Chapter 1

Sound and Fourier series

A major part of the information we receive and perceive every day is in the form of audio. Most 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. *Sound as mathematical objects.*

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 sound* .

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 last part

of this chapter we establish the framework for this study, and analyze this on some examples for sound.

1.1 Characteristics of sound: Loudness and frequency

An example of a simple sound is shown in the left plot in Figure 1.1 where the oscillations in air pressure are plotted against 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 summarizes some basic facts about air pressure.

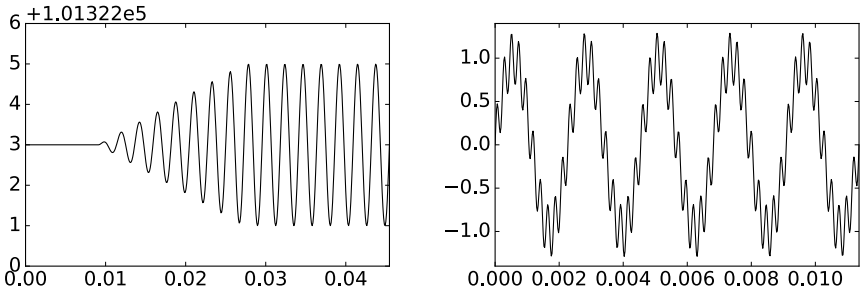

Figure 1.1: Two examples of audio signals. In terms of air pressure (left), and in terms of the difference from the ambient air pressure (right).

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 the left plot in Figure 1.1: 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 the left part of Figure 1.1. In the right plot in Figure 1.1 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 summarizes 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,

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

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 2.1 and 2.2). 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 the first half second there are so many oscillations that it is impossible to see the details, but if we zoom in on the first 0.002 seconds we can see that there is a continuous function behind all the ink. In reality the air pressure varies more than this, even over this short time period, but the measuring equipment may not be able to pick up those variations, and it is also doubtful whether we would be able to perceive such rapid variations.

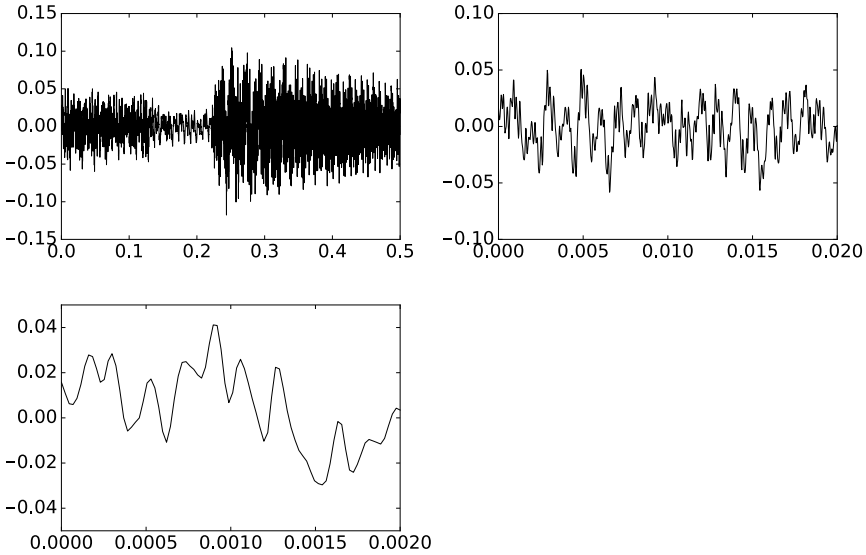

Figure 1.2: Variations in air pressure during parts of a song. The first 0.5 seconds, the first 0.02 seconds, and the first 0.002 seconds.

1.1.1 The frequency 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. *Human hearing.*
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. *Periodic functions.*
A real function f is said to be periodic with period T if

$$f(t+T) = f(t)$$

for all real numbers t.

Note that all the values of a periodic function f with period T are known if $f(t)$ is known for all t in the interval $[0, T)$. 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 $T = 1/\nu$. 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.

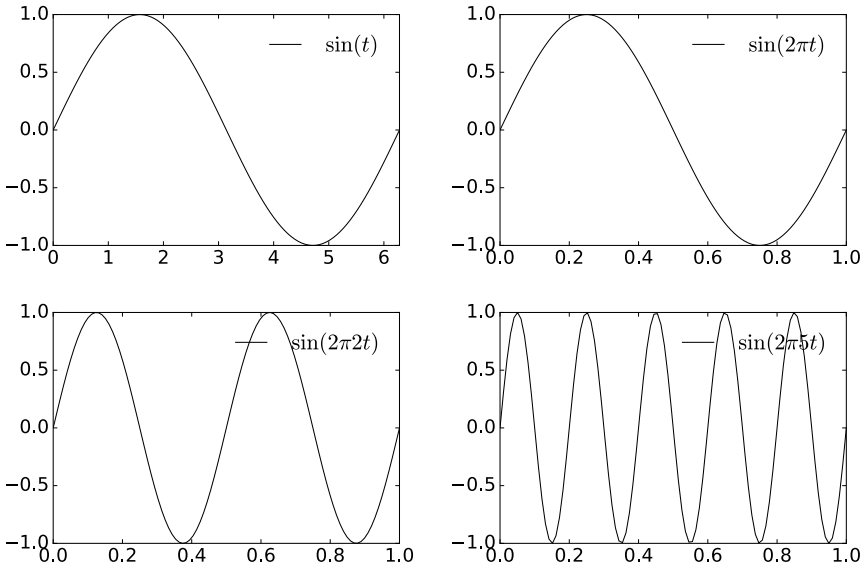

Figure 1.3: Versions of sin with different frequencies.

Figure 1.3 illustrates observation 1.6. The function in the upper left 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 the lower right 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 modeled by functions, it is reasonable to say that a sound with frequency ν is a trigonometric function with frequency ν.

Definition 1.7. *Pure tones.*

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. In Section 2.1 we will explain how we generated these sounds so that they could be played on a computer.

Any sound may be considered to be a function. In the next section we will explain why 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 also 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, most information is found 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.

Idea 1.9. *Audio compression.*

Suppose an audio signal f is given. To compress f, perform the following steps:

- Rewrite the signal f in a new format where frequency information becomes accessible.
- Remove those frequencies that only contribute marginally to human perception of the sound.
- 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 that Observation 1.8 can be used as a basis for many operations on sound. It also makes it possible to explain what it means that we only perceive sounds with a frequency in the range 20–20000 Hz: This simply says that there is a significant contribution from one of those frequencies in the decomposition.

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 frequency, as expected. There are, however, many other ways in which a function can oscillate regularly. The function in The right plot in Figure 1.1 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 1.10. *The square wave.*

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

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

In the left plot in Figure 1.4 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 2.4 you will learn how to generate this sound. We hear a sound with the same frequency 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.

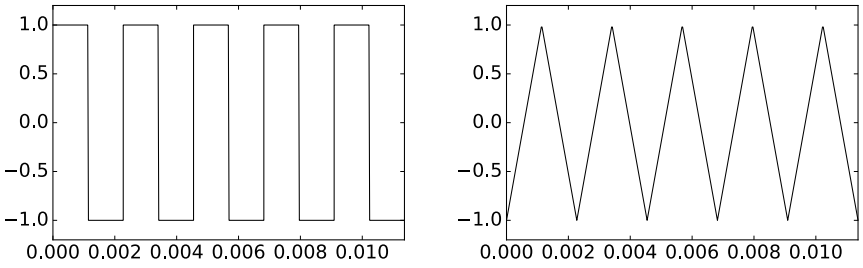

Figure 1.4: The first five periods of the square wave and the triangle wave, two functions with regular oscillations, but which are not simple, trigonometric functions.

Example 1.11. *The triangle wave.*

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

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

In the right plot in Figure 1.4 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. Again you will note that the triangle wave has the same frequency as $\sin(2\pi 440t)$, 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 1.2 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 frequency.

Exercise 1.1: The Krakatoa explosion

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

Exercise 1.2: Sum of two pure tones

Consider a sum of two pure tones, $f(t) = A_1 \sin(2\pi\nu_1 t) + A_2 \sin(2\pi\nu_2 t)$. For which values of A_1, A_2, ν_1, ν_2 is f periodic? What is the period of f when it is periodic?

1.2 Fourier series: Basic concepts

In Section 1.1.1 we identified audio signals with functions and discussed informally the idea of decomposing a sound into basis sounds (pure sounds) to make its frequency content available. In this chapter we will make this kind of decomposition more precise by discussing how a given function can be expressed in terms of the basic trigonometric functions. This is similar to Taylor series where functions are approximated by combinations of polynomials. But it is also different from Taylor series because we use trigonometric series rather than power series, and the approximations are computed in a very different way. The theory of approximation of functions with trigonometric functions is generally referred to as *Fourier analysis* . This is a central tool in practical fields like image- and signal processing, but it is also an important field of research within pure mathematics.

In the start of this chapter we had no constraints on the function f. Although Fourier analysis can be performed for very general functions, it turns out that it takes its simplest form when we assume that the function is periodic. Periodic

functions are fully known when we know their values on a period $[0, T]$. In this case we will see that we can carry out the Fourier analysis in finite dimensional vector spaces of functions. This makes linear algebra a very useful tool in Fourier analysis: Many of the tools from your linear algebra course will be useful, in a situation that at first may seem far from matrices and vectors.

The basic idea of Fourier series is to approximate a given function by a combination of simple cos and sin functions. This means that we have to address at least three questions:

- How general do we allow the given function to be?
- What exactly are the combinations of cos and sin that we use for the approximations?
- How do we determine the approximation?

Each of these questions will be answered in this section. Since we restrict to periodic functions, we will without much loss of generality assume that the functions are defined on $[0, T]$, where T is some positive number. Mostly we will also assume that f is continuous, but the theory can also be extended to functions which are only Riemann-integrable, and more precisely, to square integrable functions.

Definition 1.12. *Continuous and square-integrable functions.*

The set of continuous, real functions defined on an interval $[0, T]$ is denoted $C[0, T]$.

A real function f defined on $[0, T]$ is said to be square integrable if f^2 is Riemann-integrable, i.e., if the Riemann integral of f^2 on $[0, T]$ exists,

$$\int_0^T f(t)^2 \, dt < \infty.$$

The set of all square integrable functions on $[0, T]$ is denoted $L^2[0, T]$.

The sets of continuous and square-integrable functions can be equipped with an inner-product, a generalization of the so-called dot-product for vectors.

Theorem 1.13. *Inner product spaces.*

Both $L^2[0, T]$ and $C[0, T]$ are vector spaces. Moreover, if the two functions f and g lie in $L^2[0, T]$ (or in $C[0, T]$), then the product fg is Riemann-integrable (or in $C[0, T]$). Moreover, both spaces are inner product spaces [1] with inner product [2] defined by

$$\langle f, g \rangle = \frac{1}{T} \int_0^T f(t)g(t) \, dt, \tag{1.3}$$

and associated norm

[1] See Section 6.1 in [20] for a review of inner products and orthogonality.

[2] See Section 6.7 in [20] for a review of function spaces as inner product spaces.

$$\|f\| = \sqrt{\frac{1}{T}\int_0^T f(t)^2 dt}. \tag{1.4}$$

The mysterious factor $1/T$ is included so that the constant function $f(t) = 1$ has norm 1, i.e., its role is as a normalizing factor.

Definition 1.12 and Theorem 1.13 answer the first question above, namely how general we allow our functions to be. Theorem 1.13 also gives an indication of how we are going to determine approximations: we are going to use inner products. We recall from linear algebra that the projection of a function f onto a subspace W with respect to an inner product $\langle \cdot, \cdot \rangle$ is the function $g \in W$ which minimizes $\|f - g\|$, also called *the error* in the approximation [3]. This projection is therefore also called a best approximation of f from W and is characterized by the fact that the function $f - g$, also called the *error function*, should be orthogonal to the subspace W, i.e. we should have

$$\langle f - g, h \rangle = 0, \quad \text{for all } h \in W.$$

More precisely, if $\boldsymbol{\phi} = \{\phi_i\}_{i=1}^m$ is an orthogonal basis for W, then the best approximation g is given by

$$g = \sum_{i=1}^{m} \frac{\langle f, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle} \phi_i. \tag{1.5}$$

The error $\|f - g\|$ is often referred to as the *least square error*.

We have now answered the second of our primary questions. What is left is a description of the subspace W of trigonometric functions. This space is spanned by the pure tones we discussed in Section 1.1.1.

Definition 1.14. *Fourier series.*

Let $V_{N,T}$ be the subspace of $C[0, T]$ spanned by the set of functions given by

$$\begin{aligned} \mathcal{D}_{N,T} = \{&1, \cos(2\pi t/T), \cos(2\pi 2t/T), \cdots, \cos(2\pi N t/T), \\ &\sin(2\pi t/T), \sin(2\pi 2t/T), \cdots, \sin(2\pi N t/T)\}. \end{aligned} \tag{1.6}$$

The space $V_{N,T}$ is called the *N'th order Fourier space.*

The Nth-order Fourier series approximation of f, denoted f_N, is defined as the best approximation of f from $V_{N,T}$ with respect to the inner product defined by (1.3).

The space $V_{N,T}$ can be thought of as the space spanned by the pure tones of frequencies $1/T$, $2/T$, ..., N/T, and the Fourier series can be thought of as linear combination of all these pure tones. From our discussion in Section 1.1.1, we should expect that if N is sufficiently large, $V_{N,T}$ can be used to approximate most sounds in real life. The approximation f_N of a sound f from a space $V_{N,T}$

[3] See Section 6.3 in [20] for a review of projections and least squares approximations.

can also serve as a compressed version if many of the coefficients can be set to 0 without the error becoming too big.

Note that all the functions in the set $\mathcal{D}_{N,T}$ are periodic with period T, but most have an even shorter period. More precisely, $\cos(2\pi nt/T)$ has period T/n, and frequency n/T. In general, the term *fundamental frequency* is used to denote the lowest frequency of a given periodic function.

Definition 1.14 characterizes the Fourier series. The next lemma gives precise expressions for the coefficients.

Theorem 1.15. *Fourier coefficients.*

The set $\mathcal{D}_{N,T}$ is an orthogonal basis for $V_{N,T}$. In particular, the dimension of $V_{N,T}$ is $2N+1$, and if f is a function in $L^2[0,T]$, we denote by $a_0,\dots,a_N$ and $b_1,\dots,b_N$ the coordinates of f_N in the basis $\mathcal{D}_{N,T}$, i.e.

$$f_N(t) = a_0 + \sum_{n=1}^{N} \left(a_n \cos(2\pi nt/T) + b_n \sin(2\pi nt/T)\right). \tag{1.7}$$

The $a_0,\dots,a_N$ and $b_1,\dots,b_N$ are called the (real) Fourier coefficients of f, and they are given by

$$a_0 = \langle f, 1\rangle = \frac{1}{T}\int_0^T f(t)\,dt, \tag{1.8}$$

$$a_n = 2\big\langle f, \cos(2\pi nt/T)\big\rangle = \frac{2}{T}\int_0^T f(t)\cos(2\pi nt/T)\,dt \quad \text{for } n \geq 1, \tag{1.9}$$

$$b_n = 2\langle f, \sin(2\pi nt/T)\rangle = \frac{2}{T}\int_0^T f(t)\sin(2\pi nt/T)\,dt \quad \text{for } n \geq 1. \tag{1.10}$$

Proof. To prove orthogonality, assume first that $m \neq n$. We compute the inner product

$$\begin{aligned}
&\langle \cos(2\pi mt/T), \cos(2\pi nt/T)\rangle \\
&= \frac{1}{T}\int_0^T \cos(2\pi mt/T)\cos(2\pi nt/T)dt \\
&= \frac{1}{2T}\int_0^T \left(\cos(2\pi mt/T + 2\pi nt/T) + \cos(2\pi mt/T - 2\pi nt/T)\right) \\
&= \frac{1}{2T}\left[\frac{T}{2\pi(m+n)}\sin(2\pi(m+n)t/T) + \frac{T}{2\pi(m-n)}\sin(2\pi(m-n)t/T)\right]_0^T \\
&= 0.
\end{aligned}$$

Here we have added the two identities $\cos(x \pm y) = \cos x \cos y \mp \sin x \sin y$ together to obtain an expression for $\cos(2\pi mt/T)\cos(2\pi nt/T)dt$ in terms of $\cos(2\pi mt/T + 2\pi nt/T)$ and $\cos(2\pi mt/T - 2\pi nt/T)$. By testing all other combinations of sin

and cos also, we obtain the orthogonality of all functions in $\mathcal{D}_{N,T}$ in the same way.

