Chapter 1

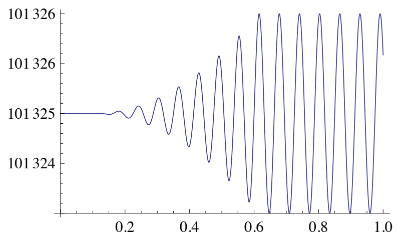
Sound

A major part of the information we receive and perceive every day is in the form of audio. Most of these sounds are transferred directly from the source to our ears, like when we have a face to face conversation with someone or listen to the sounds in a forest or a street. However, a considerable part of the sounds are generated by loudspeakers in various kinds of audio machines like cell phones, digital audio players, home cinemas, radios, television sets and so on. The sounds produced by these machines are either generated from information stored inside, or electromagnetic waves are picked up by an antenna, processed, and then converted to sound. It is this kind of sound we are going to study in this chapter. The sound that is stored inside the machines or picked up by the antennas is usually represented as *digital sound*. This has certain limitations, but at the same time makes it very easy to manipulate and process the sound on a computer.

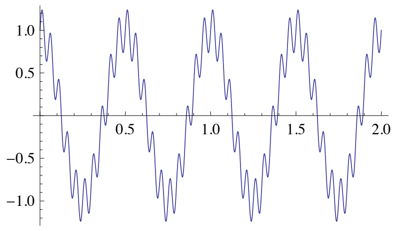
What we perceive as sound corresponds to the physical phenomenon of slight variations in air pressure near our ears. Larger variations mean louder sounds, while faster variations correspond to sounds with a higher pitch. The air pressure varies continuously with time, but at a given point in time it has a precise value. This means that sound can be considered to be a mathematical function.

Observation 1.1. A sound can be represented by a mathematical function, with time as the free variable. When a function represents a sound, it is often referred to as a *continuous signal*.

In the following we will briefly discuss the basic properties of sound: first the significance of the size of the variations, and then how many variations there are per second, the *frequency* of the sound. We also consider the important fact that any reasonable sound may be considered to be built from very simple basis sounds. Since a sound may be viewed as a function, the mathematical equivalent of this is that any decent function may be constructed from very simple basis functions. Fourier-analysis is the theoretical study of this, and in the next chapters we are going to study this from a practical and computational



(a) A sound shown in terms of air pressure



(b) A sound shown in terms of the difference from the ambient air pressure

Figure 1.1: Two examples of audio signals.

perspective. Towards the end of this chapter we also consider the basics of digital audio, and illustrate its power by performing some simple operations on digital sounds.

1.1 Loudness: Sound pressure and decibels

An example of a simple sound is shown in Figure 1.1(a) where the oscillations in air pressure are plotted agains time. We observe that the initial air pressure has the value 101 325 (we will shortly return to what unit is used here), and then the pressure starts to vary more and more until it oscillates regularly between the values 101 323 and 101 327. In the area where the air pressure is constant, no sound will be heard, but as the variations increase in size, the sound becomes louder and louder until about time $t = 0.6$ where the size of the oscillations becomes constant. The following summarises some basic facts about air pressure.

Fact 1.2 (Air pressure)**.** Air pressure is measured by the SI-unit Pa (Pascal) which is equivalent to N/m^2 (force / area). In other words, 1 Pa corresponds to the force exerted on an area of 1 m^2 by the air column above this area. The normal air pressure at sea level is 101 325 Pa.

Fact 1.2 explains the values on the vertical axis in Figure 1.1(a): The sound was recorded at the normal air pressure of 101 325 Pa. Once the sound started, the pressure started to vary both below and above this value, and after a short transient phase the pressure varied steadily between 101 324 Pa and 101 326 Pa, which corresponds to variations of size 1 Pa about the fixed value. Everyday sounds typically correspond to variations in air pressure of about 0.00002–2 Pa, while a jet engine may cause variations as large as 200 Pa. Short exposure to variations of about 20 Pa may in fact lead to hearing damage. The volcanic eruption at Krakatoa, Indonesia, in 1883, produced a sound wave with variations as large as almost 100 000 Pa, and the explosion could be heard 5000 km away.

When discussing sound, one is usually only interested in the variations in air pressure, so the ambient air pressure is subtracted from the measurement. This corresponds to subtracting 101 325 from the values on the vertical axis in Figure 1.1(a). In Figure 1.1(b) the subtraction has been performed for another sound, and we see that the sound has a slow, cos-like, variation in air pressure, with some smaller and faster variations imposed on this. This combination of several kinds of systematic oscillations in air pressure is typical for general sounds. The size of the oscillations is directly related to the loudness of the sound. We have seen that for audible sounds the variations may range from 0.00002 Pa all the way up to 100 000 Pa. This is such a wide range that it is common to measure the loudness of a sound on a logarithmic scale. Often air pressure is normalized so that it lies between -1 and 1: The value 0 then represents the ambient air pressure, while -1 and 1 represent the lowest and highest representable air pressure, respectively. The following fact box summarises the previous discussion of what a sound is, and introduces the logarithmic decibel scale.

Fact 1.3 (Sound pressure and decibels)**.** The physical origin of sound is variations in air pressure near the ear. The *sound pressure* of a sound is obtained by subtracting the average air pressure over a suitable time interval from the measured air pressure within the time interval. A square of this difference is then averaged over time, and the sound pressure is the square root of this average.

