic label file will be read and converted automatically to a TextGrid object. This TextGrid will have two tiers, the first tier contains the original labels from the phn-file while the second tier contains its IPA translation. In lines 19 to 23 we determine the maximum formant frequency, depending on whether the sound file is from a male or a female speaker. We test the first character of the speaker initials which is either "m" or "f". For a female speaker the maximum formant frequency is set to 5500 Hz, for a male speaker it is set to 5000 Hz. The actual formant analysis is then performed and we suppress the progress bar that would pop up for every file. The time step we use for the formant analysis is five milliseconds and we search for five formants.

Line 28 starts the interesting part, a loop over all the labels in the first tier of the TextGrid. Line 30 gets a label from tier 1 and in the next line we try to find if it is one of the labels in the `vowels$` string. We explicitly put white space around the tier's label to be sure we really match a vowel from the string (otherwise "ax" would match with "ax", "axr" and "ax-h", for example). If the label matches a vowel we process this vowel in lines 33 to 62. This part is only bookkeeping. We start with querying the start and end point of the vowel interval. These times, `.tb` and `.te`, are then used to calculate the three times we need at 20%, 50% and 80% of the interval's duration. Once these three times are known we query the formant object for its values and we store these in local variables. Now all the things we want to know for this vowel are known and we can store the values (lines 46 to 60). Once all values are stored we increase the row number and proceed with the next label. When all the labels are processed we proceed to the next file, and so on.

Finally after all 6300 files have been processed and their analysis data stored in the results table we trim it down by extracting only those rows that have some text in it (actually it boils down to 78374 rows). Line 66 takes care of this: the regular expression syntax "`.+`" means

[7]This is in general a good idea: analyze everything and select afterwards. Reading the file and formant frequency analysis itself will, in general, use the major part of the analysis time. Storing somewhat more than we need only costs some memory. And we have gigabytes of them…

“one or more characters”. Therefore, we only extract those rows from the table when there is at least one character in the usage column.

Script B.5 Script for directory traversal in TIMIT.

```
procedure timit_basename_table_create: .timitroot$
    .table = Create Table with column names: "timit", 6300,
        ... "usage dialect speaker sentence basepath"

    .usage$[1] = "train"
    .usage$[2] = "test"
    .irow =  1
    for .idir1 to 2
        # test|train level
        .dir1$ = .usage$[.idir1]
        .path1$ = .timitroot$ + "/" + .dir1$
        .dirList1 = Create Strings as directory list:
            ... "dirList1",  .path1$
        .dirList1_n = Get number of strings

        for .idir2 to .dirList1_n
            # dr1 .. dr8
            selectObject: .dirList1
            .dir2$ = Get string: .idir2
            .path2$ = .path1$ + "/" + .dir2$
            .dirList2 = Create Strings as directory list:
                ... "dirList2", .path2$
            .dirList2_n = Get number of strings
            for .idir3 to .dirList2_n
                # speakers
                selectObject: .dirList2
                .dir3$ = Get string: .idir3
                .path3$ = .path2$ + "/" + .dir3$
                .fileList = Create Strings as file list:
                    ... "fileList", .path3$ + "/*.wav"
                .fileList_n = Get number of strings
                for .ifile to .fileList_n
                    selectObject: .fileList
                    .file$ = Get string: .ifile
                    .base$ = .file$ - ".wav$"
                    .pathc$ = .path3$ + "/" +.base$
                    selectObject (.table)
                    Set string value: .irow, "usage", .dir1$
                    Set string value: .irow, "dialect", .dir2$
                    Set string value: .irow, "speaker", .dir3$
                    Set string value: .irow, "sentence", .base$
                    Set string value: .irow, "basepath", .pathc$
                    .irow += 1
                endfor
                removeObject: .fileList
            endfor
            removeObject: .dirList2
        endfor
        removeObject: .dirList1
    endfor
    selectObject: .table
endproc
```

Script B.7 Example skeleton script that uses the procedure in script B.5 to process all audio files in TIMIT.

```
@timit_basename_table_create2 ("/data/TIMIT")
filetable = selected ("Table")
<selections>
nfiles = Get number of rows
for ifile to nfiles
  selectObject: filetable
  wavfile$ = Object_'filetable'$[ifile, "basepath"] + ".wav"
  sound = Read from file: wavfile$
  <do the processing...>
  removeObject: sound
endfor
```

C. Scripting syntax

C.1. Variables

Variable names start with a lowercase letter and are case sensitive, i.e. ‘aBc’ and ‘abc’ are not the same variable. String variables end with a ‘$’, numeric variables don’t.