We find the expressions for the Fourier coefficients from the general formula (1.5). We first need to compute the following inner products of the basis functions,

$$\begin{aligned}\langle \cos(2\pi mt/T), \cos(2\pi mt/T)\rangle &= \frac{1}{2}\\ \langle \sin(2\pi mt/T), \sin(2\pi mt/T)\rangle &= \frac{1}{2}\\ \langle 1, 1\rangle &= 1,\end{aligned}$$

which are easily derived in the same way as above. The orthogonal decomposition theorem (1.5) now gives

$$\begin{aligned}f_N(t) &= \frac{\langle f, 1\rangle}{\langle 1, 1\rangle}1 + \sum_{n=1}^{N} \frac{\langle f, \cos(2\pi nt/T)\rangle}{\langle \cos(2\pi nt/T), \cos(2\pi nt/T)\rangle} \cos(2\pi nt/T)\\ &+ \sum_{n=1}^{N} \frac{\langle f, \sin(2\pi nt/T)\rangle}{\langle \sin(2\pi nt/T), \sin(2\pi nt/T)\rangle} \sin(2\pi nt/T)\\ &= \frac{\frac{1}{T}\int_0^T f(t)dt}{1} + \sum_{n=1}^{N} \frac{\frac{1}{T}\int_0^T f(t)\cos(2\pi nt/T)dt}{\frac{1}{2}} \cos(2\pi nt/T)\\ &+ \sum_{n=1}^{N} \frac{\frac{1}{T}\int_0^T f(t)\sin(2\pi nt/T)dt}{\frac{1}{2}} \sin(2\pi nt/T)\\ &= \frac{1}{T}\int_0^T f(t)dt + \sum_{n=1}^{N} \left(\frac{2}{T}\int_0^T f(t)\cos(2\pi nt/T)dt\right) \cos(2\pi nt/T)\\ &+ \sum_{n=1}^{N} \left(\frac{2}{T}\int_0^T f(t)\sin(2\pi nt/T)dt\right) \sin(2\pi nt/T).\end{aligned}$$

Equations (1.8)-(1.10) now follow by comparison with Equation (1.7). □

Since f is a function in time, and the a_n, b_n represent contributions from different frequencies, the Fourier series can be thought of as a change of coordinates, from what we vaguely can call the *time domain* , to what we can call the *frequency domain* (or *Fourier domain*). We will call the basis $\mathcal{D}_{N,T}$ the *N'th order Fourier basis* for $V_{N,T}$.

We note that $\mathcal{D}_{N,T}$ is not an orthonormal basis; it is only orthogonal.

In the signal processing literature, Equation (1.7) is known as *the synthesis equation*, since the original function f is synthesized as a sum of trigonometric functions. Similarly, equations (1.8)-(1.10) are called *analysis equations.*

A major topic in harmonic analysis is to state conditions on f which guarantees the convergence of its Fourier series. We will not discuss this in detail here,

since it turns out that, by choosing N large enough, any reasonable periodic function can be approximated arbitrarily well by its Nth-order Fourier series approximation. More precisely, we have the following result for the convergence of the Fourier series, stated without proof.

Theorem 1.16. *Convergence of Fourier series.*
Suppose that f is periodic with period T, and that

- f has a finite set of discontinuities in each period.
- f contains a finite set of maxima and minima in each period.
- $\int_0^T |f(t)|dt < \infty$.

Then we have that $\lim_{N\to\infty} f_N(t) = f(t)$ for all t, except at those points t where f is not continuous.

The conditions in Theorem 1.16 are called the *Dirichlet conditions* for the convergence of the Fourier series. They are just one example of conditions that ensure the convergence of the Fourier series. There also exist much more general conditions that secure convergence. These can require deep mathematical theory in order to prove, depending on the generality.

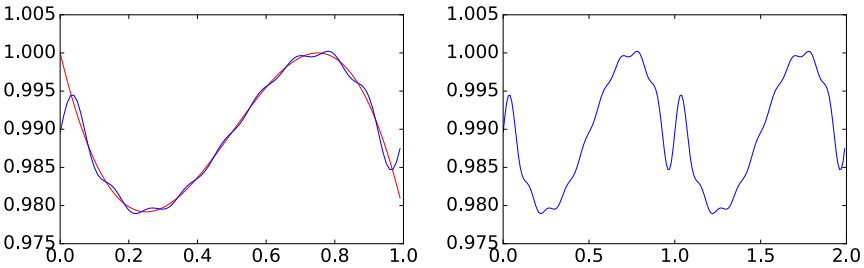

Figure 1.5: The cubic polynomial $f(x) = -\frac{1}{3}x^3 + \frac{1}{2}x^2 - \frac{3}{16}x + 1$ on the interval $[0, 1]$, together with its Fourier series approximation from $V_{9,1}$. The function and its Fourier series is shown left. The Fourier series on a larger interval is shown right.

An illustration of Theorem 1.16 is shown in Figure 1.5 where the cubic polynomial $f(x) = -\frac{1}{3}x^3 + \frac{1}{2}x^2 - \frac{3}{16}x + 1$ is approximated by a 9th order Fourier series. The trigonometric approximation is periodic with period 1 so the approximation becomes poor at the ends of the interval since the cubic polynomial is not periodic. The approximation is plotted on a larger interval in the right plot in Figure 1.5, where its periodicity is clearly visible.

Let us compute the Fourier series of some interesting functions.

Example 1.17. *Fourier coefficients of the square wave.*
Let us compute the Fourier coefficients of the square wave, as defined by Equation (1.1) in Example 1.10. If we first use Equation (1.8) we obtain

$$a_0 = \frac{1}{T}\int_0^T f_s(t)dt = \frac{1}{T}\int_0^{T/2} dt - \frac{1}{T}\int_{T/2}^T dt = 0.$$

Using Equation (1.9) we get

$$\begin{aligned} a_n &= \frac{2}{T}\int_0^T f_s(t)\cos(2\pi nt/T)dt \\ &= \frac{2}{T}\int_0^{T/2} \cos(2\pi nt/T)dt - \frac{2}{T}\int_{T/2}^T \cos(2\pi nt/T)dt \\ &= \frac{2}{T}\left[\frac{T}{2\pi n}\sin(2\pi nt/T)\right]_0^{T/2} - \frac{2}{T}\left[\frac{T}{2\pi n}\sin(2\pi nt/T)\right]_{T/2}^{T} \\ &= \frac{2}{T}\frac{T}{2\pi n}((\sin(n\pi) - \sin 0) - (\sin(2n\pi) - \sin(n\pi)) = 0. \end{aligned}$$

Finally, using Equation (1.10) we obtain

$$\begin{aligned} b_n &= \frac{2}{T}\int_0^T f_s(t)\sin(2\pi nt/T)dt \\ &= \frac{2}{T}\int_0^{T/2} \sin(2\pi nt/T)dt - \frac{2}{T}\int_{T/2}^T \sin(2\pi nt/T)dt \\ &= \frac{2}{T}\left[-\frac{T}{2\pi n}\cos(2\pi nt/T)\right]_0^{T/2} + \frac{2}{T}\left[\frac{T}{2\pi n}\cos(2\pi nt/T)\right]_{T/2}^{T} \\ &= \frac{2}{T}\frac{T}{2\pi n}((-\cos(n\pi) + \cos 0) + (\cos(2n\pi) - \cos(n\pi))) \\ &= \frac{2(1-\cos(n\pi)}{n\pi} \\ &= \begin{cases} 0, & \text{if } n \text{ is even;} \\ 4/(n\pi), & \text{if } n \text{ is odd.} \end{cases} \end{aligned}$$

In other words, only the b_n-coefficients with n odd in the Fourier series are nonzero. This means that the Fourier series of the square wave is

$$\frac{4}{\pi}\sin(2\pi t/T) + \frac{4}{3\pi}\sin(2\pi 3t/T) + \frac{4}{5\pi}\sin(2\pi 5t/T) + \frac{4}{7\pi}\sin(2\pi 7t/T) + \cdots . \quad (1.11)$$

With $N = 20$, there are 10 trigonometric terms in this sum. The corresponding Fourier series can be plotted on the same interval with the following code.

```
t = linspace(0, T, 100);
y = zeros(size(t));
for n = 1:2:19
    y = y + (4/(n*pi))*sin(2*pi*n*t/T);
end
plot(t,y)
```

The left plot in Figure 1.6 shows the Fourier series of the square wave when $T = 1/440$, and when $N = 20$. In the right plot the values of the first 100 Fourier coefficients b_n are shown, to see that they actually converge to zero. This is clearly necessary in order for the Fourier series to converge.

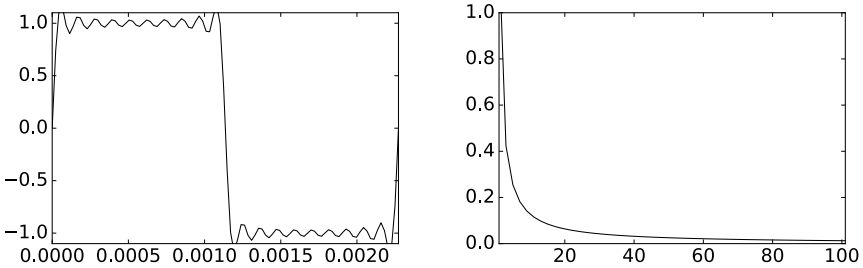

Figure 1.6: The Fourier series with $N = 20$ for the square wave of Example 1.17, and the values for the first 100 Fourier coefficients b_n.

Even though f oscillates regularly between -1 and 1 with period T, the discontinuities mean that it is far from the simple $\sin(2\pi t/T)$ which corresponds to a pure tone of frequency $1/T$. From Figure 1.6(b) we see that the dominant coefficient in the Fourier series is b_1, which tells us how much there is of the pure tone $\sin(2\pi t/T)$ in the square wave. This is not surprising since the square wave oscillates T times every second as well, but the additional nonzero coefficients pollute the pure sound. As we include more and more of these coefficients, we gradually approach the square wave, as shown for $N = 20$.

There is a connection between how fast the Fourier coefficients go to zero, and how we perceive the sound. A pure sine sound has only one nonzero coefficient, while the square wave Fourier coefficients decrease as $1/n$, making the sound less pleasant. This explains what we heard when we listened to the sound in Example 1.10. Also, it explains why we heard the same pitch as the pure tone, since the first frequency in the Fourier series has the same frequency as the pure tone we listened to, and since this had the highest value.

Let us listen to the Fourier series approximations of the square wave. For $N = 1$ and with $T = 1/440$ as above, it sounds like this. This sounds exactly like the pure sound with frequency 440Hz, as noted above. For $N = 5$ the Fourier series approximation sounds like this, and for $N = 9$ it sounds like this. Indeed, these sounds are more like the square wave itself, and as we increase N we can hear how the introduction of more frequencies gradually pollutes the sound more and more. In Exercise 2.5 you will be asked to write a program which verifies this.

Example 1.18. *Fourier coefficients of the triangle wave.*

Let us also compute the Fourier coefficients of the triangle wave, as defined by Equation (1.2) in Example 1.11. We now have

$$a_0 = \frac{1}{T}\int_0^{T/2} \frac{4}{T}\left(t - \frac{T}{4}\right) dt + \frac{1}{T}\int_{T/2}^{T} \frac{4}{T}\left(\frac{3T}{4} - t\right) dt.$$

Instead of computing this directly, it is quicker to see geometrically that the graph of f_t has as much area above as below the x-axis, so that this integral must be zero. Similarly, since f_t is symmetric about the midpoint $T/2$, and $\sin(2\pi nt/T)$ is antisymmetric about $T/2$, we have that $f_t(t)\sin(2\pi nt/T)$ also is antisymmetric about $T/2$, so that

$$\int_0^{T/2} f_t(t)\sin(2\pi nt/T)dt = -\int_{T/2}^{T} f_t(t)\sin(2\pi nt/T)dt.$$

This means that, for $n \geq 1$,

$$b_n = \frac{2}{T}\int_0^{T/2} f_t(t)\sin(2\pi nt/T)dt + \frac{2}{T}\int_{T/2}^{T} f_t(t)\sin(2\pi nt/T)dt = 0.$$

For the final coefficients, since both f and $\cos(2\pi nt/T)$ are symmetric about $T/2$, we get for $n \geq 1$,

$$\begin{aligned}
a_n &= \frac{2}{T}\int_0^{T/2} f_t(t)\cos(2\pi nt/T)dt + \frac{2}{T}\int_{T/2}^{T} f_t(t)\cos(2\pi nt/T)dt \\
&= \frac{4}{T}\int_0^{T/2} f_t(t)\cos(2\pi nt/T)dt = \frac{4}{T}\int_0^{T/2} \frac{4}{T}\left(t - \frac{T}{4}\right)\cos(2\pi nt/T)dt \\
&= \frac{16}{T^2}\int_0^{T/2} t\cos(2\pi nt/T)dt - \frac{4}{T}\int_0^{T/2} \cos(2\pi nt/T)dt \\
&= \frac{4}{n^2\pi^2}(\cos(n\pi) - 1) \\
&= \begin{cases} 0, & \text{if } n \text{ is even;} \\ -8/(n^2\pi^2), & \text{if } n \text{ is odd.} \end{cases}
\end{aligned}$$

where we have dropped the final tedious calculations (use integration by parts). From this it is clear that the Fourier series of the triangle wave is

$$-\frac{8}{\pi^2}\cos(2\pi t/T) - \frac{8}{3^2\pi^2}\cos(2\pi 3t/T) - \frac{8}{5^2\pi^2}\cos(2\pi 5t/T) - \frac{8}{7^2\pi^2}\cos(2\pi 7t/T) + \cdots. \tag{1.12}$$

In Figure 1.7 we have repeated the plots used for the square wave, for the triangle wave. As before, we have used $T = 1/440$. The figure clearly shows that the Fourier series coefficients decay much faster.

Let us also listen to different Fourier series approximations of the triangle wave. For $N = 1$ and with $T = 1/440$ as above, it sounds like this. Again, this sounds exactly like the pure sound with frequency 440Hz. For $N = 5$ the Fourier

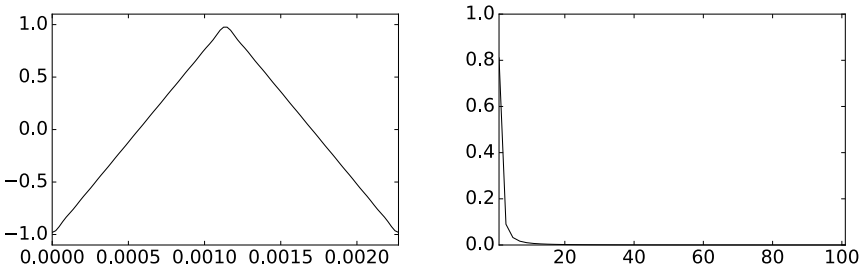

Figure 1.7: The Fourier series with $N = 20$ for the triangle wave of Example 1.18 and the values for the first 100 Fourier coefficients a_n.

series approximation sounds like this, and for $N = 9$ it sounds like this. Again these sounds are more like the triangle wave itself, and as we increase N we can hear that the introduction of more frequencies pollutes the sound. However, since the triangle wave Fourier coefficients decrease as $1/n^2$ instead of $1/n$ as for the square wave, the sound is, although unpleasant due to pollution by many frequencies, not as unpleasant as the square wave. Also, it converges faster to the triangle wave itself, as also can be heard. In Exercise 2.5 you will be asked to write a program which verifies this.

There is an important lesson to be learnt from the previous examples: Even if the signal is nice and periodic, it may not have a nice representation in terms of trigonometric functions. Thus, trigonometric functions may not be the best bases to use for expressing other functions. Unfortunately, many more such cases can be found, as the next example shows.

Example 1.19. *Fourier coefficients of a simple function.*

Let us consider a periodic function which is 1 on $[0, T_0]$, but 0 is on $[T_0, T]$. This is a signal with short duration when T_0 is small compared to T. We compute that $y_0 = T_0/T$, and

$$a_n = \frac{2}{T}\int_0^{T_0} \cos(2\pi nt/T)dt = \frac{1}{\pi n}\left[\sin(2\pi nt/T)\right]_0^{T_0} = \frac{\sin(2\pi nT_0/T)}{\pi n}$$

for $n \geq 1$. Similar computations hold for b_n. We see that $|a_n|$ is of the order $1/(\pi n)$, and that infinitely many n contribute, This function may be thought of as a simple building block, corresponding to a small time segment. However, we see that it is not a simple building block in terms of trigonometric functions. This time segment building block may be useful for restricting a function to smaller time segments, and later on we will see that it still can be useful.

1.2.1 Fourier series for symmetric and antisymmetric functions

In Example 1.17 we saw that the Fourier coefficients b_n vanished, resulting in a sine-series for the Fourier series of the square wave. Similarly, in Example 1.18 we saw that a_n vanished, resulting in a cosine-series for the triangle wave. This is not a coincident, and is captured by the following result, since the square wave was defined so that it was antisymmetric about 0, and the triangle wave so that it was symmetric about 0.