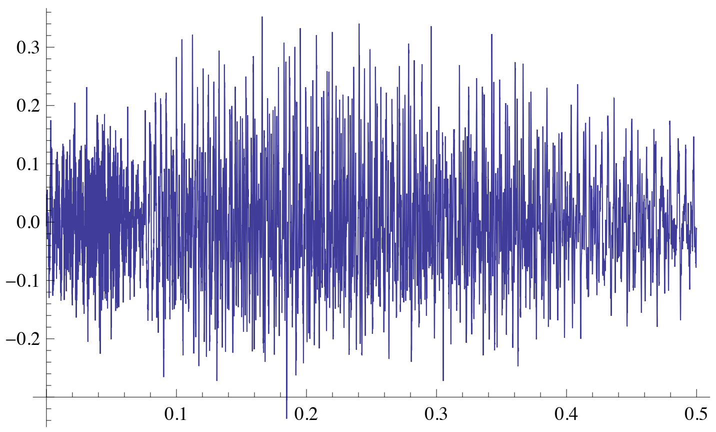
It is common to relate a given sound pressure to the smallest sound pressure that can be perceived, as a level on a decibel scale,

$$L_p = 10\log_{10}\left(\frac{p^2}{p_{\mathrm{ref}}^2}\right) = 20\log_{10}\left(\frac{p}{p_{\mathrm{ref}}}\right).$$

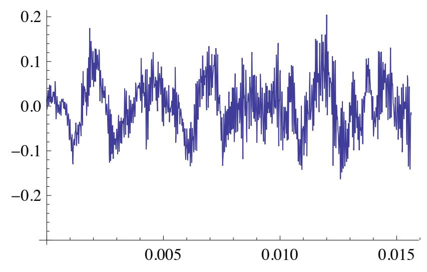
Here p is the measured sound pressure while p_{ref} is the sound pressure of a just perceivable sound, usually considered to be 0.00002 Pa.

The square of the sound pressure appears in the definition of L_p since this represents the *power* of the sound which is relevant for what we perceive as loudness.

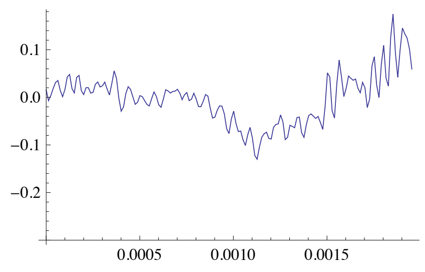
The sounds in Figure 1.1 are synthetic in that they were constructed from mathematical formulas (see Exercises 1.4.2 and 1.4.3). The sounds in Figure 1.2 on the other hand show the variation in air pressure when there is no mathematical formula involved, such as is the case for a song. In (a) there are so many oscillations that it is impossible to see the details, but if we zoom in as in (c) we can see that there is a continuous function behind all the ink. It is important to realise that in reality the air pressure varies more than this, even over the short time period in (c). However, the measuring equipment was not able to pick up those variations, and it is also doubtful whether we would be able to perceive such rapid variations.



(a) 0.5 seconds of the song



(b) the first 0.015 seconds



(c) the first 0.002 seconds

Figure 1.2: Variations in air pressure during parts of a song.

1.2 The pitch of a sound

Besides the size of the variations in air pressure, a sound has another important characteristic, namely the frequency (speed) of the variations. For most sounds the frequency of the variations varies with time, but if we are to perceive variations in air pressure as sound, they must fall within a certain range.

Fact 1.4. For a human with good hearing to perceive variations in air pressure as sound, the number of variations per second must be in the range 20–20 000.

To make these concepts more precise, we first recall what it means for a function to be periodic.

Definition 1.5. A real function f is said to be periodic with period τ if

$$f(t+\tau) = f(t)$$

for all real numbers t.

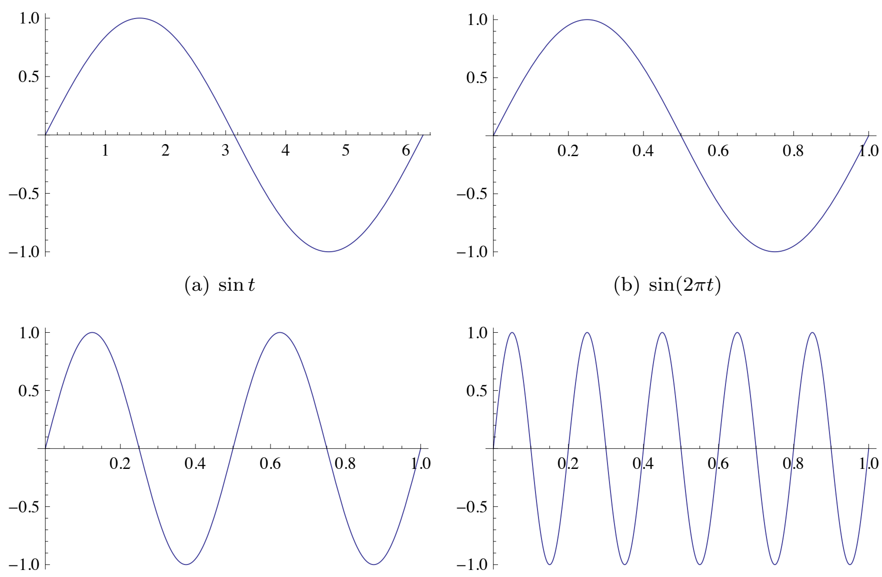
Note that all the values of a periodic function f with period τ are known if $f(t)$ is known for all t in the interval $[0,\tau)$. The prototypes of periodic functions are the trigonometric ones, and particularly $\sin t$ and $\cos t$ are of interest to us. Since $\sin(t+2\pi) = \sin t$, we see that the period of $\sin t$ is 2π and the same is true for $\cos t$.

There is a simple way to change the period of a periodic function, namely by multiplying the argument by a constant.

Observation 1.6 (Frequency)**.** If ν is an integer, the function $f(t) = \sin(2\pi\nu t)$ is periodic with period $\tau = 1/\nu$. When t varies in the interval $[0,1]$, this function covers a total of ν periods. This is expressed by saying that f has *frequency* ν.

Figure 1.3 illustrates observation 1.6. The function in (a) is the plain $\sin t$ which covers one period when t varies in the interval $[0,2\pi]$. By multiplying the argument by 2π, the period is squeezed into the interval $[0,1]$ so the function $\sin(2\pi t)$ has frequency $\nu = 1$. Then, by also multiplying the argument by 2, we push two whole periods into the interval $[0,1]$, so the function $\sin(2\pi 2t)$ has frequency $\nu = 2$. In (d) the argument has been multiplied by 5 — hence the frequency is 5 and there are five whole periods in the interval $[0,1]$. Note that any function on the form $\sin(2\pi\nu t + a)$ has frequency ν, regardless of the value of a.

Since sound can be modelled by functions, it is reasonable to say that a sound with frequency ν is a trigonometric function with frequency ν.