Examples: `length = 17.0`, `text$ = "some words"`

C.1.1. Predefined variables

data$ current date and time

e the number e=2.718...

macintosh equals 1 if scripts runs on Macintosh computer else it is 0

windows equals 1 if scripts runs on Windows computer else it is 0

unix equals 1 if scripts runs on Unix platform computer else it is 0

pi the number 3.14159...

tab$ the tab character

newline$ the newline character(s)

info$ the contents of the info window

shellDirectory$ default directory when Praat started

homeDirectory$ the directory where you log in

preferencesDirectory$ directory where your preferences are stored

temporaryDirectory$ a directory for saving temporary files

defaultDirectory$ the directory that contains the script

praatVersion$ the current version of Praat as text (e.g. "5.3.77")

praatVersion the current version of Praat as a number (e.g. 5377)

C.2. Conditional expressions

```
if conditional
    <statements>
endif

# somewhat more complex

if conditional
    <statements_1>
else
    <statements_2>
endif

# 2 conditionals are tested

if conditional_1
    <statements_1>
elsif conditional_2
    <statements_2>
else
    <statements_3>
endif

# test more conditionals

if conditional_1
    <statements_1>
elsif conditional_2
    <statements_2>
...
elsif conditional_n
    <statements_n>
else
    <statements_3>
endif
```

Examples:

```
if age < 3
  writeInfoLine: "Younger than 3"
elsif age < 12
  writeInfoLine: "Younger than 12"
elsif age < 20
  writeInfoLine: "Younger than 20"
else
  writeInfoLine: "Older than 20"
endif
```

C.3. Loops

C.3.1. Repeat until loop

```
repeat
    <statements>
```

```
until expression
```

Repeats executing the *statements* between repeat and the matching until line as long as the evaluation of *expression* does not return zero or false.

C.3.2. While loop

```
while expression
    <statements>
endwhile
```

Repeats executing the *statements* between the while and the matching endwhile as long as the evaluation of *expression* does not return zero or false.

C.3.3. For loop

```
for variable [from expression_1] to expression_2
    <statements>
endfor
```

If `expression_1` evaluates to 1, the part between the [and the] can be left out as in:

```
for variable to expression_2
    <statements>
endfor
```

The semantics of the first for loop are equivalent to the following while loop:

```
variable = expression_1
while variable <= expression_2
    <statements>
    variable = variable + 1
endwhile
```

C.3.4. Procedures

Define a procedure:

```
procedure nameOfTheProcedure: .argument1, ..., .argument_n
    <do something with the arguments>
endproc

# call the procedure:

@nameOfTheProcedure: arg1, ..., arg_n
```

C.3.5. Executing Praat commands

Copy name from button and fields of the form:

```
# preferred syntax: end command with ":" and separate arguments with a comma ","
# "text" between double quotes

Create Sound from formula: "name", 1, 0, 1, 44100, "sin(2*pi*500*x)"
```

```
Play

# old syntax (deprecated, very wordy):
do ("Create Sound from formula...", "name", 1, 0, 1, 44100, "sin(2*pi*500*x)")
do ("Play")

# even older (deprecated): no comma’s between arguments
Create Sound from formula... name 1 0 1 44100 sin(2*pi*500*x)
Play
```

D. Terminology

ADC An Analog to Digital Converter converts an analog electrical signal into a series of numbers.

ADPCM A variant of DPCM that varies the size of the quantization step.

Aliasing the ambiguity of a sampled signal. See section 3.6.4 on analog to digital conversion.

Bandwidth The bandwidth of a sound is the difference between the highest frequency in the sound and the lowest frequency in the sound. The bandwidth of a filter is the difference between the two frequencies where the

Big-endian The big-end is stored first. See endianness.

DAC A Digital to Analog Converter transform a series of numbers to an analog electrical signal.

DPCM A variant of PCM, encodes PCM values as differences between the current and the predicted value.

Endianness. Refers to the way things are ordered in computer memory. An entity that consists of 4 bytes, say the number $0x0A0B0C0D$ in hexadecimal notation is stored in the memory of big-endian hardware in four consecutive bytes with contents $0x0A$, $0x0B$, $0xCA$, and $0x0D$, respectively. In little-endian hardware this 4-byte entity will be stored in four consecutive bytes as $0x0D$, $0xCA$, $0x0B$ and $0x0A$. A vague analogy would be in the representation of dates: yyyy-mm-dd would be big-endian, while dd-mm-yyyy would be little-endian.