Theorem 1.20. *Symmetry and antisymmetry.*

If f is antisymmetric about 0 (that is, if $f(-t) = -f(t)$ for all t), then $a_n = 0$, so the Fourier series is actually a sine-series. If f is symmetric about 0 (which means that $f(-t) = f(t)$ for all t), then $b_n = 0$, so the Fourier series is actually a cosine-series.

Proof. Note first that we can write

$$a_n = \frac{2}{T}\int_{-T/2}^{T/2} f(t)\cos(2\pi nt/T)dt \qquad b_n = \frac{2}{T}\int_{-T/2}^{T/2} f(t)\sin(2\pi nt/T)dt,$$

i.e. we can change the integration bounds from $[0, T]$ to $[-T/2, T/2]$. This follows from the fact that all $f(t)$, $\cos(2\pi nt/T)$ and $\sin(2\pi nt/T)$ are periodic with period T.

Suppose first that f is symmetric. We obtain

$$\begin{aligned}
b_n &= \frac{2}{T}\int_{-T/2}^{T/2} f(t)\sin(2\pi nt/T)dt \\
&= \frac{2}{T}\int_{-T/2}^{0} f(t)\sin(2\pi nt/T)dt + \frac{2}{T}\int_{0}^{T/2} f(t)\sin(2\pi nt/T)dt \\
&= \frac{2}{T}\int_{-T/2}^{0} f(t)\sin(2\pi nt/T)dt - \frac{2}{T}\int_{0}^{-T/2} f(-t)\sin(-2\pi nt/T)dt \\
&= \frac{2}{T}\int_{-T/2}^{0} f(t)\sin(2\pi nt/T)dt - \frac{2}{T}\int_{-T/2}^{0} f(t)\sin(2\pi nt/T)dt = 0.
\end{aligned}$$

where we have made the substitution $u = -t$, and used that sin is antisymmetric. The case when f is antisymmetric can be proved in the same way, and is left as an exercise. □

In fact, the connection between symmetric and antisymmetric functions, and sine- and cosine series can be made even stronger by observing the following:

- Any cosine series $a_0 + \sum_{n=1}^{N} a_n \cos(2\pi nt/T)$ is a symmetric function.
- Any sine series $\sum_{n=1}^{N} b_n \sin(2\pi nt/T)$ is an antisymmetric function.

- Any periodic function can be written as a sum of a symmetric - and an antisymmetric function by writing $f(t) = \frac{f(t)+f(-t)}{2} + \frac{f(t)-f(-t)}{2}$.
- If $f_N(t) = a_0 + \sum_{n=1}^{N}(a_n \cos(2\pi nt/T) + b_n \sin(2\pi nt/T))$, then

$$\frac{f_N(t) + f_N(-t)}{2} = a_0 + \sum_{n=1}^{N} a_n \cos(2\pi nt/T)$$
$$\frac{f_N(t) - f_N(-t)}{2} = \sum_{n=1}^{N} b_n \sin(2\pi nt/T).$$

What you should have learned in this section.

- The inner product which we use for function spaces.
- Definition of the Fourier spaces, and the orthogonality of the Fourier basis.
- Fourier series approximations as best approximations.
- Formulas for the Fourier coefficients.
- Using the computer to plot Fourier series.
- For symmetric/antisymmetric functions, Fourier series are actually cosine/sine series.

Exercise 1.3: Riemann-integrable functions which are not square-integrable

Find a function f which is Riemann-integrable on $[0, T]$, and so that $\int_0^T f(t)^2 dt$ is infinite.

Exercise 1.4: When are Fourier spaces included in each other?

Given the two Fourier spaces V_{N_1,T_1}, V_{N_2,T_2}. Find necessary and sufficient conditions in order for $V_{N_1,T_1} \subset V_{N_2,T_2}$.

Exercise 1.5: antisymmetric functions are sine-series

Prove the second part of Theorem 1.20, i.e. show that if f is antisymmetric about 0 (i.e. $f(-t) = -f(t)$ for all t), then $a_n = 0$, i.e. the Fourier series is actually a sine-series.

Exercise 1.6: Fourier series for low-degree polynomials

Find the Fourier series coefficients of the periodic functions with period T defined by being $f(t) = t$, $f(t) = t^2$, and $f(t) = t^3$, on $[0, T]$.

Exercise 1.7: Fourier series for polynomials

Write down difference equations for finding the Fourier coefficients of $f(t) = t^{k+1}$ from those of $f(t) = t^k$, and write a program which uses this recursion. Use the program to verify what you computed in Exercise 1.6.

Exercise 1.8: Fourier series of a given polynomial

Use the previous exercise to find the Fourier series for $f(x) = -\frac{1}{3}x^3 + \frac{1}{2}x^2 - \frac{3}{16}x + 1$ on the interval $[0, 1]$. Plot the 9th order Fourier series for this function. You should obtain the plots from Figure 1.5.

1.3 Complex Fourier series

In Section 1.2 we saw how a function can be expanded in a series of sines and cosines. These functions are related to the complex exponential function via Eulers formula

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

where i is the imaginary unit with the property that $i^2 = -1$. Because the algebraic properties of the exponential function are much simpler than those of cos and sin, it is often an advantage to work with complex numbers, even though the given setting is real numbers. This is definitely the case in Fourier analysis. More precisely, we will make the substitutions

$$\cos(2\pi nt/T) = \frac{1}{2}\left(e^{2\pi int/T} + e^{-2\pi int/T}\right) \tag{1.13}$$

$$\sin(2\pi nt/T) = \frac{1}{2i}\left(e^{2\pi int/T} - e^{-2\pi int/T}\right) \tag{1.14}$$

in Definition 1.14. From these identities it is clear that the set of complex exponential functions $e^{2\pi int/T}$ also is a basis of periodic functions (with the same period) for $V_{N,T}$. We may therefore reformulate Definition 1.14 as follows:

Definition 1.21. *Complex Fourier basis.*
We define the set of functions

$$\mathcal{F}_{N,T} = \{e^{-2\pi iNt/T}, e^{-2\pi i(N-1)t/T}, \cdots, e^{-2\pi it/T}, \tag{1.15}$$

$$1, e^{2\pi it/T}, \cdots, e^{2\pi i(N-1)t/T}, e^{2\pi iNt/T}\}, \tag{1.16}$$

and call this the order N *complex Fourier basis* for $V_{N,T}$.

The function $e^{2\pi int/T}$ is also called a pure tone with frequency n/T, just as sines and cosines are. We would like to show that these functions also are orthogonal. To show this, we need to say more on the inner product we have defined by Equation (1.3). A weakness with this definition is that we have assumed real functions f and g, so that this can not be used for the complex exponential functions $e^{2\pi int/T}$. For general complex functions we will extend the definition of the inner product as follows:

$$\langle f, g\rangle = \frac{1}{T}\int_0^T f\bar{g}\,dt. \tag{1.17}$$

The associated norm now becomes

$$\|f\| = \sqrt{\frac{1}{T}\int_0^T |f(t)|^2 dt}. \tag{1.18}$$

The motivation behind Equation (1.17), where we have conjugated the second function, lies in the definition of an *inner product for vector spaces over complex numbers*. From before we are used to vector spaces over real numbers, but vector spaces over complex numbers are defined through the same set of axioms as for real vector spaces, only replacing real numbers with complex numbers. For complex vector spaces, the axioms defining an inner product are the same as for real vector spaces, except for that the axiom

$$\langle f, g\rangle = \langle g, f\rangle \tag{1.19}$$

is replaced with the axiom

$$\langle f, g\rangle = \overline{\langle g, f\rangle}, \tag{1.20}$$

i.e. a conjugation occurs when we switch the order of the functions. This new axiom can be used to prove the property $\langle f, cg\rangle = \bar{c}\langle f, g\rangle$, which is a somewhat different property from what we know for real inner product spaces. This follows by writing

$$\langle f, cg\rangle = \overline{\langle cg, f\rangle} = \overline{c\langle g, f\rangle} = \bar{c}\overline{\langle g, f\rangle} = \bar{c}\langle f, g\rangle.$$

Clearly the inner product given by (1.17) satisfies Axiom (1.20). With this definition it is quite easy to see that the functions $e^{2\pi int/T}$ are orthonormal. Using the orthogonal decomposition theorem we can therefore write

$$\begin{aligned} f_N(t) &= \sum_{n=-N}^{N} \frac{\langle f, e^{2\pi int/T}\rangle}{\langle e^{2\pi int/T}, e^{2\pi int/T}\rangle} e^{2\pi int/T} = \sum_{n=-N}^{N} \langle f, e^{2\pi int/T}\rangle e^{2\pi int/T} \\ &= \sum_{n=-N}^{N} \left(\frac{1}{T}\int_0^T f(t)e^{-2\pi int/T}dt\right) e^{2\pi int/T}. \end{aligned}$$

We summarize this in the following theorem, which is a version of Theorem 1.15 which uses the complex Fourier basis:

Theorem 1.22. *Complex Fourier coefficients.*

We denote by $y_{-N}, \ldots, y_0, \ldots, y_N$ the coordinates of f_N in the basis $\mathcal{F}_{N,T}$, i.e.

$$f_N(t) = \sum_{n=-N}^{N} y_n e^{2\pi i n t/T}. \tag{1.21}$$

The y_n are called the complex Fourier coefficients of f, and they are given by.

$$y_n = \langle f, e^{2\pi i n t/T} \rangle = \frac{1}{T} \int_0^T f(t) e^{-2\pi i n t/T} dt. \tag{1.22}$$

Let us consider some examples where we compute complex Fourier series.

Example 1.23. *Complex Fourier coefficients of a simple function.*

Let us consider the pure sound $f(t) = e^{2\pi i t/T_2}$ with period T_2, but let us consider it only on the interval $[0, T]$ instead, where $T < T_2$. Note that this f is not periodic, since we only consider the part $[0, T]$ of the period $[0, T_2]$. The Fourier coefficients are

$$\begin{aligned} y_n &= \frac{1}{T} \int_0^T e^{2\pi i t/T_2} e^{-2\pi i n t/T} dt = \frac{1}{2\pi i T(1/T_2 - n/T)} \left[e^{2\pi i t(1/T_2 - n/T)} \right]_0^T \\ &= \frac{1}{2\pi i (T/T_2 - n)} \left(e^{2\pi i T/T_2} - 1 \right). \end{aligned}$$

Here it is only the term $1/(T/T_2 - n)$ which depends on n, so that y_n can only be large when n is close T/T_2. In Figure 1.8 we have plotted $|y_n|$ for two different combinations of T, T_2.

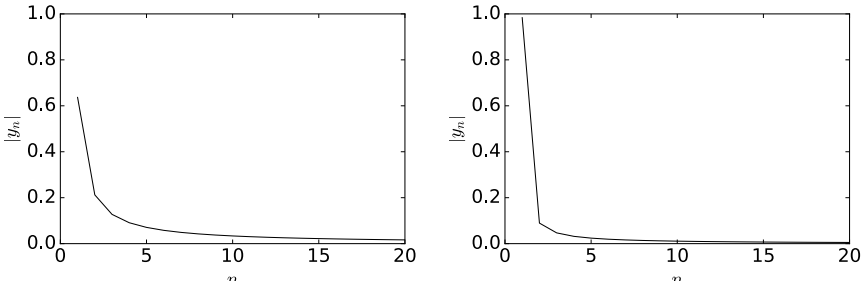

Figure 1.8: Plot of $|y_n|$ when $f(t) = e^{2\pi i t/T_2}$, and $T_2 > T$. Left: $T/T_2 = 0.5$. Right: $T/T_2 = 0.9$.

In both examples it is seen that many Fourier coefficients contribute, but this is more visible when $T/T_2 = 0.5$. When $T/T_2 = 0.9$, most conribution is

seen to be in the y_1-coefficient. This sounds reasonable, since f then is closest to the pure tone $f(t) = e^{2\pi it/T}$ of frequency $1/T$ (which in turn has $y_1 = 1$ and all other $y_n = 0$).

Apart from computing complex Fourier series, there is an important lesson to be learnt from the previous example: In order for a periodic function to be approximated by other periodic functions, their period must somehow match. Let us consider another example as well.

Example 1.24. *Complex Fourier coefficients of composite function.*

What often is the case is that a sound changes in content over time. Assume that it is equal to a pure tone of frequency n_1/T on $[0, T/2)$, and equal to a pure tone of frequency n_2/T on $[T/2, T)$, i.e.

$$f(t) = \begin{cases} e^{2\pi i n_1 t/T} & \text{on } [0, T_2] \\ e^{2\pi i n_2 t/T} & \text{on} [T_2, T) \end{cases}.$$

When $n \neq n_1, n_2$ we have that

$$\begin{aligned} y_n &= \frac{1}{T}\left(\int_0^{T/2} e^{2\pi i n_1 t/T} e^{-2\pi i n t/T} dt + \int_{T/2}^{T} e^{2\pi i n_2 t/T} e^{-2\pi i n t/T} dt\right) \\ &= \frac{1}{T}\left(\left[\frac{T}{2\pi i(n_1 - n)} e^{2\pi i (n_1 - n)t/T}\right]_0^{T/2} + \left[\frac{T}{2\pi i(n_2 - n)} e^{2\pi i (n_2 - n)t/T}\right]_{T/2}^{T}\right) \\ &= \frac{e^{\pi i(n_1 - n)} - 1}{2\pi i(n_1 - n)} + \frac{1 - e^{\pi i(n_2 - n)}}{2\pi i(n_2 - n)}. \end{aligned}$$

Let us restrict to the case when n_1 and n_2 are both even. We see that

$$y_n = \begin{cases} \frac{1}{2} + \frac{1}{\pi i(n_2 - n_1)} & n = n_1, n_2 \\ 0 & n \text{ even }, n \neq n_1, n_2 \\ \frac{n_1 - n_2}{\pi i(n_1 - n)(n_2 - n)} & n \text{ odd} \end{cases}$$

Here we have computed the cases $n = n_1$ and $n = n_2$ as above. In Figure 1.9 we have plotted $|y_n|$ for two different combinations of n_1, n_2.

We see from the figure that, when n_1, n_2 are close, the Fourier coefficients are close to those of a pure tone with $n \approx n_1, n_2$, but that also other frequencies contribute. When n_1, n_2 are further apart, we see that the Fourier coefficients are like the sum of the two base frequencies, but that other frequencies contribute also here.

There is an important lesson to be learnt from this as well: We should be aware of changes in a sound over time, and it may not be smart to use a frequency representation over a large interval when we know that there are simpler frequency representations on the smaller intervals. The following example shows that, in some cases it is not necessary to compute the Fourier integrals at all, in order to compute the Fourier series.

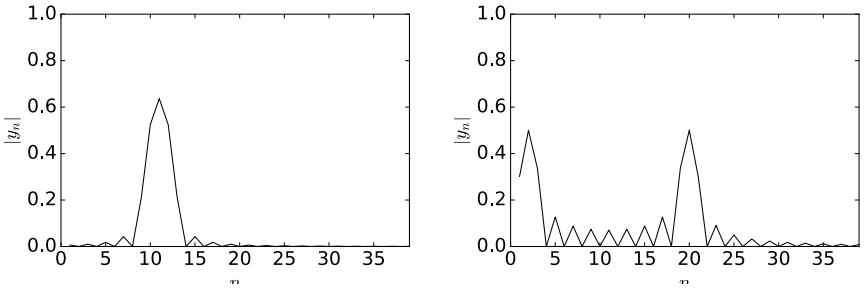

Figure 1.9: Plot of $|y_n|$ when we have two different pure tones at the different parts of a period. Left: $n_1 = 10$, $n_2 = 12$. Right: $n_1 = 2$, $n_2 = 20$.

Example 1.25. *Complex Fourier coefficients of* $f(t) = \cos^3(2\pi t/T)$.

Let us compute the complex Fourier series of the function $f(t) = \cos^3(2\pi t/T)$, where T is the period of f. We can write

$$\begin{aligned}\cos^3(2\pi t/T) &= \left(\frac{1}{2}(e^{2\pi it/T} + e^{-2\pi it/T})\right)^3 \\ &= \frac{1}{8}(e^{2\pi i3t/T} + 3e^{2\pi it/T} + 3e^{-2\pi it/T} + e^{-2\pi i3t/T}) \\ &= \frac{1}{8}e^{2\pi i3t/T} + \frac{3}{8}e^{2\pi it/T} + \frac{3}{8}e^{-2\pi it/T} + \frac{1}{8}e^{-2\pi i3t/T}.\end{aligned}$$

From this we see that the complex Fourier series is given by $y_1 = y_{-1} = \frac{3}{8}$, and that $y_3 = y_{-3} = \frac{1}{8}$. In other words, it was not necessary to compute the Fourier integrals in this case, and we see that the function lies in $V_{3,T}$, i.e. there are finitely many terms in the Fourier series. In general, if the function is some trigonometric function, we can often use trigonometric identities to find an expression for the Fourier series.