(a) $\sin t$ (b) $\sin(2\pi t)$

(c) $\sin(2\pi 2t)$ (d) $\sin(2\pi 5t)$

Figure 1.3: Versions of sin with different frequencies.

Definition 1.7. The function $\sin(2\pi\nu t)$ represents what we will call a pure tone with frequency ν. Frequency is measured in Hz (Herz) which is the same as s^{-1} (the time t is measured in seconds).

A pure tone with frequency 440 Hz sounds like this, and a pure tone with frequency 1500 Hz sounds like this.

Any sound may be considered to be a function. In the next chapter we are going to see that any reasonable function may be written as a sum of simple sin- and cos- functions with integer frequencies. When this is translated into properties of sound, we obtain an important principle.

Observation 1.8 (Decomposition of sound into pure tones)**.** Any sound f is a sum of pure tones at different frequencies. The amount of each frequency required to form f is the frequency content of f. Any sound can be reconstructed from its frequency content.

The most basic consequence of observation 1.8 is that it gives us an understanding of how any sound can be built from the simple building blocks of pure tones. This means that we can store a sound f by storing its frequency content, as an alternative to storing f itself. This also gives us a possibility for lossy compression of digital sound: It turns out that in a typical audio signal there will be most information in the lower frequencies, and some frequencies will be almost completely absent. This can be exploited for compression if we change the frequencies with small contribution a little bit and set them to 0, and then store the signal by only storing the nonzero part of the frequency content. When the sound is to be played back, we first convert the adjusted values to the adjusted frequency content back to a normal function representation with an inverse mapping.

Fact 1.9 (Basic idea behind audio compression)**.** Suppose an audio signal f is given. To compress f, perform the following steps:

1. Rewrite the signal f in a new format where frequency information becomes accessible.
2. Remove those frequencies that only contribute marginally to human perception of the sound.
3. Store the resulting sound by coding the adjusted frequency content with some lossless coding method.

This lossy compression strategy is essentially what is used in practice by commercial audio formats. The difference is that commercial software does everything in a more sophisticated way and thereby gets better compression rates. We will return to this in later chapters.

We will see later that Observation 1.8 also is the basis for many operations on sounds. The same observation also makes it possible to explain more precisely what it means that we only perceive sounds with a frequency in the range 20–20000 Hz:

Fact 1.10. Humans can only perceive variations in air pressure as sound if the Fourier series of the sound signal contains at least one sufficiently large term with frequency in the range 20–20 000 Hz.

With appropriate software it is easy to generate a sound from a mathematical function; we can 'play' the function. If we play a function like $\sin(2\pi 440t)$, we hear a pleasant sound with a very distinct pitch, as expected. There are, however, many other ways in which a function can oscillate regularly. The function in Figure 1.1(b) for example, definitely oscillates 2 times every second, but it does not have frequency 2 Hz since it is not a pure tone. This sound is also not that pleasant to listen to. We will consider two more important examples of this, which are very different from smooth, trigonometric functions.

Example 1.11. We define the *square wave* of period T as the function which repeats with period T, and is 1 on the first half of each period, and -1 on the second half. This means that we can define it as the function

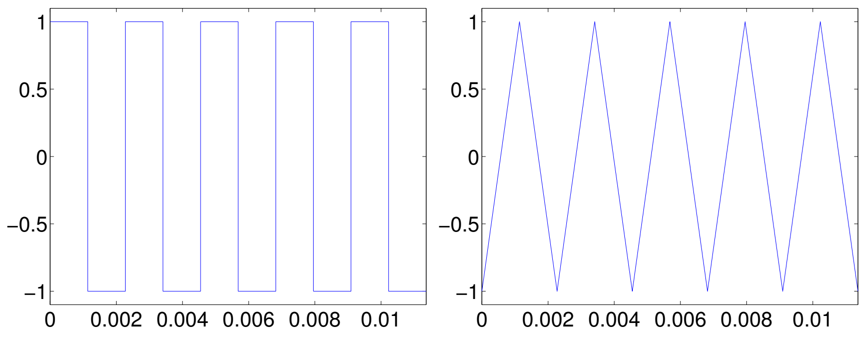
$$f(t) = \begin{cases} 1, & \text{if } 0 \le t < T/2; \\ -1, & \text{if } T/2 \le t < T. \end{cases} \tag{1.1}$$

In Figure 1.4(a) we have plotted the square wave when $T = 1/440$. This period is chosen so that it corresponds to the pure tone we already have listened to, and you can listen to this square wave here (in Exercise 5 you will learn how to generate this sound). We hear a sound with the same pitch as $\sin(2\pi 440t)$, but note that the square wave is less pleasant to listen to: There seems to be some sharp corners in the sound, translating into a rather shrieking, piercing sound. We will later explain this by the fact that the square wave can be viewed as a sum of many frequencies, and that all the different frequencies pollute the sound so that it is not pleasant to listen to.

Example 1.12. We define the *triangle wave* of period T as the function which repeats with period T, and increases linearly from -1 to 1 on the first half of each period, and decreases linearly from 1 to -1 on the second half of each period. This means that we can define it as the function

$$f(t) = \begin{cases} 4t/T - 1, & \text{if } 0 \le t < T/2; \\ 3 - 4t/T, & \text{if } T/2 \le t < T. \end{cases} \tag{1.2}$$

In Figure 1.4(b) we have plotted the triangle wave when $T = 1/440$. Again, this same choice of period gives us an audible sound, and you can listen to the triangle wave here (in Exercise 5 you will learn how to generate this sound). Again you will note that the triangle wave has the same pitch as $\sin(2\pi 440t)$,



(a) The first five periods of the square wave (b) The first five periods of the triangle wave

Figure 1.4: The square wave and the triangle wave, two functions with regular oscillations, but which are not simple, trigonometric functions.

and is less pleasant to listen to than this pure tone. However, one can argue that it is somewhat more pleasant to listen to than a square wave. This will also be explained in terms of pollution with other frequencies later.

In Section 2.1 we will begin to peek behind the curtains as to why these waves sound so different, even though we recognize them as having the exact same pitch.