Little-endian The little-end is stored first. See endianness.

Nyquist frequency. The bandwidth of a sampled signal. The Nyquist frequency equals half the sampling frequency of the signal. If the sampled signal represents a continuous spectral range starting at 0 Hz, which is normally the case for sound recordings, the Nyquist frequency is the highest frequency that can be represented unambiguously by the sampled signal.

PCM Pulse Code modulation, a digital representation of a sound. The amplitude is sampled at a constant rate and always quantized with the same number of bits.

Sensitivity of an electronic device is the minimum magnitude of the input signal required to produce a specified output signal. For the microphone input of a soundcard, for example, it is that voltage that provides the maximum allowed voltage that the ADC accepts if the input volume control is set to its maximum. Generally the sensitivity levels are mentioned in the specifications of all audio voltage accepting equipment.

S/PDIF Sony/Philips Digital Interconnect Format, a version of the AES/EBU format for digital audio connections for consumer sound cards.

Transducer a device that converts one type of energy to another. A microphone converts acoustic energy to electric energy while the reverse process is accomplished by a speaker. A light bulb is another transducer, it converts electrical energy into light.

Bibliography

Patti Adank, Roeland Van Hout, and Roel Smits. An acoustic description of the vowels of Northern and Southern Standard Dutch. *J. Acoust. Soc. Am.*, 116:1729–1738, 2004.

Paul Boersma. Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. *Proc. Institute of Phonetic Sciences University of Amsterdam*, 17:97–110, 1993.

D.G. Childers. *Modern spectrum analysis*. IEEE Press, 1978.

G. Fant. *The Acoustic Theory of Speech Production*. Mouton, The Hague, 1960.

Jonathan Harrington and Steve Cassidy. *Techniques in Speech Acoustics*. Kluwer Academic Publishers, 1999.

Toshio Irino and Roy D. Patterson. A time-domain, level-dependent auditory filter: the gammachirp. *J. Acoust. Soc. Am.*, 101(1):412–419, 1997.

Keith Johnson. *Acoustic and Auditory Phonetics*. Blackwell, 1997. ISBN 0-631-20095-9.

Dennis H. Klatt. Software for a cascade/parallel formant synthesizer. *J. Acoust. Soc. Am.*, 67: 971–995, 1980.

Dennis H. Klatt and Laura C. Klatt. Analysis, synthesis, and perception of voice quality variations among female and male talkers. *J. Acoust. Soc. Am.*, 87:820–857, 1990.

Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1998.

W. Koenig, H.K. Dunn, and L.Y. Lacey. The sound spectrograph. *J. Acoust. Soc. Am.*, 18: 19–49, 1946.

L. F. Lamel, R. H. Kassel, and S. Seneff. Speech database development: Design and analysis of the acoustic-phonetic corpus. In *Proc. DARPA Speech Recognition Workshop*, pages 100–109, 1986.

Chin-Hui Lee. On robust linear prediction of speech. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 36:642–649, 1988.

J. Makhoul. Linear prediction: A tutorial review. *Proc. IEEE*, 63:561–580, 1975.

J. D. Markel and A. H. Gray, Jr. *Linear prediction of speech*. Springer Verlag, Berlin, 1976.

Athanasios Papoulis. *Signal analysis*. McGraw-Hill, 1988.

G. E. Peterson and H. L. Barney. Control methods used in a study of the vowels. *J. Acoust. Soc. Am.*, 24:175–184, 1952.

Louis C. W. Pols, H. R. C. Tromp, and Reinier Plomp. Frequency analysis of Dutch vowels from 50 male speakers. *J. Acoust. Soc. Am.*, 53:1093–1101, 1973.

Rollin Rachelle. *Overtone Singing Study Guide*. Cryptic Voices Productions, Amsterdam, 1995.

K. Saberi and D. R. Perrot. Cognitive restoration of reversed speech. *Nature*, 398:760, 1999.

Hiroaki Sakoe and Seibi Chiba. Dynamic programming optimization for spoken word recognition. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 26:43–49, 1978.

Kenneth N. Stevens. *Acoustic phonetics*. MIT Press, 2nd edition, 2000.

Stanley Smith Stevens, John Volkman, and Edwin B. Newman. A scale for the measurement of the psychological magnitude pitch. *J. Acoust. Soc. Am.*, 8:185–190, 1937.