If we reorder the real and complex Fourier bases so that the two functions $\{\cos(2\pi nt/T), \sin(2\pi nt/T)\}$ and $\{e^{2\pi int/T}, e^{-2\pi int/T}\}$ have the same index in the bases, equations (1.13)-(1.14) give us that the change of coordinates matrix [4] from $\mathcal{D}_{N,T}$ to $\mathcal{F}_{N,T}$, denoted $P_{\mathcal{F}_{N,T} \leftarrow \mathcal{D}_{N,T}}$, is represented by repeating the matrix

$$\frac{1}{2}\begin{pmatrix} 1 & 1/i \\ 1 & -1/i \end{pmatrix}$$

along the diagonal (with an additional 1 for the constant function 1). In other words, since a_n, b_n are coefficients relative to the real basis and y_n, y_{-n} the corresponding coefficients relative to the complex basis, we have for $n > 0$,

[4] See Section 4.7 in [20], to review the mathematics behind change of coordinates.

$$\begin{pmatrix} y_n \\ y_{-n} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1/i \\ 1 & -1/i \end{pmatrix} \begin{pmatrix} a_n \\ b_n \end{pmatrix}.$$

This can be summarized by the following theorem:

Theorem 1.26. *Change of coefficients between real and complex Fourier bases.*
The complex Fourier coefficients y_n and the real Fourier coefficients a_n, b_n of a function f are related by

$$\begin{aligned} y_0 &= a_0, \\ y_n &= \frac{1}{2}(a_n - ib_n), \\ y_{-n} &= \frac{1}{2}(a_n + ib_n), \end{aligned}$$

for $n = 1, \ldots, N$.

Combining with Theorem 1.20, Theorem 1.26 can help us state properties of complex Fourier coefficients for symmetric- and antisymmetric functions. We look into this in Exercise 1.16.

Due to the somewhat nicer formulas for the complex Fourier coefficients when compared to the real Fourier coefficients, we will write most Fourier series in complex form in the following.

What you should have learned in this section.

- The complex Fourier basis and its orthonormality.

Exercise 1.9: Orthonormality of Complex Fourier basis

Show that the complex functions $e^{2\pi int/T}$ are orthonormal.

Exercise 1.10: Complex Fourier series of $f(t) = \sin^2(2\pi t/T)$

Compute the complex Fourier series of the function $f(t) = \sin^2(2\pi t/T)$.

Exercise 1.11: Complex Fourier series of polynomials

Repeat Exercise 1.6 in Section 1.2, computing the complex Fourier series instead of the real Fourier series.

Exercise 1.12: Complex Fourier series and Pascals triangle

In this exercise we will find a connection with certain Fourier series and the rows in Pascal's triangle.

a) Show that both $\cos^n(t)$ and $\sin^n(t)$ are in $V_{N,2\pi}$ for $1 \leq n \leq N$.

b) Write down the N'th order complex Fourier series for $f_1(t) = \cos t$, $f_2(t) = \cos^2 t$, og $f_3(t) = \cos^3 t$.

c) In (b) you should be able to see a connection between the Fourier coefficients and the three first rows in Pascal's triangle. Formulate and prove a general relationship between row n in Pascal's triangle and the Fourier coefficients of $f_n(t) = \cos^n t$.

Exercise 1.13: Complex Fourier coefficients of the square wave

Compute the complex Fourier coefficients of the square wave using Equation (1.22), i.e. repeat the calculations from Example 1.17 for the complex case. Use Theorem 1.26 to verify your result.

Exercise 1.14: Complex Fourier coefficients of the triangle wave

Repeat Exercise 1.13 for the triangle wave.

Exercise 1.15: Complex Fourier coefficients of low-degree polynomials

Use Equation (1.22) to compute the complex Fourier coefficients of the periodic functions with period T defined by, respectively, $f(t) = t$, $f(t) = t^2$, and $f(t) = t^3$, on $[0, T]$. Use Theorem 1.26 to verify your calculations from Exercise 1.6.

Exercise 1.16: Complex Fourier coefficients for symmetric and antisymmetric functions

In this exercise we will prove a version of Theorem 1.20 for complex Fourier coefficients.

a) If f is symmetric about 0, show that y_n is real, and that $y_{-n} = y_n$.

b) If f is antisymmetric about 0, show that the y_n are purely imaginary, $y_0 = 0$, and that $y_{-n} = -y_n$.

c) Show that $\sum_{n=-N}^{N} y_n e^{2\pi i n t/T}$ is symmetric when $y_{-n} = y_n$ for all n, and rewrite it as a cosine-series.

d) Show that $\sum_{n=-N}^{N} y_n e^{2\pi i n t/T}$ is antisymmetric when $y_0 = 0$ and $y_{-n} = -y_n$ for all n, and rewrite it as a sine-series.

1.4 Some properties of Fourier series

We continue by establishing some important properties of Fourier series, in particular the Fourier coefficients for some important functions. In these lists, we will use the notation $f \to y_n$ to indicate that y_n is the n'th (complex) Fourier coefficient of $f(t)$.

Theorem 1.27. *Fourier series pairs.*
The functions 1, $e^{2\pi int/T}$, and $\chi_{-a,a}$ have the Fourier coefficients

$$\begin{aligned} 1 &\to \boldsymbol{e}_0 = (1, 0, 0, 0 \ldots,) \\ e^{2\pi int/T} &\to \boldsymbol{e}_n = (0, 0, \ldots, 1, 0, 0, \ldots) \\ \chi_{-a,a} &\to \frac{\sin(2\pi na/T)}{\pi n}. \end{aligned}$$

The 1 in $\boldsymbol{e}_n$ is at position n and the function $\chi_{-a,a}$ is the characteristic function of the interval $[-a, a]$, defined by

$$\chi_{-a,a}(t) = \begin{cases} 1, & \text{if } t \in [-a, a]; \\ 0, & \text{otherwise.} \end{cases}$$

The first two pairs are easily verified, so the proofs are omitted. The case for $\chi_{-a,a}$ is very similar to the square wave, but easier to prove, and therefore also omitted.

Theorem 1.28. *Fourier series properties.*
The mapping $f \to y_n$ is linear: if $f \to x_n$, $g \to y_n$, then

$$af + bg \to ax_n + by_n$$

For all n. Moreover, if f is real and periodic with period T, the following properties hold:

1. $y_n = \overline{y_{-n}}$ for all n.
2. If $f(t) = f(-t)$ (i.e. f is symmetric), then all y_n are real, so that b_n are zero and the Fourier series is a cosine series.
3. If $f(t) = -f(-t)$ (i.e. f is antisymmetric), then all y_n are purely imaginary, so that the a_n are zero and the Fourier series is a sine series.
4. If $g(t) = f(t - d)$ (i.e. g is the function f delayed by d) and $f \to y_n$, then $g \to e^{-2\pi ind/T} y_n$.
5. If $g(t) = e^{2\pi idt/T} f(t)$ with d an integer, and $f \to y_n$, then $g \to y_{n-d}$.
6. Let d be a number. If $f \to y_n$, then $f(d+t) = f(d-t)$ for all t if and only if the argument of y_n is $-2\pi nd/T$ for all n.

Proof. The proof of linearity is left to the reader. Property 1 follows immediately by writing

$$\begin{aligned} y_n &= \frac{1}{T}\int_0^T f(t)e^{-2\pi int/T}dt = \overline{\frac{1}{T}\int_0^T f(t)e^{2\pi int/T}dt} \\ &= \overline{\frac{1}{T}\int_0^T f(t)e^{-2\pi i(-n)t/T}dt} = \overline{y_{-n}}. \end{aligned}$$

Also, if $g(t) = f(-t)$, we have that

$$\begin{aligned} \frac{1}{T}\int_0^T g(t)e^{-2\pi int/T}dt &= \frac{1}{T}\int_0^T f(-t)e^{-2\pi int/T}dt = -\frac{1}{T}\int_0^{-T} f(t)e^{2\pi int/T}dt \\ &= \frac{1}{T}\int_0^T f(t)e^{2\pi int/T}dt = \overline{y_n}. \end{aligned}$$

The first part of property 2 follows from this. The second part follows directly by noting that

$$y_n e^{2\pi int/T} + y_{-n}e^{-2\pi int/T} = y_n(e^{2\pi int/T} + e^{-2\pi int/T}) = 2y_n\cos(2\pi nt/T),$$

or by invoking Theorem 1.20. Property 3 is proved in a similar way. To prove property 4, we observe that the Fourier coefficients of $g(t) = f(t-d)$ are

$$\begin{aligned} \frac{1}{T}\int_0^T g(t)e^{-2\pi int/T}dt &= \frac{1}{T}\int_0^T f(t-d)e^{-2\pi int/T}dt \\ &= \frac{1}{T}\int_0^T f(t)e^{-2\pi in(t+d)/T}dt \\ &= e^{-2\pi ind/T}\frac{1}{T}\int_0^T f(t)e^{-2\pi int/T}dt = e^{-2\pi ind/T}y_n. \end{aligned}$$

For property 5 we observe that the Fourier coefficients of $g(t) = e^{2\pi idt/T}f(t)$ are

$$\begin{aligned} \frac{1}{T}\int_0^T g(t)e^{-2\pi int/T}dt &= \frac{1}{T}\int_0^T e^{2\pi idt/T}f(t)e^{-2\pi int/T}dt \\ &= \frac{1}{T}\int_0^T f(t)e^{-2\pi i(n-d)t/T}dt = y_{n-d}. \end{aligned}$$

If $f(d+t) = f(d-t)$ for all t, we define the function $g(t) = f(t+d)$ which is symmetric about 0, so that it has real Fourier coefficients. But then the Fourier coefficients of $f(t) = g(t-d)$ are $e^{-2\pi ind/T}$ times the (real) Fourier coefficients of g by property 4. It follows that y_n, the Fourier coefficients of f, has argument $-2\pi nd/T$. The proof in the other direction follows by noting that any function where the Fourier coefficients are real must be symmetric about 0, once the Fourier series is known to converge. This proves property 6. □

Let us analyze these properties, to see that they match the notion we already have for frequencies and sound. We will say that two sounds "essentially are the same" if the absolute values of each Fourier coefficient are equal. Note that this does not mean that the sounds sound the same, it merely says that the contributions at different frequencies are comparable.

The first property says that the positive and negative frequencies in a (real) sound essentially are the same. The second says that, when we play a sound backwards, the frequency content is essentially the same. This is certainly the case for all pure sounds. The third property says that, if we delay a sound, the frequency content also is essentially the same. This also matches our intuition on sound, since we think of the frequency representation as something which is time-independent. The fourth property says that, if we multiply a sound with a pure tone, the frequency representation is shifted (delayed), according to the value of the frequency. This is something we see in early models for the transmission of audio, where an audio signal is transmitted after having been multiplied with what is called a 'carrier wave'. You can think of the carrier signal as a pure tone. The result is a signal where the frequencies have been shifted with the frequency of the carrier wave. The point of shifting the frequency of the transmitted signal is to make it use a frequency range in which one knows that other signals do not interfere. The last property looks a bit mysterious. We will not have use for this property before the next chapter.

From Theorem 1.28 we also see that there exist several cases of duality between a function and its Fourier series:

- Delaying a function corresponds to multiplying the Fourier coefficients with a complex exponential. Vice versa, multiplying a function with a complex exponential corresponds to delaying the Fourier coefficients.
- Symmetry/antisymmetry for a function corresponds to the Fourier coefficients being real/purely imaginary. Vice versa, a function which is real has Fourier coefficients which are conjugate symmetric.

Actually, one can show that these dualities are even stronger if we had considered Fourier series of complex functions instead of real functions. We will not go into this.

1.4.1 Rate of convergence for Fourier series

We have earlier mentioned criteria which guarantee that the Fourier series converges. Another important topic is the rate of convergence, given that it actually converges. If the series converges quickly, we may only need a few terms in the Fourier series to obtain a reasonable approximation. We have already seen examples which illustrate different convergence rates: The square wave seemed to have very slow convergence rate near the discontinuities, while the triangle wave did not seem to have the same problem.

Before discussing results concerning convergence rates we consider a simple lemma which will turn out to be useful.

Lemma 1.29. *The order of computing Fourier series and differentiation does not matter.*

Assume that f is differentiable. Then $(f_N)'(t) = (f')_N(t)$. In other words, the derivative of the Fourier series equals the Fourier series of the derivative.

Proof. We first compute

$$\begin{aligned}\langle f, e^{2\pi int/T}\rangle &= \frac{1}{T}\int_0^T f(t)e^{-2\pi int/T}dt \\ &= \frac{1}{T}\left(\left[-\frac{T}{2\pi in}f(t)e^{-2\pi int/T}\right]_0^T + \frac{T}{2\pi in}\int_0^T f'(t)e^{-2\pi int/T}dt\right) \\ &= \frac{T}{2\pi in}\frac{1}{T}\int_0^T f'(t)e^{-2\pi int/T}dt = \frac{T}{2\pi in}\langle f', e^{2\pi int/T}\rangle.\end{aligned}$$

where we used integration by parts, and that $-\frac{T}{2\pi in}f(t)e^{-2\pi int/T}$ are periodic with period T. It follows that $\langle f, e^{2\pi int/T}\rangle = \frac{T}{2\pi in}\langle f', e^{2\pi int/T}\rangle$. From this we get that

$$\begin{aligned}(f_N)'(t) &= \left(\sum_{n=-N}^{N}\langle f, e^{2\pi int/T}\rangle e^{2\pi int/T}\right)' = \frac{2\pi in}{T}\sum_{n=-N}^{N}\langle f, e^{2\pi int/T}\rangle e^{2\pi int/T} \\ &= \sum_{n=-N}^{N}\langle f', e^{2\pi int/T}\rangle e^{2\pi int/T} = (f')_N(t).\end{aligned}$$

where we substituted the connection between the inner products we just found. □

Example 1.30. *Computing the Fourier series of the triangle wave through differentiation of the square wave.*

The connection between the Fourier series of the function and its derivative can be used to simplify the computation of Fourier series for new functions. Let us see how we can use this to compute the Fourier series of the triangle wave, which was quite a tedious job in Example 1.18. However, the relationship $f_t'(t) = \frac{4}{T}f_s(t)$ is straightforward to see from the plots of the square wave f_s and the triangle wave f_t. From this relationship and from Equation (1.11) for the Fourier series of the square wave it follows that

$$((f_t)')_N(t) = \frac{4}{T}\left(\frac{4}{\pi}\sin(2\pi t/T) + \frac{4}{3\pi}\sin(2\pi 3t/T) + \frac{4}{5\pi}\sin(2\pi 5t/T) + \cdots\right).$$

If we integrate this we obtain

$$(f_t)_N(t) = -\frac{8}{\pi^2}\left(\cos(2\pi t/T) + \frac{1}{3^2}\cos(2\pi 3t/T) + \frac{1}{5^2}\cos(2\pi 5t/T) + \cdots\right) + C.$$

What remains is to find the integration constant C. This is simplest found if we set $t = T/4$, since then all cosine terms are 0. Clearly then $C = 0$, and we arrive at the same expression as in Equation (1.12) for the Fourier series of the triangle wave. This approach clearly had less computations involved. There is a minor point here which we have not addressed: the triangle wave is not differentiable at two points, as required by Lemma 1.29. It is, however, not too difficult to see that this result still holds in cases where we have a finite number of nondifferentiable points only.

We get the following corollary to Lemma 1.29:

Corollary 1.31. *Connection between the Fourier coefficients of $f(t)$ and $f'(t)$.*
If the complex Fourier coefficients of f are y_n and f is differentiable, then the Fourier coefficients of $f'(t)$ are $\frac{2\pi in}{T} y_n$.

If we turn this around, we note that the Fourier coefficients of $f(t)$ are $T/(2\pi in)$ times those of $f'(t)$. If f is s times differentiable, we can repeat this argument to show that the Fourier coefficients of $f(t)$ are $\big(T/(2\pi in)\big)^s$ times those of $f^{(s)}(t)$. In other words, the Fourier coefficients of a function which is many times differentiable decay to zero very fast.

Observation 1.32. *Convergence speed of differentiable functions.*
The Fourier series converges quickly when the function is many times differentiable.

An illustration is found in examples 1.17 and 1.18, where we saw that the Fourier series coefficients for the triangle wave converged more quickly to zero than those of the square wave. This is explained by the fact that the square wave is discontinuous, while the triangle wave is continuous with a discontinuous first derivative. Also, the functions considered in examples 1.23 and 1.24 are not continuous, which partially explain why we there saw contributions from many frequencies.

The requirement of continuity in order to obtain quickly converging Fourier series may seem like a small problem. However, often the function is not defined on the whole real line: it is often only defined on the interval $[0, T)$. If we extend this to a periodic function on the whole real line, by repeating one period as shown in the left plot in Figure 1.10, there is no reason why the new function should be continuous at the boundaries $0, T, 2T$ etc., even though the function we started with may be continuous on $[0, T)$. This would require that $f(0) = \lim_{t \to T} f(t)$. If this does not hold, the function may not be well approximated with trigonometric functions, due to a slowly convergence Fourier series.