1.3 Digital sound

In the previous section we considered some basic properties of sound, but it was all in terms of functions defined for all time instances in some interval. On computers and various kinds of media players the sound is usually *digital* which means that the sound is represented by a large number of function values, and not by a function defined for all times in some interval.

Definition 1.13 (Digital sound)**.** A digital sound is a sequence $\boldsymbol{x} = \{x_i\}_{i=0}^{N}$ that corresponds to measurements of the air pressure of a sound f, recorded at a fixed rate of f_s (the sampling frequency or sampling rate) measurements per second, i.e.,

$$x_i = f(i/f_s), \quad \text{for } i = 0,\, 1;\, \ldots,\, N.$$

The measurements are often referred to as samples. The time between successive measurements is called the sampling period and is usually denoted T_s. If the sound is in stereo there will be two arrays $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, one for each channel. Measuring the sound is also referred to as sampling the sound, or analog to digital (AD) conversion.

In most cases, a digital sound is sampled from an analog (continuous) audio signal. This is usually done with a technique called Pulse Code Modulation (PCM). The audio signal is sampled at regular intervals and the sampled values stored in a suitable number format. Both the sampling frequency, and the accuracy and number format used for storing the samples, may vary for different kinds of audio, and both influence the quality of the resulting sound. For simplicity the quality is often measured by the number of bits per second, i.e., the product of the sampling rate and the number of bits (binary digits) used to store each sample. This is also referred to as the *bit rate*. For the computer to be able to play a digital sound, samples must be stored in a file or in memory on a computer. To do this efficiently, digital sound formats are used. A couple of them are described in the examples below.

In Exercise 4 you will be asked to implement a Matlab-function which plays a pure sound with a given frequency on your computer. For this you will need to know that for a pure tone with frequency f, you can obtain its samples over a period of 3 seconds with sampling rate f_s from the code

```
t=0:(1/fs):3;
sd=sin(2*pi*f*t);
```

Here the code is in MATLAB. MATLAB code will be displayed in this way throughout these notes.

Example 1.14. In the classical CD-format the audio signal is sampled 44 100 times per second and the samples stored as 16-bit integers. This works well for music with a reasonably uniform dynamic range, but is problematic when the range varies. Suppose for example that a piece of music has a very loud passage. In this passage the samples will typically make use of almost the full range of integer values, from $-2^{15}-1$ to 2^{15}. When the music enters a more quiet passage the sample values will necessarily become much smaller and perhaps only vary in the range -1000 to 1000, say. Since $2^{10}=1024$ this means that in the quiet passage the music would only be represented with 10-bit samples. This problem can be avoided by using a floating-point format instead, but very few audio formats appear to do this.

The bit rate for CD-quality stereo sound is $44100 \times 2 \times 16$ bits/s = 1411.2 kb/s. This quality measure is particularly popular for lossy audio formats where the uncompressed audio usually is the same (CD-quality). However, it should be remembered that even two audio files in the same file format and with the same bit rate may be of very different quality because the encoding programs may be of different quality.

Example 1.15. For telephony it is common to sample the sound 8000 times per second and represent each sample value as a 13-bit integer. These integers are then converted to a kind of 8-bit floating-point format with a 4-bit significand. Telephony therefore generates a bit rate of 64 000 bits per second, i.e. 64 kb/s.

Newer formats with higher quality are available. Music is distributed in various formats on DVDs (DVD-video, DVD-audio, Super Audio CD) with sampling rates up to 192 000 and up to 24 bits per sample. These formats also support surround sound (up to seven channels in contrast to the two stereo channels on a CD). In the following we will assume all sound to be digital. Later we will return to how we reconstruct audible sound from digital sound.

1.4 Simple operations on digital sound

Simple operations and computations with digital sound can be done in any programming environment. Let us take a look at how this can be done. From Definition 1.13, digital sound is just an array of sample values $\boldsymbol{x} = (x_i)_{i=0}^{N-1}$, together with the sample rate f_s. Performing operations on the sound therefore amounts to doing the appropriate computations with the sample values and the sample rate. The most basic operation we can perform on a sound is simply playing it, and if we are working with sound we need a mechanism for doing this.

1.4.1 Playing a sound

You may already have listened to pure tones, square waves and triangle waves in the last section. The corresponding sound files were generated in a way we will describe shortly, placed in a directory available on the internet, and linked to from these notes. A program on your computer was able to play these files when you clicked on them. We will now describe how to use Matlab to play the same sounds. There we have the two functions

```
playblocking(playerobj)
playblocking(playerobj,[start stop])
```

These simply play an audio segment encapsulated by the object `playerobj` (we will shortly see how we can construct such an object from given audio samples and sampling rate). `playblocking` means that the method playing the sound will block until it has finished playing. We will have use for this functionality later on, since we may play sounds in successive order. With the first function the entire audio segment is played. With the second function the playback starts at sample `start`, and ends at sample `stop`. These functions are just software interfaces to the sound card in your computer. It basically sends the array of sound samples and sample rate to the sound card, which uses some method for reconstructing the sound to an analog sound signal. This analog signal is then sent to the loudspeakers and we hear the sound.

Fact 1.16. The basic command in a programming environment that handles sound takes as input an array of sound samples $\boldsymbol{x}$ and a sample rate s, and plays the corresponding sound through the computer's loudspeakers.

The mysterious `playerobj` object above can be obtained from the sound samples (represented by a vector `S`) and the sampling rate (`fs`) by the function:

```
playerobj=audioplayer(S,fs)
```

The sound samples can have different data types. We will always assume that they are of type `double`. MATLAB requires that they have values between -1 and 1 (i.e. these represent the range of numbers which can be played through the sound card of the computer). Also, `S` can actually be a matrix: Each column in the matrix represents a sound channel. Sounds we generate from a mathematical function on our own will typically have one only one channel, so that `S` has only one column. If `S` originates from a stereo sound file, it will have two columns.