We can therefore ask ourselves the following question:

Idea 1.33. *Continuous Extension.*
Assume that f is continuous on $[0, T)$. Can we construct another periodic function which agrees with f on $[0, T]$, and which is both continuous and periodic (maybe with period different from T)?

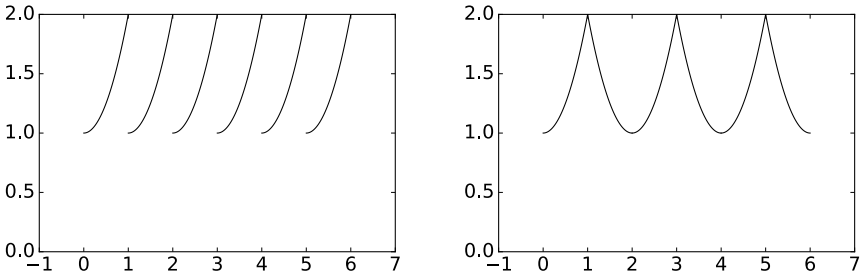

Figure 1.10: Two different extensions of f to a periodic function on the whole real line. Periodic extension (left) and symmetric extension (right).

If this is possible the Fourier series of the new function could produce better approximations for f. It turns out that the following extension strategy does the job:

Definition 1.34. *Symmetric extension of a function.*
Let f be a function defined on $[0, T]$. By the *symmetric extension* of f, denoted $\breve{f}$, we mean the function defined on $[0, 2T]$ by

$$\breve{f}(t) = \begin{cases} f(t), & \text{if } 0 \leq t \leq T; \\ f(2T - t), & \text{if } T < t \leq 2T. \end{cases}$$

Clearly the following holds:

Theorem 1.35. *Continuous Extension.*
If f is continuous on $[0, T]$, then $\breve{f}$ is continuous on $[0, 2T]$, and $\breve{f}(0) = \breve{f}(2T)$. If we extend $\breve{f}$ to a periodic function on the whole real line (which we also will denote by $\breve{f}$), this function is continuous, agrees with f on $[0, T)$, and is a symmetric function.

This also means that the Fourier series of $\breve{f}$ is a cosine series, so that it is determined by the cosine-coefficients a_n. The symmetric extension of f is shown in the right plot in Figure 1.10. $\breve{f}$ is symmetric since, for $0 \leq t \leq T$,

$$\breve{f}(-t) = \breve{f}(2T - t) = f(2T - (2T - t)) = f(t) = \breve{f}(t).$$

In summary, we now have two possibilities for approximating a function f defined only on $[0, T)$, where the latter addresses a shortcoming of the first:

- By the Fourier series of f
- By the Fourier series of $\breve{f}$ restricted to $[0, T)$ (which actually is a cosine-series)

Example 1.36. *Periodic extension.*

Let f be the function with period T defined by $f(t) = 2t/T - 1$ for $0 \leq t < T$. In each period the function increases linearly from -1 to 1. Because f is discontinuous at the boundaries, we would expect the Fourier series to converge slowly. The Fourier series is a sine-series since f is antisymmetric, and we can compute b_n as

$$\begin{aligned} b_n &= \frac{2}{T}\int_0^T \frac{2}{T}\left(t - \frac{T}{2}\right)\sin(2\pi nt/T)dt = \frac{4}{T^2}\int_0^T \left(t - \frac{T}{2}\right)\sin(2\pi nt/T)dt \\ &= \frac{4}{T^2}\int_0^T t\sin(2\pi nt/T)dt - \frac{2}{T}\int_0^T \sin(2\pi nt/T)dt = -\frac{2}{\pi n}, \end{aligned}$$

so that

$$f_N(t) = -\sum_{n=1}^{N} \frac{2}{n\pi}\sin(2\pi nt/T),$$

which indeed converges slowly to 0. Let us now instead consider the symmetric extension of f. Clearly this is the triangle wave with period $2T$, and the Fourier series of this was

$$(\breve{f})_N(t) = -\sum_{n \leq N,\ n \text{ odd}} \frac{8}{n^2\pi^2}\cos(2\pi nt/(2T)).$$

The second series clearly converges faster than the first, since its Fourier coefficients are $a_n = -8/(n^2\pi^2)$ (with n odd), while the Fourier coefficients in the first series are $b_n = -2/(n\pi)$.

If we use $T = 1/440$, the symmetric extension has period $1/220$, which gives a triangle wave where the first term in the Fourier series has frequency 220Hz. Listening to this we should hear something resembling a 220Hz pure tone, since the first term in the Fourier series is the most dominating in the triangle wave. Listening to the periodic extension we should hear a different sound. The first term in the Fourier series has frequency 440Hz, but this drounds a bit in the contribution of the other terms in the Fourier series, due to the slow convergence of the Fourier series, just as for the square wave.

The Fourier series with $N = 7$ terms of both f itself and the symmetric extensions of f are shown in Figure 1.11. It is clear from the plot that the Fourier series for f itself is not a very good approximation, while we cannot differentiate between the Fourier series and the function itself for the symmetric extension.

What you should have learned in this section.

- Simple Fourier series pairs.
- Certain properties of Fourier series, for instance how delay of a function or multiplication with a complex exponential affect the Fourier coefficients.

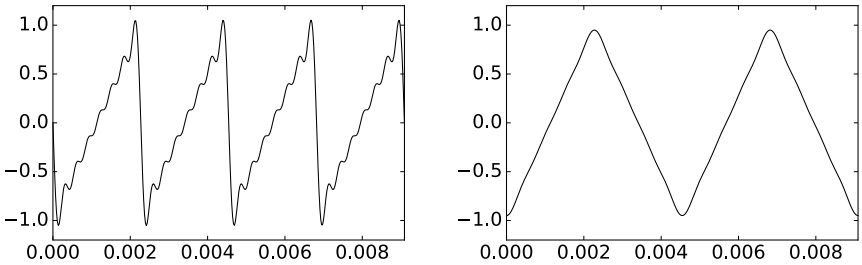

Figure 1.11: The Fourier series with $N = 7$ terms of the periodic (left) and symmetric (right) extensions of the function in Example 1.36.

- The convergence rate of a Fourier series depends on the regularity of the function. How this motivates the symmetric extension of a function.

Exercise 1.17: Fourier series of a delayed square wave

Define the function f with period T on $[-T/2, T/2)$ by

$$f(t) = \begin{cases} 1, & \text{if } -T/4 \leq t < T/4; \\ -1, & \text{if } T/4 \leq |t| < T/2. \end{cases}$$

f is just the square wave, delayed with $d = -T/4$. Compute the Fourier coefficients of f directly, and use Property 4 in Theorem 1.28 to verify your result.

Exercise 1.18: Find function from its Fourier series

Find a function f which has the complex Fourier series

$$\sum_{n \text{ odd}} \frac{4}{\pi(n+4)} e^{2\pi i n t/T}.$$

Hint. Attempt to use one of the properties in Theorem 1.28 on the Fourier series of the square wave.

Exercise 1.19: Relation between complex Fourier coefficients of f and cosine-coefficients of $\breve{f}$

Show that the complex Fourier coefficients y_n of f, and the cosine-coefficients a_n of $\breve{f}$ are related by $a_{2n} = y_n + y_{-n}$. This result is not enough to obtain the entire Fourier series of $\breve{f}$, but at least it gives us half of it.

1.5 Operations on sound: filters

It is easy to see how we can use Fourier coefficients to analyse or improve sound: Noise in a sound often corresponds to the presence of some high frequencies with large coefficients, and by removing these, we remove the noise. For example, we could set all the coefficients except the first one to zero. This would change the unpleasant square wave to the pure tone $\sin(2\pi 440t)$, which we started our experiments with. Doing so is an example of an important operation on sound called *filtering*:

Definition 1.37. *Analog filters.*

An operation on sound is called an *analog filter* if it preserves the different frequencies in the sound. In other words, s is an analog filter if, for any sound $f = \sum_\nu c(\nu)e^{2\pi i\nu t}$, the output $s(f)$ is a sound which can be written on the form

$$s(f) = s\left(\sum_\nu c(\nu)e^{2\pi i\nu t}\right) = \sum_\nu c(\nu)\lambda_s(\nu)e^{2\pi i\nu t},$$

where $\lambda_s(\nu)$ is a function describing how s treats the different frequencies. $\lambda_s(\nu)$ uniquely determines s, and is also called the *frequency response* of s.

The following is clear:

Theorem 1.38. *Properties of analog filters.*

The following hold for an analog filter s:

- When f is periodic with period T, $s(f)$ is also periodic with period T.
- When $s(f)$ we have that $(s(f))_N = s(f_N)$, i.e. s maps the N'th order Fourier series of f to the N'th order Fourier series of $s(f)$.
- Any pure tone is an eigenvector of s.

The analog filters we will look at have the following form:

Theorem 1.39. *Convolution kernels.*

Assume that $g \in L^1(\mathbb{R})$. The operation

$$f(t) \to h(t) = \int_{-\infty}^{\infty} g(s)f(t-s)ds. \tag{1.23}$$

is an analog filter. Analog filters which can be expressed like this are also called *convolutions*. Also

- When $f \in L^2(\mathbb{R})$, then $h \in L^2(\mathbb{R})$.
- The frequency response of the filter is $\lambda_s(\nu) = \int_\infty^\infty g(s)e^{-2\pi i\nu s}ds$

The function g is also called a *convolution kernel* . We also write s_g for the analog filter with convolution kernel g.

The name convolution kernel comes from the fact that filtering operations are also called convolution operations in the literature. In the analog filters we will look at later, the convolution kernel will always have *compact support* . The *support* of a function f defined on a subset I of $\mathbb{R}$ is given by the closure of the set of points where the function is nonzero,

$$\text{supp}(f) = \overline{\{t \in I \mid f(t) \neq 0\}}.$$

Compact support simply means that the support is contained in some interval on the form $[a, b]$ for some constants a, b. In this case the filter takes the form $f(t) \to h(t) = \int_a^b g(s)f(t-s)ds$. Also note that the integral above may not exist, so that one needs to put some restrictions on the functions, such that $f \in L^2(\mathbb{R})$. Note also that all analog filters may not be expressed as convolutions.

Proof. We compute

$$s(e^{2\pi i\nu t}) = \int_{-\infty}^{\infty} g(s)e^{2\pi i\nu(t-s)}ds = \int_{-\infty}^{\infty} g(s)e^{-2\pi i\nu s}ds e^{2\pi i\nu t} = \lambda_s(f)e^{2\pi i\nu t},$$

which shows that s is a filter with the stated frequency response. That $h \in L^2(\mathbb{R})$, when $f \in L^2(\mathbb{R})$ follows from Minkowski's inequality for integrals [12]. □

The function g is arbitrary, so that this strategy leads to a wide class of analog filters. We may ask the question of whether the general analog filter always has this form. We will not go further into this, although one can find partially affirmative answers to this question.

We also need to say something about the connection between filters and symmetric functions. We saw that the symmetric extension of a function took the form of a cosine-series, and that this converged faster to the symmetric extension than the Fourier series did to the function. If a filter preserves cosine-series it will also preserve symmetric extensions, and therefore also map fast-converging Fourier series to fast-converging Fourier series. The following result will be useful in this respect:

Theorem 1.40. *Properties of filters.*

If the frequency response of a filter satisfies $\lambda_s(\nu) = \lambda_s(-\nu)$ for all frequencies ν, then the filter preserves cosine series and sine series.

Proof. We have that

$$\begin{aligned} s(\cos(2\pi nt/T)) &= s\left(\frac{1}{2}(e^{2\pi int/T} + e^{-2\pi int/T})\right) \\ &= \frac{1}{2}\lambda_s(n/T)e^{2\pi int/T} + \frac{1}{2}\lambda_s(-n/T)e^{-2\pi int/T} \\ &= \lambda_s(n/T)\left(\frac{1}{2}(e^{2\pi int/T} + e^{-2\pi int/T})\right) = \lambda_s(n/T)\cos(2\pi nt/T). \end{aligned}$$

This means that s preserves cosine-series. A similar computation holds for sine-series holds as well. □

An analog filter where $\lambda_s(\nu) = \lambda_s(-\nu)$ is also called a *symmetric filter*. As an example, consider the analog filter $s(f_1) = \int_{-a}^{a} g(s) f_1(t-s) ds$ where g is symmetric around 0 and supported on $[-a, a]$. s is a symmetric filter since

$$\lambda_s(\nu) = \int_{-a}^{a} g(s) e^{-2\pi i \nu s} ds = \int_{-a}^{a} g(s) e^{2\pi i \nu s} ds = \lambda_s(-\nu).$$

Filters are much used in practice, but the way we have defined them here makes them not very useful for computation. We will handle the problem of making filters suitable for computation in Chapter 3.

1.6 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'etudes de television et telecommunications), IRT (Institut fur 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. It leaves a substantial amount of freedom in the encoder, so that different encoders can exploit properties of sound in various ways, in order to alter the sound in removing inaudible components therein. 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 is split into *frequency bands*, each band corresponding to a particular frequency range. In the simplest model, 32 frequency bands are used. A frequency analysis of the sound, based on what is called a *psycho-acoustic model*, is the basis for further transformation of these bands. The psycho-acoustic model computes 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 thresholds* are assigned for each band. The computed masking thresholds have to do with a phenomenon called *masking*. 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 omitted by the encoder, since the model says that the sound should be inaudible.

Masking effects are just one example of what is called psycho-acoustic effects, and all such effects can be taken into account in a psycho-acoustic model. 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 they 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).

1.7 Summary

We discussed the basic question of what is sound is, and concluded that sound could be modeled as a sum of frequency components. If the function was periodic we could define its Fourier series, which can be thought of as an approximation scheme for periodic functions using finite-dimensional spaces of trigonometric functions. We established the basic properties of Fourier series, and some duality relationships between the function and its Fourier series. We have also computed the Fourier series of the square wave and the triangle wave, and we saw that we could speed up the convergence of the Fourier series by instead considering the symmetric extension of the function.

We also discussed the MP3 standard for compression of sound, and its relation to a psychoacoutic model which describes how the human auditory system perceives sound. There exist a wide variety of documents on this standard. In [24], an overview is given, which, although written in a signal processing friendly language and representing most relevant theory such as for the psychoacoutic model, does not dig into all the details.

we also defined analog filters, which were operations which operate on continuous sound, without any assumption on periodicity. In signal processing literature onde defines the *Continuous-time Fourier transform* , or CTFT . We will not use this concept in this book. We have instead disguised this concept as the frequency response of an analog filter. To be more precise: in the literature, the CTFT of g

is nothing but the frequency response of an analog filter with g as convolution kernel.

Chapter 2

Digital sound and Discrete Fourier analysis

In Chapter 1 we saw how a periodic function can be decomposed into a linear combination of sines and cosines, or equivalently, a linear combination of complex exponential functions. This kind of decomposition is, however, not very convenient from a computational point of view. First of all, the coefficients are given by integrals that in most cases cannot be evaluated exactly, so some kind of numerical integration technique needs to be applied. Secondly, functions are defined for all time instances. On computers and various kinds of media players, however, the sound is *digital*, meaning that it is represented by a large number of function values, and not by a function defined for all time instances.

In this chapter our starting point is simply a vector which represents the sound values, rather than a function $f(t)$. We start by seeing how we can make use of this on a computer, either by playing it as a sound, or performing simple operations on it. After this we continue by decomposing vectors in terms of linear combinations of vectors built from complex exponentials. As before it turns out that this is simplest when we assume that the values in the vector repeat periodically. Then a vector of finite dimension can be used to represent all sound values, and a transformation to the frequency domain, where operations which change the sound can easily be made, simply amounts to multiplying the vector by a matrix. This transformation is called the Discrete Fourier transform, and we will see how we can implement this efficiently. It turns out that these algorithms can also be used for computing approximations to the Fourier series, and for sampling a sound in order to create a vector of sound data.

The examples in this chapter and the next chapter can be run from the notebook `applinalgnbchap2.m`.

2.1 Digital sound and simple operations on digital sound

We start by defining what a digital sound is and by establishing some notation and terminology.

Definition 2.1. *Digital sound.*

A digital sound is a sequence $\boldsymbol{x} = \{x_i\}_{i=0}^{N-1}$ 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.,

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

The measurements are often referred to as samples. The time between successive measurements is called the *sampling period* and is usually denoted T_s. The length of the vector is usually assumed to be N, and it is indexed from 0 to $N-1$. 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.*

Note that this indexing convention for vectors is not standard in mathematics, where vector indices start at 1, as they do in Matlab. 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.

Example 2.2. *The CD-format.*

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.

This value 44 100 for the sampling rate is not coincidental, and we will return to this later.

Example 2.3. *Telephony.*

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.

Simple operations and computations with digital sound can be done in any programming environment. Let us take a look at how these. From Definition 2.1, 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.

2.1.1 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. Let us take a closer look at the different steps here. You will need these steps in Exercise 2.3, where you will be asked to implement a function which plays a pure sound with a given frequency on your computer.

First we need to know how we can obtain the samples of a pure tone. The following code does this when we have defined the variables `f` for its frequency, `antsec` for its length in seconds, and `fs` for the sampling rate.

```
t = linspace(0, antsec, fs*antsec);
x = sin(2*pi*f*t);
```

Code will be displayed in this way throughout these notes. We will mostly use the value 44100 for `fs`, to abide to the sampling rate used on CD's. We also need a function to help us listen to the sound samples. We have the two functions

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

These simply play the audio segment encapsulated by the object `playerobj`. `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`. The mysterious `playerobj` object above can be obtained from the sound samples (represented by a vector `x`) and the sampling rate (`fs`) by the function
:

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

This function 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 2.4. *Basic command to handle sound.*
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 sound samples can have different data types. We will always assume that they are of type `double`. The computer 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, `x` can actually be a matrix: Each column in the matrix represents a sound channel . Sounds we generate on our own from a mathematical function (as for the pure tone above) will typically have only one channel, so that `x` has only one column. If `x` originates from a stereo sound file, it will have two columns.

You can create `x` on your own, either by filling it with values from a mathematical function as we did for the pure tone above, or filling in with samples from a sound file. To do this from a file in the `wav`-format named `filename`, simply write

```
[x, fs] = audioread(filename)
```

The `wav`-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 `x`, this function also returns the sampling rate `fs` used in the file. The function

```
audiowrite(filename, x, fs)
```

can similarly be used to write the data stored in the vector `x` to the `wav`-file by the name `filename`. As an example, we can listen to and write the pure tone above with the help of the following code:

```
playerobj = audioplayer(x, fs);
playblocking(playerobj);
audiowrite('puretone440.wav', x, fs);
```

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. In order to do this we can first create the samples of one period of the square wave as follows:

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

Here we have first computed the number of samples in one period. With the following code we can then repeat this period so that the produced sound has the desired length (`fs` copies of one period per second), and then play it:

```
x=repmat(oneperiod,1,antsec*f);
playerobj=audioplayer(x, fs);
playblocking(playerobj);
```

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

```
[x, fs] = audioread('sounds/castanets.wav');
```

reads the file castanets.wav, and stores the sound samples in the matrix `x`. In this case there are two sound channels, so there are two columns in `x`. To listen to the sound from only one channel, we can write

```
playerobj=audioplayer(x(:, 2), fs);
playblocking(playerobj);
```

In the following we will usually not to do this, as it is possible to apply operations to all channels simultaneously using the same simple syntax. `audioread` returns sound samples with floating point precision.

It may be that some other environment 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 2.5. *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 neighboring 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.

A digital sound can be played at normal, double and half sampling rate by writing

```
playerobj = audioplayer(x, fs);
playblocking(playerobj);

playerobj = audioplayer(x, 2*fs);
playblocking(playerobj);

playerobj = audioplayer(x, fs/2);
playblocking(playerobj);
```

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 2.6. *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.

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

```
z = x(N:(-1):1, :);
```

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

Example 2.7. *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.

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

```
z = x + c*(2*rand(size(x))-1);
z = z/max(abs(z));
```

Here `rand` is a 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)` above is that random numbers between -1 and 1 are returned instead of random numbers between 0 and 1. Note that we also have scaled the sound samples so that they lie between -1 and 1 (as required by our representation of sound), since the addition may lead to numbers which are outside this range. Without this we may obtain an unrecognizable sound, as values outside the legal range are changed.

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 $\boldsymbol{z}$: `castanets.wav` with noise added with $c = 0.4$ sounds like this, while with $c = 0.1$ it sounds like this.

In addition to the operations listed above, the most important operations on digital sound are *digital filters*. These are given a separate treatment in Chapter 3.

What you should have learned in this section.

- Computer operations for reading, writing, and listening to sound.
- Construct sounds such as pure tones, and the square and triangle waves, from mathematical formulas.
- Comparing a sound with its Fourier series.
- Changing the sample rate, adding noise, or playing a sound backwards.

Exercise 2.1: Sound with increasing loudness

Define the following sound signal

$$f(t) = \begin{cases} 0 & 0 \le t \le 4/440 \\ 2^{\frac{440t-4}{8}} \sin(2\pi 440t) & 4/440 \le t \le 12/440 \\ 2\sin(2\pi 440t) & 12/440 \le t \le 20/440 \end{cases}$$

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 function which generates this sound, and listen to it.

Exercise 2.2: Sum of two pure tones

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.

Exercise 2.3: Playing general pure tones.

Let us write some code so that we can experiment with different pure sounds

a) Write a function `play_pure_sound(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 `play_pure_sound` 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 `play_pure_sound`? How low frequencies are you able to hear?

Exercise 2.4: Playing the square- and triangle waves

Write functions `play_square` and `play_triangle` which take T as input, and which play the square wave of Example 1.10 and the triangle wave of Example 1.11, respectively. In your code, let the samples of the waves be taken at a frequency of 44100 samples per second. Verify that you generate the same sounds as you played in these examples when you set $T = \frac{1}{440}$.

Exercise 2.5: Playing Fourier series of the square- and triangle waves

Let us write programs so that we can listen to the Fourier approximations of the square wave and the triangle wave.

a) Write functions `play_square_fourier` and `play_triangle_fourier` which take T and N as input, and which play the order N Fourier approximation of the square wave and the triangle wave, respectively, for three seconds. Verify that you can generate the sounds you played in examples 1.17 and 1.18.

b) For these Fourier approximations, how high must you choose N for them to be indistuingishable from the square/triangle waves themselves? Also describe how the characteristics of the sound changes when n increases.

Exercise 2.6: Playing with different sample rates

Write a function `play_with_different_fs` which takes the sound samples `x` and a sampling rate `fs` as input, and plays the sound samples 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). When applied to the sample audio file, are the sounds the same as those you heard in Example 2.5?

Exercise 2.7: Playing the reverse sound

Let us also experiment with reversing the samples in a sound file.

a) Write a function `play_reverse` which takes sound data and a sample rate as input, and plays the sound samples backwards. When you run the code on our sample audio file, is the sound the same as the one you heard in Example 2.6?

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

Exercise 2.8: Play sound with added noise

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

a) Write a function `play_with_noise` which takes sound data, sampling rate, and the damping constant c as input, and plays the sound samples with noise added 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 2.7, 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?

2.2 Discrete Fourier analysis and the discrete Fourier transform

In this section we will parallel the developments we did for Fourier series, assuming instead that vectors (rather than functions) are involved. As with Fourier series we will assume that the vector is periodic. This means that we can represent it with the values from only the first period. In the following we will only work with these values, but we will remind ourselves from time to time that the values actually come from a periodic vector. As for functions, we will call denote the periodic vector as the periodic extension of the finite vector. To illustrate this, we have in Figure 2.1 shown a vector $\boldsymbol{x}$ and its periodic extension $\boldsymbol{x}$.

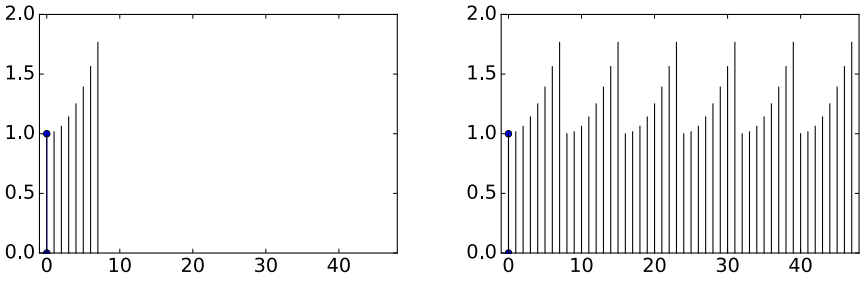

Figure 2.1: A vector and its periodic extension.

At the outset our vectors will have real components, but since we use complex exponentials we must be able to work with complex vectors also. We therefore first need to define the standard inner product and norm for complex vectors.

Definition 2.8. *Euclidean inner product.*

For complex vectors of length N the Euclidean inner product is given by

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{k=0}^{N-1} x_k \overline{y_k}. \tag{2.1}$$

The associated norm is

$$\|\boldsymbol{x}\| = \sqrt{\sum_{k=0}^{N-1} |x_k|^2}. \tag{2.2}$$

In the previous chapter we saw that, using a Fourier series, a function with period T could be approximated by linear combinations of the functions (the pure tones) $\{e^{2\pi int/T}\}_{n=0}^{N}$. This can be generalized to vectors (digital sounds), but then the pure tones must of course also be vectors.

Definition 2.9. *Discrete Fourier analysis.*

In Discrete Fourier analysis, a vector $\boldsymbol{x} = (x_0, \ldots, x_{N-1})$ is represented as a linear combination of the N vectors

$$\boldsymbol{\phi}_n = \frac{1}{\sqrt{N}} \left(1, e^{2\pi in/N}, e^{2\pi i 2n/N}, \ldots, e^{2\pi ikn/N}, \ldots, e^{2\pi in(N-1)/N}\right).$$

These vectors are called the normalised complex exponentials, or the pure digital tones of order N. n is also called frequency index. The whole collection $\mathcal{F}_N = \{\boldsymbol{\phi}_n\}_{n=0}^{N-1}$ is called the N-point Fourier basis.

Note that pure digital tones can be considered as samples of a pure tone, taken uniformly over one period: If $f(t) = e^{2\pi int/T}/\sqrt{N}$ is the pure tone with frequency n/T, then $f(kT/N) = e^{2\pi in(kT/N)/T}/\sqrt{N} = e^{2\pi ink/N}/\sqrt{N} = \boldsymbol{\phi}_n$. When mapping a pure tone to a digital pure tone, the index n corresponds to

frequency $\nu = n/T$, and N the number of samples takes over one period. Since $Tf_s = N$, where f_s is the sampling frequency, we have the following connection between frequency and frequency index:

$$\nu = \frac{nf_s}{N} \text{ and } n = \frac{\nu N}{f_s} \tag{2.3}$$

The following lemma shows that the vectors in the Fourier basis are orthonormal, so they do indeed form a basis.

Lemma 2.10. *Complex exponentials are an orthonormal basis.*
The normalized complex exponentials $\{\boldsymbol{\phi_n}\}_{n=0}^{N-1}$ of order N form an orthonormal basis in $\mathbb{R}^N$.

Proof. Let n_1 and n_2 be two distinct integers in the range $[0, N-1]$. The inner product of $\boldsymbol{\phi}_{n_1}$ and $\boldsymbol{\phi}_{n_2}$ is then given by

$$\begin{aligned}
\langle \boldsymbol{\phi}_{n_1}, \boldsymbol{\phi}_{n_2} \rangle &= \frac{1}{N} \langle e^{2\pi i n_1 k/N}, e^{2\pi i n_2 k/N} \rangle \\
&= \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i n_1 k/N} e^{-2\pi i n_2 k/N} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i (n_1 - n_2) k/N} \\
&= \frac{1}{N} \frac{1 - e^{2\pi i (n_1 - n_2)}}{1 - e^{2\pi i (n_1 - n_2)/N}} \\
&= 0.
\end{aligned}$$

In particular, this orthogonality means that the the complex exponentials form a basis. Clearly also $\langle \boldsymbol{\phi}_n, \boldsymbol{\phi}_n \rangle = 1$, so that the N-point Fourier basis is in fact an orthonormal basis. □

Note that the normalizing factor $\frac{1}{\sqrt{N}}$ was not present for pure tones in the previous chapter. Also, the normalizing factor $\frac{1}{T}$ from the last chapter is not part of the definition of the inner product in this chapter. These are small differences which have to do with slightly different notation for functions and vectors, and which will not cause confusion in what follows.

The focus in Discrete Fourier analysis is to change coordinates from the standard basis to the Fourier basis, performing some operations on this "Fourier representation", and then change coordinates back to the standard basis. Such operations are of crucial importance, and in this section we study some of their basic properties. We start with the following definition.

Definition 2.11. *Discrete Fourier Transform.*

We will denote the change of coordinates matrix from the standard basis of $\mathbb{R}^N$ to the Fourier basis $\mathcal{F}_N$ by F_N. We will also call this the (N-point) *Fourier matrix*.

The matrix $\sqrt{N}F_N$ is also called the (N-point) *discrete Fourier transform* , or DFT. If $\boldsymbol{x}$ is a vector in R^N, then $\boldsymbol{y} = \text{DFT}\boldsymbol{x}$ are called the DFT coefficients of $\boldsymbol{x}$. (the DFT coefficients are thus the coordinates in $\mathcal{F}_N$, scaled with $\sqrt{N}$). $\text{DFT}\boldsymbol{x}$ is sometimes written as $\hat{\boldsymbol{x}}$.

Note that we define the Fourier matrix and the DFT as two different matrices, the one being a scaled version of the other. The reason for this is that there are different traditions in different fields. In pure mathematics, the Fourier matrix is mostly used since it is, as we wil see, a unitary matrix. In signal processing, the scaled version provided by the DFT is mostly used. We will normally write $\boldsymbol{x}$ for the given vector in $\mathbb{R}^N$, and $\boldsymbol{y}$ for its DFT. In applied fields, the Fourier basis vectors are also called *synthesis vectors* , since they can be used used to "synthesize" the vector $\boldsymbol{x}$, with weights provided by the coordinates in the Fourier basis. To be more precise, we have that the change of coordinates performed by the Fourier matrix can be written as

$$\boldsymbol{x} = y_0\boldsymbol{\phi}_0 + y_1\boldsymbol{\phi}_1 + \cdots + y_{N-1}\boldsymbol{\phi}_{N-1} = \begin{pmatrix}\boldsymbol{\phi}_0 & \boldsymbol{\phi}_1 & \cdots & \boldsymbol{\phi}_{N-1}\end{pmatrix}\boldsymbol{y} = F_N^{-1}\boldsymbol{y}, \quad (2.4)$$

where we have used the inverse of the defining relation $\boldsymbol{y} = F_N\boldsymbol{x}$, and that the $\boldsymbol{\phi}_n$ are the columns in F_N^{-1} (this follows from the fact that F_N^{-1} is the change of coordinates matrix from the Fourier basis to the standard basis, and the Fourier basis vectors are clearly the columns in this matrix). Equation (2.4) is also called the synthesis equation.

Example 2.12. *DFT of a cosine.*

Let $\boldsymbol{x}$ be the vector of length N defined by $x_k = \cos(2\pi 5k/N)$, and $\boldsymbol{y}$ the vector of length N defined by $y_k = \sin(2\pi 7k/N)$. Let us see how we can compute $F_N\left(2\boldsymbol{x} + 3\boldsymbol{y}\right)$. By the definition of the Fourier matrix as a change of coordinates, $F_N(\phi_n) = \boldsymbol{e}_n$. We therefore get

$$\begin{aligned}
F_N\left(2\boldsymbol{x} + 3\boldsymbol{y}\right) &= F_N(2\cos(2\pi 5\cdot/N) + 3\sin(2\pi 7\cdot/N)) \\
&= F_N(2\frac{1}{2}(e^{2\pi i5\cdot/N} + e^{-2\pi i5\cdot/N}) + 3\frac{1}{2i}(e^{2\pi i7\cdot/N} - e^{-2\pi i7\cdot/N})) \\
&= F_N(\sqrt{N}\phi_5 + \sqrt{N}\phi_{N-5} - \frac{3i}{2}\sqrt{N}(\phi_7 - \phi_{N-7})) \\
&= \sqrt{N}(F_N(\phi_5) + F_N(\phi_{N-5}) - \frac{3i}{2}F_N\phi_7 + \frac{3i}{2}F_N\phi_{N-7}) \\
&= \sqrt{N}\boldsymbol{e}_5 + \sqrt{N}\boldsymbol{e}_{N-5} - \frac{3i}{2}\sqrt{N}\boldsymbol{e}_7 + \frac{3i}{2}\sqrt{N}\boldsymbol{e}_{N-7}.
\end{aligned}$$

Let us find an expression for the matrix F_N. From Lemma 2.10 we know that the columns of F_N^{-1} are orthonormal. If the matrix was real, it would have been

called orthogonal, and the inverse matrix could have been obtained by transposing. F_N^{-1} is complex, however, and it is easy to see that the conjugation present in the definition of the inner product (2.1), implies that the inverse of F_N can be obtained if we also conjugate, in addition to transpose, i.e. $(F_N)^{-1} = (\overline{F_N})^T$. We call $(\overline{A})^T$ the *conjugate transpose* of A, and denote this by A^H. We thus have that $(F_N)^{-1} = (F_N)^H$. Matrices which satisfy $A = A^H$ are called *unitary* . For complex matrices, this is the parallel to orthogonal matrices.