You can create `S` on your own, and set the sampling rate to whatever value you like. However, we can also fill in the sound samples from a sound file. To do this from a file in the `wav`-format named `filename`, simply write

```
[S,fs]=wavread(filename)
```

The `wav`-format format was developed by Microsoft and IBM, and is one of the most common file formats for CD-quality audio. It uses a 32-bit integer to specify the file size at the beginning of the file which means that a WAV-file cannot be larger than 4 GB. In addition to filling in the sound samples in the vector `S`, this function also returns the sampling rate `fs` used in the file. The function

```
wavwrite(S,fs,filename)
```

can similarly be used to write the data stored in the vector `S` to the `wav`-file by the name `filename`. In the following we will both fill in the vector `S` on our own by using values from mathematical functions, as well as from a file. As an example of the first, we can listen to and write to a file the pure tone of frequency 440Hz considered above with the help of the following code:

```
antsec=3;
fs=40000;
t=linspace(0,antsec,fs*antsec);
S=sin(2*pi*440*t);
playerobj=audioplayer(S,fs);
playblocking(playerobj);
wavwrite(S,fs,'puretone440.wav');
```

The code creates a pure tone which lasts for three seconds (if you want the tone to last longer, you can change the value of the variable `antsec`). We also tell the computer that there are 40000 samples per second. This value is not coincidental, and we will return to this. In fact, the sound file for the pure tone embedded into this document was created in this way! In the same way we can listen to the square wave with the help of the following code:

```
antsec=3;
fs=44100;
```

```
samplesperperiod=round(fs/440);
oneperiod=[ones(1,round(samplesperperiod/2)) ...
           -ones(1,round(samplesperperiod/2))];
allsamples=zeros(1,antsec*440*length(oneperiod));
for k=1:(antsec*440)
  allsamples(((k-1)*length(oneperiod)+1):k*length(oneperiod))=oneperiod;
end
playerobj=audioplayer(allsamples,fs);
playblocking(playerobj);
```

The code creates 440 copies of the square wave per second by first computing the number of samples needed for one period when it is known that we should have a total of 40000 samples per second, and then constructing the samples needed for one period. In the same fashion we can listen to the triangle wave simply by replacing the code for generating the samples for one period with the following:

```
oneperiod=[linspace(-1,1,round(samplesperperiod/2)) ...
           linspace(1,-1,round(samplesperperiod/2))];
```

Instead of using the formula for the triangle wave, directly, we have used the function `linspace`.

As an example of how to fill in the sound samples from a file, the code

```
[S fs] = wavread('castanets.wav');
```

reads the file castanets.wav, and stores the sound samples in the matrix `S`. In this case there are two sound channels, so there are two columns in `S`. To work with sound from only one channel, we extract the second channel as follows:

```
x=S(:,2);
```

`wavread` returns sound samples with floating point precision. If we have made any changes to the sound samples, we need to secure that they are between -1 and 1 before we play them. If the sound samples are stored in `x`, this can be achieved as follows:

```
x = x / max(abs(x));
```

`x` can now be played just as the signals we constructed from mathematical formulas above.

It may be that some other environment than Matlab gives you the `play` functionality on your computer. Even if no environment on your computer supports such `play`-functionality at all, you may still be able to play the result of your computations if there is support for saving the sound in some standard format like mp3. The resulting file can then be played by the standard audio player on your computer.

Example 1.17 (Changing the sample rate)**.** We can easily play back a sound with a different sample rate than the standard one. If we in the code above instead wrote `fs=80000`, the sound card will assume that the time distance between neighbouring samples is half the time distance in the original. The result is that the sound takes half as long, and the frequency of all tones is doubled. For voices the result is a characteristic Donald Duck-like sound.

Conversely, the sound can be played with half the sample rate by setting `fs=20000`. Then the length of the sound is doubled and all frequencies are halved. This results in low pitch, roaring voices.

Fact 1.18. A digital sound can be played back with a double or half sample rate by replacing

```
playerobj=audioplayer(S,fs);
```

with

```
playerobj=audioplayer(S,2*fs);
```

and

```
playerobj=audioplayer(S,fs/2);
```

respectively.

The sample file `castanets.wav` played at double sampling rate sounds like this, while it sounds like this when it is played with half the sampling rate.

Example 1.19 (Playing the sound backwards)**.** At times a popular game has been to play music backwards to try and find secret messages. In the old days of analog music on vinyl this was not so easy, but with digital sound it is quite simple; we just need to reverse the samples. To do this we just loop through the array and put the last samples first.

Fact 1.20. Let $\boldsymbol{x} = (x_i)_{i=0}^{N-1}$ be the samples of a digital sound. Then the samples $\boldsymbol{y} = (y_i)_{i=0}^{N-1}$ of the reverse sound are given by

$$y_i = x_{N-i-1}, \text{ for } i = 0, 1, \ldots N-1.$$

When we reverse the sound samples with Matlab, we have to reverse the elements in both sound channels. This can be performed as follows

```
sz=size(S,1);
newS=[S(sz:(-1):1,1) S(sz:(-1):1,2)];
```

Performing this on our sample file you generate a sound which sounds like this.

Example 1.21 (Adding noise)**.** To remove noise from recorded sound can be very challenging, but adding noise is simple. There are many kinds of noise, but one kind is easily obtained by adding random numbers to the samples of a sound.

Fact 1.22. Let $\boldsymbol{x}$ be the samples of a digital sound of length N. A new sound $\boldsymbol{y}$ with noise added can be obtained by adding a random number to each sample,

```
y=x+c*(2*rand(1,N)-1);
```

where `rand` is a MATLAB function that returns random numbers in the interval $[0,1]$, and c is a constant (usually smaller than 1) that dampens the noise. The effect of writing `(2*rand(1,N)-1)` is that random numbers between -1 and 1 are returned instead of random numbers between 0 and 1.

Adding noise in this way will produce a general hissing noise similar to the noise you hear on the radio when the reception is bad. As before you should add noise to both channels. Note alse that the sound samples may be outside $[-1,1]$ after adding noise, so that you should scale the samples before writing them to file. The factor c is important, if it is too large, the noise will simply drown the signal $\boldsymbol{y}$: `castanets.wav` with noise added with $c = 0.4$ sounds like this, while with $c = 0.1$ it sounds like this.