Theorem 2.13. *Fourier matrix is unitary.*

The Fourier matrix F_N is the unitary $N \times N$-matrix with entries given by

$$(F_N)_{nk} = \frac{1}{\sqrt{N}} e^{-2\pi ink/N},$$

for $0 \leq n, k \leq N-1$.

Since the Fourier matrix is easily inverted, the DFT is also easily inverted. Note that, since $(F_N)^T = F_N$, we have that $(F_N)^{-1} = \overline{F_N}$. Let us make the following definition.

Definition 2.14. *IDFT.*

The matrix $\overline{F_N}/\sqrt{N}$ is the inverse of the matrix DFT $= \sqrt{N}F_N$. We call this inverse matrix the *inverse discrete Fourier transform*, or IDFT.

We can thus also view the IDFT as a change of coordinates (this time from the Fourier basis to the standard basis), with a scaling of the coordinates by $1/\sqrt{N}$ at the end. The IDFT is often called the *reverse DFT*. Similarly, the DFT is often called the *forward DFT*.

That $\boldsymbol{y} = \text{DFT}\boldsymbol{x}$ and $\boldsymbol{x} = \text{IDFT}\boldsymbol{y}$ can also be expressed in component form as

$$y_n = \sum_{k=0}^{N-1} x_k e^{-2\pi ink/N} \qquad\qquad x_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n e^{2\pi ink/N} \tag{2.5}$$

In applied fields such as signal processing, it is more common to state the DFT and IDFT in these component forms, rather than in the matrix forms $\boldsymbol{y} = \text{DFT}\boldsymbol{y}$ and $\boldsymbol{x} = \text{IDFT}\boldsymbol{y}$.

Let us now see how these formulas work out in practice by considering some examples.

Example 2.15. *DFT on a square wave.*

Let us attempt to apply the DFT to a signal $\boldsymbol{x}$ which is 1 on indices close to 0, and 0 elsewhere. Assume that

$$x_{-L} = \ldots = x_{-1} = x_0 = x_1 = \ldots = x_L = 1,$$

while all other values are 0. This is similar to a square wave, with some modifications: First of all we assume symmetry around 0, while the square wave

of Example 1.10 assumes antisymmetry around 0. Secondly the values of the square wave are now 0 and 1, contrary to -1 and 1 before. Finally, we have a different proportion of where the two values are assumed. Nevertheless, we will also refer to the current digital sound as a square wave.

Since indices with the DFT are between 0 an $N-1$, and since $\boldsymbol{x}$ is assumed to have period N, the indices $[-L, L]$ where our signal is 1 translates to the indices $[0, L]$ and $[N-L, N-1]$ (i.e., it is 1 on the first and last parts of the vector). Elsewhere our signal is zero. Since $\sum_{k=N-L}^{N-1} e^{-2\pi ink/N} = \sum_{k=-L}^{-1} e^{-2\pi ink/N}$ (since $e^{-2\pi ink/N}$ is periodic with period N), the DFT of $\boldsymbol{x}$ is

$$\begin{aligned} y_n &= \sum_{k=0}^{L} e^{-2\pi ink/N} + \sum_{k=N-L}^{N-1} e^{-2\pi ink/N} = \sum_{k=0}^{L} e^{-2\pi ink/N} + \sum_{k=-L}^{-1} e^{-2\pi ink/N} \\ &= \sum_{k=-L}^{L} e^{-2\pi ink/N} = e^{2\pi inL/N} \frac{1 - e^{-2\pi in(2L+1)/N}}{1 - e^{-2\pi in/N}} \\ &= e^{2\pi inL/N} e^{-\pi in(2L+1)/N} e^{\pi in/N} \frac{e^{\pi in(2L+1)/N} - e^{-\pi in(2L+1)/N}}{e^{\pi in/N} - e^{-\pi in/N}} \\ &= \frac{\sin(\pi n(2L+1)/N)}{\sin(\pi n/N)}. \end{aligned}$$

This computation does in fact also give us the IDFT of the same vector, since the IDFT just requires a change of sign in all the exponents, in addition to the $1/N$ normalizing factor. From this example we see that, in order to represent $\boldsymbol{x}$ in terms of frequency components, all components are actually needed. The situation would have been easier if only a few frequencies were needed.

Example 2.16. *Computing the DFT by hand.*

In most cases it is difficult to compute a DFT by hand, due to the entries $e^{-2\pi ink/N}$ in the matrices, which typically can not be represented exactly. The DFT is therefore usually calculated on a computer only. However, in the case $N = 4$ the calculations are quite simple. In this case the Fourier matrix takes the form

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

We now can compute the DFT of a vector like $(1, 2, 3, 4)^T$ simply as

$$\mathrm{DFT}_4 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1+2+3+4 \\ 1-2i-3+4i \\ 1-2+3-4 \\ 1+2i-3-4i \end{pmatrix} = \begin{pmatrix} 10 \\ -2+2i \\ -2 \\ -2-2i \end{pmatrix}.$$

In general, computing the DFT implies using floating point multiplication. For $N = 4$, however, we see that there is no need for floating point multiplication at all, since DFT_4 has unit entries which are either real or purely imaginary.

Example 2.17. *Direct implementation of the DFT.*
The DFT can be implemented very simply and directly by the code

```
function y = DFTImpl(x)
    N = size(x, 1);
    y = zeros(size(x));
    for n = 1:N
        D = exp(*2*pi*1i*(n-1)*(0:(N-1))/N);
        y(n) = dot(D, x);
    end
```

n has been replaced by $n-1$ in this code since n runs from 1 to N (array indices must start at 1 in Matlab). In exercise 2.16 we will extend this to a general implementation we will use later. Note that we do not allocate the entire matrix F_N in this code, as this quickly leads to out of memory situations, even for N of moderate size. Instead we construct one row of F_N at a time, and use use this to compute one entry in the output. The method `dot` can be used here, since each entry in matrix multiplication can be viewed as an inner product. It is likely that the `dot` function is more efficient than using a `for`-loop, since Matlab may have an optimized way for computing this. Note that `dot` in Matlab conjugates the first components, contrary to what we do in our definition of a complex inner product. This is why we have dropped a sign in the exponent here. This can be rewritten to a direct implementation of the IDFT also. We will look at this in the exercises, where we also make the method more general, so that the DFT can be applied to a series of vectors at a time (it can then be applied to all the channels in a sound in one call). Multiplying a full $N \times N$ matrix by a vector requires roughly N^2 arithmetic operations. The DFT algorithm above will therefore take a long time when N becomes moderately large. It turns out that a much more efficient algorithm exists for computing the DFT, which we will study at the end of this chapter. Matlab also has a built-in implementation of the DFT which uses such an efficient algorithm.

The DFT has properties which are very similar to those of Fourier series, as they were listed in Theorem 1.28. The following theorem sums this up:

Theorem 2.18. *Properties of the DFT.*
Let $\boldsymbol{x}$ be a real vector of length N. The DFT has the following properties:

1. $(\widehat{\boldsymbol{x}})_{N-n} = \overline{(\widehat{\boldsymbol{x}})_n}$ for $0 \leq n \leq N-1$.
2. If $x_k = x_{N-k}$ for all n (so $\boldsymbol{x}$ is symmetric), then $\widehat{\boldsymbol{x}}$ is a real vector.
3. If $x_k = -x_{N-k}$ for all k (so $\boldsymbol{x}$ is antisymmetric), then $\widehat{\boldsymbol{x}}$ is a purely imaginary vector.

4. If d is an integer and $\boldsymbol{z}$ is the vector with components $z_k = x_{k-d}$ (the vector $\boldsymbol{x}$ with its elements delayed by d), then $(\widehat{\boldsymbol{z}})_n = e^{-2\pi idn/N} (\widehat{\boldsymbol{x}})_n$.
5. If d is an integer and $\boldsymbol{z}$ is the vector with components $z_k = e^{2\pi idk/N} x_k$, then $(\widehat{\boldsymbol{z}})_n = (\widehat{\boldsymbol{x}})_{n-d}$.

Proof. The methods used in the proof are very similar to those used in the proof of Theorem 1.28. From the definition of the DFT we have

$$(\widehat{\boldsymbol{x}})_{N-n} = \sum_{k=0}^{N-1} e^{-2\pi ik(N-n)/N} x_k = \sum_{k=0}^{N-1} e^{2\pi ikn/N} x_k = \overline{\sum_{k=0}^{N-1} e^{-2\pi ikn/N} x_k} = \overline{(\widehat{\boldsymbol{x}})_n}$$

which proves property 1.

To prove property 2, we write

$$\begin{aligned}(\widehat{\boldsymbol{z}})_n &= \sum_{k=0}^{N-1} z_k e^{-2\pi ikn/N} = \sum_{k=0}^{N-1} x_{N-k} e^{-2\pi ikn/N} = \sum_{u=1}^{N} x_u e^{-2\pi i(N-u)n/N} \\ &= \sum_{u=0}^{N-1} x_u e^{2\pi iun/N} = \overline{\sum_{u=0}^{N-1} x_u e^{-2\pi iun/N}} = \overline{(\widehat{\boldsymbol{x}})_n}.\end{aligned}$$

If $\boldsymbol{x}$ is symmetric it follows that $\boldsymbol{z} = \boldsymbol{x}$, so that $(\widehat{\boldsymbol{x}})_n = \overline{(\widehat{\boldsymbol{x}})_n}$. Therefore $\boldsymbol{x}$ must be real. The case of antisymmetry in property 3 follows similarly.

To prove property 4 we observe that

$$\begin{aligned}(\widehat{\boldsymbol{z}})_n &= \sum_{k=0}^{N-1} x_{k-d} e^{-2\pi ikn/N} = \sum_{k=0}^{N-1} x_k e^{-2\pi i(k+d)n/N} \\ &= e^{-2\pi idn/N} \sum_{k=0}^{N-1} x_k e^{-2\pi ikn/N} = e^{-2\pi idn/N} (\widehat{\boldsymbol{x}})_n .\end{aligned}$$

For the proof of property 5 we note that the DFT of $\boldsymbol{z}$ is

$$(\widehat{\boldsymbol{z}})_n = \sum_{k=0}^{N-1} e^{2\pi idk/N} x_n e^{-2\pi ikn/N} = \sum_{k=0}^{N-1} x_n e^{-2\pi i(n-d)k/N} = (\widehat{\boldsymbol{x}})_{n-d} .$$

This completes the proof. □

These properties have similar interpretations as the ones listed in Theorem 1.28 for Fourier series. Property 1 says that we need to store only about one half of the DFT coefficients, since the remaining coefficients can be obtained by conjugation. In particular, when N is even, we only need to store $y_0, y_1, \ldots, y_{N/2}$. This also means that, if we plot the (absolute value) of the DFT of a real vector, we will see a symmetry around the index $n = N/2$. The theorem generalizes the properties from Theorem 1.28, except for the last property where the signal had a point of symmetry. We will delay the generalization of this property to later.

Example 2.19. *Computing the DFT when multiplying with a complex exponential.*

To see how we can use the fourth property of Theorem 2.18, consider a vector $\boldsymbol{x} = (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ with length $N = 8$, and assume that $\boldsymbol{x}$ is so that $F_8(\boldsymbol{x}) = (1, 2, 3, 4, 5, 6, 7, 8)$. Consider the vector $\boldsymbol{z}$ with components $z_k = e^{2\pi i 2k/8} x_k$. Let us compute $F_8(\boldsymbol{z})$. Since multiplication of $\boldsymbol{x}$ with $e^{2\pi ikd/N}$ delays the output $\boldsymbol{y} = F_N(\boldsymbol{x})$ with d elements, setting $d = 2$, the $F_8(\boldsymbol{z})$ can be obtained by delaying $F_8(\boldsymbol{x})$ by two elements, so that $F_8(\boldsymbol{z}) = (7, 8, 1, 2, 3, 4, 5, 6)$. It is straightforward to compute this directly also:

$$\begin{aligned}(F_N \boldsymbol{z})_n &= \sum_{k=0}^{N-1} z_k e^{-2\pi ikn/N} = \sum_{k=0}^{N-1} e^{2\pi i 2k/N} x_k e^{-2\pi ikn/N} \\ &= \sum_{k=0}^{N-1} x_k e^{-2\pi ik(n-2)/N} = (F_N(\boldsymbol{x}))_{n-2}.\end{aligned}$$

What you should have learned in this section.

- The definition of the Fourier basis and its orthonormality.
- The definition of the Discrete Fourier Transfrom as a change of coordinates to the Fourier basis, its inverse, and its unitarity.
- How to apply the DFT to a sum of sinusoids.
- Properties of the DFT, such as conjugate symmetry when the vector is real, how it treats delayed vectors, or vectors multiplied with a complex exponential.

Exercise 2.9: Computing the DFT by hand

Compute $F_4 \boldsymbol{x}$ when $\boldsymbol{x} = (2, 3, 4, 5)$.

Exercise 2.10: Exact form of low-order DFT matrix

As in Example 2.16, state the exact cartesian form of the Fourier matrix for the cases $N = 6$, $N = 8$, and $N = 12$.

Exercise 2.11: DFT of a delayed vector

We have a real vector $\boldsymbol{x}$ with length N, and define the vector $\boldsymbol{z}$ by delaying all elements in $\boldsymbol{x}$ with 5 cyclically, i.e. $z_5 = x_0$, $z_6 = x_1$,...,$z_{N-1} = x_{N-6}$, and $z_0 = x_{N-5}$,...,$z_4 = x_{N-1}$. For a given n, if $|(F_N \boldsymbol{x})_n| = 2$, what is then $|(F_N \boldsymbol{z})_n|$? Justify the answer.

Exercise 2.12: Using symmetry property

Given a real vector $\boldsymbol{x}$ of length 8 where $(F_8(\boldsymbol{x}))_2 = 2 - i$, what is $(F_8(\boldsymbol{x}))_6$?

Exercise 2.13: DFT of $\cos^2(2\pi k/N)$

Let $\boldsymbol{x}$ be the vector of length N where $x_k = \cos^2(2\pi k/N)$. What is then $F_N\boldsymbol{x}$?

Exercise 2.14: DFT of $c^k\boldsymbol{x}$

Let $\boldsymbol{x}$ be the vector with entries $x_k = c^k$. Show that the DFT of $\boldsymbol{x}$ is given by the vector with components

$$y_n = \frac{1 - c^N}{1 - ce^{-2\pi in/N}}$$

for $n = 0, \ldots, N - 1$.

Exercise 2.15: Rewrite a complex DFT as real DFT's

If $\boldsymbol{x}$ is complex, Write the DFT in terms of the DFT on real sequences.

Hint. Split into real and imaginary parts, and use linearity of the DFT.

Exercise 2.16: DFT implementation

Extend the code for the function `DFTImpl` in Example 2.17 so that

- The function also takes a second parameter called `forward`. If this is true the DFT is applied. If it is false, the IDFT is applied. If this parameter is not present, then the forward transform should be assumed.
- If the input `x` is two-dimensional (i.e. a matrix), the DFT/IDFT should be applied to each column of `x`. This ensures that, in the case of sound, the FFT is applied to each channel in the sound when the enrire sound is used as input, as we are used to when applying different operations to sound.

Also, write documentation for the code.

Exercise 2.17: Symmetry

Assume that N is even.

a) Show that, if $x_{k+N/2} = x_k$ for all $0 \leq k < N/2$, then $y_n = 0$ when n is odd.

b) Show that, if $x_{k+N/2} = -x_k$ for all $0 \leq k < N/2$, then $y_n = 0$ when n is even.

c) Show also the converse statements in a. and b..

d) Also show the following:

- $x_n = 0$ for all odd n if and only if $y_{k+N/2} = y_k$ for all $0 \le k < N/2$.
- $x_n = 0$ for all even n if and only if $y_{k+N/2} = -y_k$ for all $0 \le k < N/2$.

Exercise 2.18: DFT on complex and real data

Let $\boldsymbol{x}_1, \boldsymbol{x}_2$ be real vectors, and set $\boldsymbol{x} = \boldsymbol{x}_1 + i\boldsymbol{x}_2$. Use Theorem 2.18 to show that

$$(F_N(\boldsymbol{x}_1))_k = \frac{1}{2}\left((F_N(\boldsymbol{x}))_k + \overline{(F_N(\boldsymbol{x}))_{N-k}}\right)$$
$$(F_N(\boldsymbol{x}_2))_k = \frac{1}{2i}\left((F_N(\boldsymbol{x}))_k - \overline{(F_N(\boldsymbol{x}))_{N-k}}\right)$$

This shows that we can compute two DFT's on real data from one DFT on complex data, and $2N$ extra additions.

2.3 Connection between the DFT and Fourier series. Sampling and the sampling theorem

So far we have focused on the DFT as a tool to rewrite a vector in terms of the Fourier basis vectors. In practice, the given vector $\boldsymbol{x}$ will often be sampled from some real data given by a function $f(t)$. We may then compare the frequency content of $\boldsymbol{x}$ and f, and ask how they are related: What is the relationship between the Fourier coefficients of f and the DFT-coefficients of $\boldsymbol{x}$?