1.4.2 Filtering operations

Later on we will focus on particular operations on sound, where the output is constructed by combining several input elements in a particular way. We say that we *filter* the sound, and we call such operations *filtering operations*, or simply *filters*. Filters are important since they can change the frequency content in a signal in many ways. We will defer the precise definition of filters to Section 3.3, where we also will give the filters listed below a closer mathematical analysis.

Example 1.23 (Adding echo)**.** An echo is a copy of the sound that is delayed and softer than the original sound. We observe that the sample that comes m seconds before sample i has index $i - ms$ where s is the sample rate. This also makes sense even if m is not an integer so we can use this to produce delays that are less than one second. The one complication with this is that the number ms may not be an integer. We can get round this by rounding ms to the nearest integer which corresponds to adjusting the echo slightly.

Fact 1.24. Let $(\boldsymbol{x}, s)$ be a digital sound. Then the sound $\boldsymbol{y}$ with samples given by

```
y=x((d+1):N)-c*x(1:(N-d));
```

will include an echo of the original sound. Here `d=round(ms)` is the integer closest to ms, and c is a constant which is usually smaller than 1.

This is an example of a filtering operation where each output element is constructed from two input elements. As in the case of noise it is important to dampen the part that is added to the original sound, otherwise the echo will be too loud. Note also that the formula that creates the echo does not work at the beginning of the signal, and that the echo is unaudible if d is too small. You can listen to the sample file with echo added with $d = 10000$ and $c = 0.5$ here.

Example 1.25 (Reducing the treble)**.** The treble in a sound is generated by the fast oscillations (high frequencies) in the signal. If we want to reduce the treble we have to adjust the sample values in a way that reduces those fast oscillations. A general way of reducing variations in a sequence of numbers is to replace one number by the average of itself and its neighbours, and this is easily done with a digital sound signal. If we let the new sound signal be $\boldsymbol{y} = (y_i)_{i=0}^{N-1}$ we can compute it as

```
y(1)=x(1);
for t=2:(N-1)
  y(t)=(x(t-1)+x(t)+x(t+1))/3;
end
y(N)=x(N);
```

This is another example of a filtering operation, but this time three input elements are needed in order to produce an output element. Note that the vector $\{1/3, 1/3, 1/3\}$ uniquely describe how the input elements should be combined to produce the otuput. The elements in this vector are also referred to as the *filter coefficients*. Since this filter is based on forming averages it is also called a *moving average filter*.

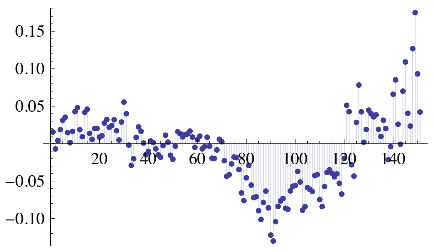
It is reasonable to let the middle sample x_i count more than the neighbours in the average, so an alternative is to compute the average by instead writing

```
  y(t)=(x(t-1)+2*x(t)+x(t+1))/4;
```

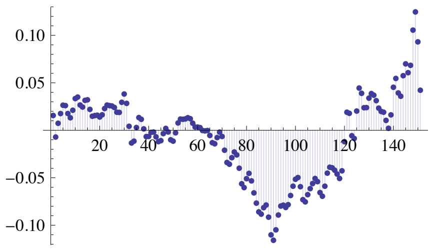
The coefficients $1, 2, 1$ here have been taken from row 2 in Pascal's triangle. It will turn out that this is a good choice of coefficients. We have not developed the tools needed to analyse the quality of filters yet, so this will be discussed later. We can also take averages of more numbers, where it will also turn out that row k of Pascals triangle also is a very good choice. The values in Pascals triangle can be computed as the coefficients of x in the expression $(1 + x)^k$, which also equal the binomial coefficients $\binom{k}{r}$ for $0 \leq r \leq k$. As an example, if we pick coefficients from row 4 of Pascals triangle instead, we would write

```
y(1)=x(1); y(2)=x(2);
for t=3:(N-2)
  y(t)=(x(t-2)+4*x(t-1)+6*x(t)+4*x(t+1)+x(t+2))/16;
end
y(N-1)=x(N-1); y(N)=x(N);
```

It will turn out that picking coefficients from a row in Pascal's triangle works better the longer the filter is:



(a) The original sound signal



(b) The result of applying the filter from row 4 of Pascal's triangle

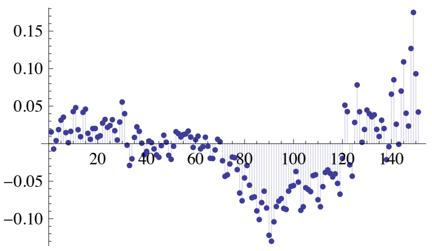
Figure 1.5: Reducing the treble.

Observation 1.26. Let $\boldsymbol{x}$ be the samples of a digital sound, and let $\{c_i\}_{i=1}^{2k+1}$ be the numbers in row $2k$ of Pascal's triangle. Then the sound with samples $\boldsymbol{y}$ given by

```
y=zeros(length(x));
y(1:k)=x(1:k);
for t=(k+1):(N-k)
  for j=1:(2*k+1)
    y(t)=y(t)+c(j)*x(t+k+1-j))/2^k;
  end
end
y((N-k+1):N)=x((N-k+1):N);
```

has reduced treble compared with the sound given by the samples $\boldsymbol{x}$.

An example of the result of averaging is shown in Figure 1.5. (a) shows a real sound sampled at CD-quality (44 100 samples per second). (b) shows the result of applying the averaging process by using row 4 of Pascals triangle. We see that the oscillations have been reduced, and if we play the sound it has considerably less treble. In Exercise 9 you will be asked to implement reducing the treble in the file `castanets.wav`. If you do this you should hear that the sound gets softer when you increase k: For $k = 32$ the sound will be like this, for $k = 256$


(a) The original sound signal

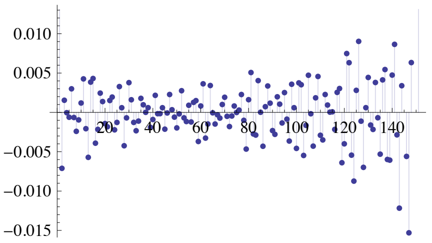

(b) The result of applying the filter deduced from row 4 in Pascals triangle

Figure 1.6: Reducing the bass.

it will be like this.

Example 1.27 (Reducing the bass)**.** Another common option in an audio system is reducing the bass. This corresponds to reducing the low frequencies in the sound, or equivalently, the slow variations in the sample values. It turns out that this can be accomplished by simply changing the sign of the coefficients used for reducing the treble. We can for instance change the filter described for the fourth row in Pascals triangle to

```
y(t)=(x(t-2)-4*x(t-1)+6*x(t)-4*x(t+1)+x(t+2))/16;
```

An example is shown in Figure 1.6. The original signal is shown in (a) and the result in (b). We observe that the samples in (b) oscillate much more than the samples in (a). If we play the sound in (b), it is quite obvious that the bass has disappeared almost completely.

Observation 1.28. Let $\boldsymbol{x}$ be the samples of a digital sound, and let $\{c_i\}_{i=1}^{2k+1}$ be the numbers in row $2k$ of Pascal's triangle. Then the sound with samples $\boldsymbol{y}$ given by

```
y=zeros(length(x));
y(1:k)=x(1:k);
for t=(k+1):(N-k)
```

```
  for j=1:(2*k+1)
    y(t)=x(t)+(-1)^(k+1-j)*c(j)*x(t+j-k-1))/2^k;
  end
end
y((N-k+1):N)=x((N-k+1):N);
```

has reduced bass compared to the sound given by the samples $\boldsymbol{y}$.

In Exercise 9 you will be asked to implement reducing the bass in the file `castanets.wav`. The new sound will be difficult to hear for large k, and we will explain why later. For $k = 1$ the sound will be like this, for $k = 2$ it will be like this. Even if the sound is quite low, you can hear that more of the bass has disappeared for $k = 2$.

There are also other operations we would like to perform for digital sound. For instance, it would be nice to adjust a specific set of frequencies only, so that we end up with a sound where unpleasant components of the sound have been removed. Later on we will establish mathematics which will enable us to contruct filters which have the properties that they work in desirable ways on any frequencies. It will also turn out that the filters listed above can be given a frequency interpretation: When the input sound has a given frequncy, the output sound has the same frequency.

Exercises for Section 1.4

Ex. 1 — Compute the loudness of the Krakatoa explosion on the decibel scale, assuming that the variation in air pressure peaked at 100 000 Pa.

Ex. 2 — Define the following sound signal

$$f(t) = \begin{cases} 0 & 0 \leq t \leq 0.2 \\ \frac{t-0.2}{0.4}\sin(2\pi 440t) & 0.2 \leq t \leq 0.6 \\ \sin(2\pi 440t) & 0.6 \leq t \leq 1 \end{cases}$$

This corresponds to the sound plotted in Figure 1.1(a), where the sound is unaudible in the beginning, and increases linearly in loudness over time with a given frequency until maximum loudness is avchieved. Write a Matlab program which generates this sound, and listen to it.

Ex. 3 — Find two constant a and b so that the function $f(t) = a\sin(2\pi 440t) + b\sin(2\pi 4400t)$ resembles the plot from Figure 1.1(b) as closely as possible. Generate the samples of this sound, and listen to it with Matlab.

Ex. 4 — Let us write some code so that we can experiment with different pure sounds

a. Write a function

```
function playpuresound(f)
```

which generates the samples over a period of 3 seconds for a pure tone with frequency f, with sampling frequency $f_s = 2.5f$ (we will explain this value later).

b. Use the function `playpuresound` to listen to pure sounds of frequency 440Hz and 1500Hz, and verify that they are the same as the sounds you already have listened to in this section.

c. How high frequencies are you able to hear with the function `playpuresound`? How low frequencies are you able to hear?

Ex. 5 — Write functions

```
function playsquare(T)
function playtriangle(T)
```

which plays the square wave of Example 1.11 and the triangle wave of Example 1.12, respectively, where T is given by the parameter. In your code, let the samples of the waves be taken at a frequency of 40000 samples per second. Verify that you generate the same sounds as you played in these examples when you set $T = \frac{1}{440}$.

Ex. 6 — In this exercise we will experiment as in the first examples of this section.

a. Write a function

```
function playdifferentfs()
```

which plays the sound samples of `castanets.wav` with the same sample rate as the original file, then with twice the sample rate, and then half the sample rate. You should start with reading the file into a matrix (as explained in this section). Are the sounds the same as those you heard in Example 1.17?

b. Write a function

```
function playreverse()
```

which plays the sound samples of `castanets.wav` backwards. Is the sound the same as the one you heard in Example 1.19?

c. Write the new sound samples from b. to a new `wav`-file, as described above, and listen to it with your favourite mediaplayer.

Ex. 7 — In this exercise, we will experiment with adding noise to a signal.

a. Write a function

```
function playnoise(c)
```

which plays the sound samples of `castanets.wav` with noise added for the damping constant c as described above. Your code should add noise to both channels of the sound, and scale the sound samples so that they are between -1 and 1.

b. With your program, generate the two sounds played in Example 1.21, and verify that they are the same as those you heard.

c. Listen to the sound samples with noise added for different values of c. For which range of c is the noise audible?

Ex. 8 — In this exercise, we will experiment with adding echo to a signal.

a. Write a function

```
function playwithecho(c,d)
```

which plays the sound samples of `castanets.wav` with echo added for damping constant c and delay d as described in Example 1.23.

b. Generate the sound from Example 1.23, and verify that it is the same as the one you heard there.

c. Listen to the sound samples for different values of d and c. For which range of d is the echo distinguisible from the sound itself? How low can you choose c in order to still hear the echo?

Ex. 9 — In this exercise, we will experiment with increasing and reducing the treble and bass in a signal as in examples 1.25 and 1.27.

a. Write functions

```
function reducetreble(k)
function reducebass(k)
```

which reduces bass and treble in the ways described above for the sound from the file `castanets.wav`, and plays the result, when row number $2k$ in Pascal' triangle is used to construct the filters. Look into the Matlab function `conv` to help you to find the values in Pascal's triangle.

b. Generate the sounds you heard in examples 1.25 and 1.27, and verify that they are the same.

c. In your code, it will not be necessary to scale the values after reducing the treble, i.e. the values are already between -1 and 1. Explain why this is the case.

d. How high must k be in order for you to hear difference from the actual sound? How high can you choose k and still recognize the sound at all?

1.5 Compression of sound and the MP3 standard

Digital audio first became commonly available when the CD was introduced in the early 1980s. As the storage capacity and processing speeds of computers increased, it became possible to transfer audio files to computers and both play and manipulate the data, in ways such as in the previous section. However, audio was represented by a large amount of data and an obvious challenge was how to reduce the storage requirements. Lossless coding techniques like Huffman and Lempel-Ziv coding were known and with these kinds of techniques the file size could be reduced to about half of that required by the CD format. However, by allowing the data to be altered a little bit it turned out that it was possible to reduce the file size down to about ten percent of the CD format, without much loss in quality. The MP3 audio format takes advantage of this.

MP3, or more precisely *MPEG-1 Audio Layer 3*, is part of an audio-visual standard called MPEG. MPEG has evolved over the years, from MPEG-1 to MPEG-2, and then to MPEG-4. The data on a DVD disc can be stored with either MPEG-1 or MPEG-2, while the data on a bluray-disc can be stored with either MPEG-2 or MPEG-4. MP3 was developed by Philips, CCETT (Centre commun d'études de télévision et télécommunications), IRT (Institut für Rundfunktechnik) and Fraunhofer Society, and became an international standard in 1991. Virtually all audio software and music players support this format. MP3 is just a sound format and does not specify the details of how the encoding should be done. As a consequence there are many different MP3 encoders available, of varying quality. In particular, an encoder which works well for higher bit rates (high quality sound) may not work so well for lower bit rates.

With MP3, the sound samples are transformed using methods we will go through in the next section. A frequency analysis of the sound is the basis for this transformation. Based on this frequency analysis, the sound is split into frequency bands, each band corresponding to a particular frequency range. With MP3, 32 frequency bands are used. Based on the frequency analysis, the encoder uses what is called a *psycho-acoustic model* to compute the significance of each band for the human perception of the sound. When we hear a sound, there is a mechanical stimulation of the ear drum, and the amount of stimulus is directly related to the size of the sample values of the digital sound. The movement of the ear drum is then converted to electric impulses that travel to the brain where they are perceived as sound. The perception process uses a transformation of the sound so that a steady oscillation in air pressure is perceived as a sound with a fixed frequency. In this process certain kinds of perturbations of the sound are hardly noticed by the brain, and this is exploited in lossy audio compression.

More precisely, when the psycho-acoustic model is applied to the frequency content resulting from our frequency analysis, *scale factors* and *masking thresh-*

olds are assigned for each band. The computed masking thresholds have to do with a phenomenon called *masking effects*. A simple example of this is that a loud sound will make a simultaneous low sound inaudible. For compression this means that if certain frequencies of a signal are very prominent, most of the other frequencies can be removed, even when they are quite large. If the sounds are below the masking threshold, it is simply ommited by the encoder, since the model says that the sound should be inaudible.

Masking effect is just one example of what is called a psycho-acoustic effect. Another obvious such effect regards computing the scale factors: the human auditory system can only perceive frequencies in the range 20 Hz – 20 000 Hz. An obvious way to do compression is therefore to remove frequencies outside this range, although there are indications that these frequencies may influence the listening experience inaudibly. The computed scaling factors tell the encoder about the precision to be used for each frequency band: If the model decides that one band is very important for our perception of the sound, it assigns a big scale factor to it, so that more effort is put into encoding it by the encoder (i.e. it uses more bits to encode this band).

Using appropriate scale factors and masking thresholds provide compression, since bits used to encode the sound are spent on parts important for our perception. Developing a useful psycho-acoustic model requires detailed knowledge of human perception of sound. Different MP3 encoders use different such models, so that may produce very different results, worse or better.

The information remaining after frequency analysis and using a psycho-acoustic model is coded efficiently with (a variant of) Huffman coding. MP3 supports bit rates from 32 to 320 kb/s and the sampling rates 32, 44.1, and 48 kHz. The format also supports variable bit rates (the bit rate varies in different parts of the file). An MP3 encoder also stores metadata about the sound, such as the title of the audio piece, album and artist name and other relevant data.

MP3 too has evolved in the same way as MPEG, from MP1 to MP2, and to MP3, each one more sophisticated than the other, providing better compression. MP3 is not the latest development of audio coding in the MPEG family: AAC (Advanced Audio Coding) is presented as the successor of MP3 by its principal developer, Fraunhofer Society, and can achieve better quality than MP3 at the same bit rate, particularly for bit rates below 192 kb/s. AAC became well known in April 2003 when Apple introduced this format (at 128 kb/s) as the standard format for their iTunes Music Store and iPod music players. AAC is also supported by many other music players, including the most popular mobile phones.

The technologies behind AAC and MP3 are very similar. AAC supports more sample rates (from 8 kHz to 96 kHz) and up to 48 channels. AAC uses the same transformation as MP3, but AAC processes 1 024 samples at a time. AAC also uses much more sophisticated processing of frequencies above 16 kHz and has a number of other enhancements over MP3. AAC, as MP3, uses Huffman coding for efficient coding of the transformed values. Tests seem quite conclusive that AAC is better than MP3 for low bit rates (typically below 192 kb/s), but for higher rates it is not so easy to differentiate between the two formats. As for MP3 (and the other formats mentioned here), the quality of an AAC file depends crucially on the quality of the encoding program.

There are a number of variants of AAC, in particular AAC Low Delay (AAC-LD). This format was designed for use in two-way communication over a network, for example the internet. For this kind of application, the encoding (and decoding) must be fast to avoid delays (a delay of at most 20 ms can be tolerated).

1.6 Summary

We discussed the basic question of what is sound is, and concluded that sound could be modeled as a sum of frequency components. We discussed meaningful operations of sound, such as adjust the bass and treble, adding echo, or adding noise. We also gave an introduction to the MP3 standard for compression of sound.