In order to study this, assume for simplicity that $f \in V_{M,T}$ for some M. This means that f equals its Fourier approximation f_M,

$$f(t) = f_M(t) = \sum_{n=-M}^{M} z_n e^{2\pi i n t/T}, \text{ where } z_n = \frac{1}{T}\int_0^T f(t)e^{-2\pi i n t/T}\,dt. \quad (2.6)$$

We here have changed our notation for the Fourier coefficients from y_n to z_n, in order not to confuse them with the DFT coefficients. We recall that in order to represent the frequency n/T fully, we need the corresponding exponentials with both positive and negative arguments, i.e., both $e^{2\pi i n t/T}$ and $e^{-2\pi i n t/T}$.

Fact 2.20. *frequency vs. Fourier coefficients.*
Suppose f is given by its Fourier series (2.6). Then the total frequency content for the frequency n/T is given by the two coefficients z_n and z_{-n}.

We have the following connection between the Fourier coefficients of f and the DFT of the samples of f.

Proposition 2.21. *Relation between Fourier coefficients and DFT coefficients.*
Let $N > 2M$, $f \in V_{M,T}$, and let $\boldsymbol{x} = \{f(kT/N)\}_{k=0}^{N-1}$ be N uniform samples from f over $[0, T]$. The Fourier coefficients z_n of f can be computed from

$$(z_0, z_1, \ldots, z_M, \underbrace{0, \ldots, 0}_{N-(2M+1)}, z_{-M}, z_{-M+1}, \ldots, z_{-1}) = \frac{1}{N}\mathrm{DFT}_N \boldsymbol{x}. \tag{2.7}$$

In particular, the total contribution in f from frequency n/T, for $0 \leq n \leq M$, is given by y_n and y_{N-n}, where $\boldsymbol{y}$ is the DFT of $\boldsymbol{x}$.

Proof. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be as defined, so that

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} y_n e^{2\pi i n k/N}. \tag{2.8}$$

Inserting the sample points $t = kT/N$ into the Fourier series, we must have that

$$\begin{aligned} x_k = f(kT/N) &= \sum_{n=-M}^{M} z_n e^{2\pi i n k/N} = \sum_{n=-M}^{-1} z_n e^{2\pi i n k/N} + \sum_{n=0}^{M} z_n e^{2\pi i n k/N} \\ &= \sum_{n=N-M}^{N-1} z_{n-N} e^{2\pi i (n-N) k/N} + \sum_{n=0}^{M} z_n e^{2\pi i n k/N} \\ &= \sum_{n=0}^{M} z_n e^{2\pi i n k/N} + \sum_{n=N-M}^{N-1} z_{n-N} e^{2\pi i n k/N}. \end{aligned}$$

This states that $\boldsymbol{x} = N\mathrm{IDFT}_N(z_0, z_1, \ldots, z_M, \underbrace{0, \ldots, 0}_{N-(2M+1)}, z_{-M}, z_{-M+1}, \ldots, z_{-1})$.
Equation (2.7) follows by applying the DFT to both sides. We also see that $z_n = y_n/N$ and $z_{-n} = y_{2M+1-n}/N = y_{N-n}/N$, when $\boldsymbol{y}$ is the DFT of $\boldsymbol{x}$. It now also follows immediately that the frequency content in f for the frequency n/T is given by y_n and y_{N-n}. This completes the proof. □

In proposition 2.21 we take N samples over $[0, T]$, i.e. we sample at rate $f_s = N/T$ samples per second. When $|n| \leq M$, a pure sound with frequency $\nu = n/T$ is then seen to correspond to the DFT indices n and $N - n$. Since $T = N/f_s$, $\nu = n/T$ can also be written as $\nu = nf_s/N$. Moreover, the highest frequencies in proposition 2.21 are those close to $\nu = M/T$, which correspond to DFT indices close to $N - M$ and M, which are the nonzero frequencies closest to $N/2$. DFT index $N/2$ corresponds to the frequency $N/(2T) = f_s/2$, which corresponds to the highest frequency we can reconstruct from samples for any M. Similarly, the lowest frequencies are those close to $\nu = 0$, which correspond to DFT indices close to 0 and N. Let us summarize this as follows.

Observation 2.22. *Connection between DFT index and frequency.*

Assume that $\boldsymbol{x}$ are N samples of a sound taken at sampling rate f_s samples per second, and let $\boldsymbol{y}$ be the DFT of $\boldsymbol{x}$. Then the DFT indices n and $N-n$ give the frequency contribution at frequency $\nu = nf_s/N$. Moreover, the low frequencies in $\boldsymbol{x}$ correspond to the y_n with n near 0 and N, while the high frequencies in $\boldsymbol{x}$ correspond to the y_n with n near $N/2$.

The theorem says that any $f \in V_{M,T}$ can be reconstructed from its samples (since we can write down its Fourier series), as long as $N > 2M$. That $f \in V_{M,T}$ is important. From Figure 2.2 it is clear that information is lost in the right plot when we discard everything but the sample values from the left plot.

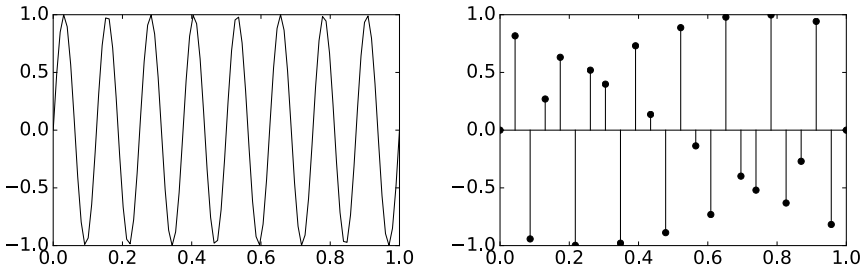

Figure 2.2: An example on how the samples are picked from an underlying continuous time function (left), and the samples on their own (right).

Here the function is $f(t) = \sin(2\pi 8t) \in V_{8,1}$, so that we need to choose N so that $N > 2M = 16$ samples. Here $N = 23$ samples were taken, so that reconstruction from the samples is possible. That the condition $N < 2M$ is also necessary can easily be observed in Figure 2.3.

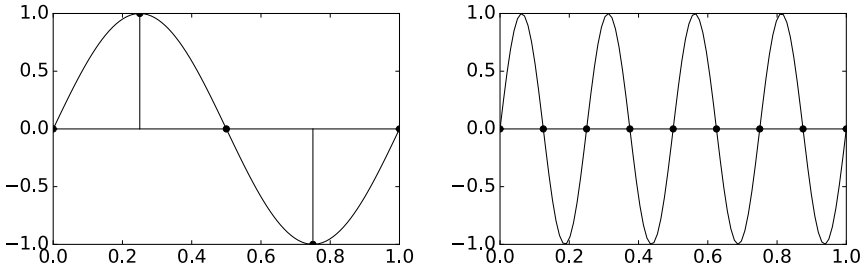

Figure 2.3: Sampling $\sin(2\pi t)$ with two points (left), and sampling $\sin(2\pi 4t)$ with eight points (right).

Right we have plotted $\sin(2\pi 4t) \in V_{4,1}$, with $N = 8$ sample points taken uniformly from $[0, 1]$. Here $M = 4$, so that we require $2M + 1 = 9$ sample points, according to proposition 2.21. Clearly there is an infinite number of possible functions in $V_{M,T}$ passing through the sample points (which are all zero): Any

$f(t) = c\sin(2\pi 4t)$ will do. Left we consider one period of $\sin(2\pi t)$. Since this is in $V_{M,T} = V_{1,1}$, reconstruction should be possible if we have $N \geq 2M + 1 = 3$ samples. Four sample points, as seen left, is thus be enough to secure reconstruct.

The special case $N = 2M + 1$ is interesting. No zeros are then inserted in the vector in Equation (2.7). Since the DFT is one-to-one, this means that there is a one-to-one correspondence between sample values and functions in $V_{M,T}$ (i.e. Fourier series), i.e. we can always find a unique interpolant in $V_{M,T}$ from $N = 2M + 1$ samples. In Exercise 2.21 you will asked to write code where you start with a given function f, Take $N = 2M+1$ samples, and plot the interpolant from $V_{M,T}$ against f. Increasing M should give an interpolant which is a better approximation to f, and if f itself resides in some $V_{M,T}$ for some M, we should obtain equality when we choose M big enough. We have in elementary calculus courses seen how to determine a polynomial of degree $N - 1$ that interpolates a set of N data points, and such polynomials are called interpolating polynomials. In mathematics many other classes than polynomials exist which are also useful for interpolation, and the Fourier basis is just one example.

Besides reconstructing a function from its samples, proposition 2.21 also enables us to approximate functions in a simple way. To elaborate on this, recall that the Fourier series approximation f_M is a best approximation to f from $V_{M,T}$. We usually can't compute f_M exactly, however, since this requires us to compute the Fourier integrals. We could instead form the samples $\boldsymbol{x}$ of f, and apply proposition 2.21. If M is high, f_M is a good approximation to f, so that the samples of f_M are a good approximation to $\boldsymbol{x}$. By continuity of the DFT, it follows that $\boldsymbol{y} = \mathrm{DFT}_N\boldsymbol{x}$ is a good approximation to the DFT of the samples of f_M, so that

$$\tilde{f}(t) = \sum_{n=0}^{N-1} y_n e^{2\pi i n t/T} \tag{2.9}$$

is a good approximation to f_M, and therefore also to f. We have illustrated this in Figure 2.4.

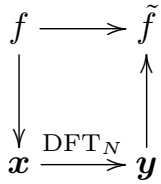

Figure 2.4: How we can interpolate f from $V_{M,T}$ with help of the DFT. The left vertical arrow represents sampling. The right vertical arrow represents interpolation, i.e. computing Equation (2.9).

The new function $\tilde{f}$ has the same values as f in the sample points. This is usually not the case for f_M, so that $\tilde{f}$ and f_M are different approximations to f. Let us summarize as follows.

Idea 2.23. *$\tilde{f}$ as approximation to f.*

The function $\tilde{f}$ resulting from sampling, taking the DFT, and interpolation, as shown in Figure 2.4, also gives an approximation to f. $\tilde{f}$ is a worse approximation in the mean square sense (since f_M is the best such), but it is much more useful since it avoids evaluation of the Fourier integrals, depends only on the samples, and is easily computed.

The condition $N > 2M$ in proposition 2.21 can also be written as $N/T > 2M/T$. The left side is now the sampling rate f_s, while the right side is the double of the highest frequency in f. The result can therefore also be restated as follows

Proposition 2.24. *Reconstruction from samples.*

Any $f \in V_{M,T}$ can be reconstructed uniquely from a uniform set of samples $\{f(kT/N)\}_{k=0}^{N-1}$, as long as $f_s > 2|\nu|$, where ν denotes the highest frequency in f.

We also refer to $f_s = 2|\nu|$ as the critical sampling rate, since it is the minimum sampling rate we need in order to reconstruct f from its samples. If f_s is substantially larger than $2|\nu|$ we say that f is oversampled, since we have takes more samples than we really need. Similarly we say that f is undersampled if f_s is smaller than $2|\nu|$, since we have not taken enough samples in order to reconstruct f. Clearly proposition 2.21 gives one formula for the reconstruction. In the literature another formula can be found, which we now will deduce. This alternative version of Theorem 2.21 is also called *the sampling theorem*. We start by substituting $N = T/T_s$ (i.e. $T = NT_s$, with T_s being the sampling period) in the Fourier series for f:

$$f(kT_s) = \sum_{n=-M}^{M} z_n e^{2\pi i n k/N} \qquad -M \le k \le M.$$

Equation (2.7) said that the Fourier coefficients could be found from the samples from

$$(z_0, z_1, \ldots, z_M, \underbrace{0, \ldots, 0}_{N-(2M+1)}, z_{-M}, z_{-M+1}, \ldots, z_{-1}) = \frac{1}{N}\mathrm{DFT}_N \boldsymbol{x}.$$

By delaying the n index with $-M$, this can also be written as

$$z_n = \frac{1}{N}\sum_{k=0}^{N-1} f(kT_s)e^{-2\pi i n k/N} = \frac{1}{N}\sum_{k=-M}^{M} f(kT_s)e^{-2\pi i n k/N}, \quad -M \le n \le M.$$

Inserting this in the reconstruction formula we get

$$\begin{aligned} f(t) &= \frac{1}{N}\sum_{n=-M}^{M}\sum_{k=-M}^{M} f(kT_s)e^{-2\pi ink/N}e^{2\pi int/T} \\ &= \sum_{k=-M}^{M}\frac{1}{N}\left(\sum_{n=-M}^{M} f(kT_s)e^{2\pi in(t/T-k/N)}\right) \\ &= \sum_{k=-M}^{M}\frac{1}{N}e^{-2\pi iM(t/T-k/N)}\frac{1-e^{2\pi i(2M+1)(t/T-k/N)}}{1-e^{2\pi i(t/T-k/N)}}f(kT_s) \\ &= \sum_{k=-M}^{M}\frac{1}{N}\frac{\sin(\pi(t-kT_s)/T_s)}{\sin(\pi(t-kT_s)/T)}f(kT_s) \end{aligned}$$

Let us summarize our findings as follows:

Theorem 2.25. *Sampling theorem and the ideal interpolation formula for periodic functions.*

Let f be a periodic function with period T, and assume that f has no frequencies higher than νHz. Then f can be reconstructed exactly from its samples $f(-MT_s),\ldots,f(MT_s)$ (where T_s is the sampling period, $N=\frac{T}{T_s}$ is the number of samples per period, and $M=2N+1$) when the sampling rate $f_s=\frac{1}{T_s}$ is bigger than 2ν. Moreover, the reconstruction can be performed through the formula

$$f(t)=\sum_{k=-M}^{M} f(kT_s)\frac{1}{N}\frac{\sin(\pi(t-kT_s)/T_s)}{\sin(\pi(t-kT_s)/T)}. \tag{2.10}$$

Formula (2.10) is also called the ideal interpolation formula for periodic functions . Such formulas, where one reconstructs a function based on a weighted sum of the sample values, are more generally called *interpolation formulas* . The function $\frac{1}{N}\frac{\sin(\pi(t-kT_s)/T_s)}{\sin(\pi(t-kT_s)/T)}$ is also called an interpolation kernel. Note that f itself may not be equal to a finite Fourier series, and reconstruction is in general not possible then. The ideal interpolation formula can in such cases still be used, but the result we obtain may be different from $f(t)$.

In fact, the following more general result holds, which we will not prove. The result is also valid for functions which are not periodic, and is frequently stated in the literature:

Theorem 2.26. *Sampling theorem and the ideal interpolation formula, general version..*

Assume that f has no frequencies higher than νHz. Then f can be reconstructed exactly from its samples $\ldots,f(-2T_s),f(-T_s),f(0),f(T_s),f(2T_s),\ldots$ when the sampling rate is bigger than 2ν. Moreover, the reconstruction can be performed through the formula

$$f(t) = \sum_{k=-\infty}^{\infty} f(kT_s) \frac{\sin(\pi(t - kT_s)/T_s)}{\pi(t - kT_s)/T_s}. \tag{2.11}$$

When f is periodic, it is possible to deduce this partly from the interpolation formula for periodic functions. An ingredient in this is that $x \approx \sin x$ for small x, so that there certainly is a connection between the terms in the two sums. The non-periodicity requires more tools in Fourier analysis, however.

The DFT coefficients represent the contribution in a sound at given frequencies. Due to this the DFT is extremely useful for performing operations on sound, and also for compression as we will see. For instance we can listen to either the lower or higher frequencies after performing a simple adjustment of the DFT coefficients. Observation 2.22 says that the $2L + 1$ lowest frequencies correspond to the DFT-indices $[0, L] \cup [N - L, N - 1]$, while the $2L + 1$ highest frequencies correspond to DFT-indices $[N/2 - L, N/2 + L]$ (if we assume that N is even). In Matlab we need to add 1 to these indices to obtain the Matlab indices into the vectors. If we perform a DFT, eliminate these low or high frequencies, and perform an inverse DFT, we recover the sound signal where these frequencies have been eliminated. With the help of the DFT implementation from Example 2.17, all this can be achieved for zeroing out the highest frequencies with the following code:

```
L = 10000;
N = size(x, 1);

y = fft(x);
y((L+2):(N-L), :) = 0;
newx = ifft(y);
playerobj=audioplayer(newx, fs);
playblocking(playerobj);
```

Example 2.27. *Using the DFT to adjust frequencies in sound.*

Let us test the above code on the sound samples in `castanets.wav`. As a first attempt, let us split the sound samples into small blocks of size $N = 32$, and zero out frequencies as described for each block. This should certainly be more efficient than applying the DFT to the entire sound, since it corresponds to applying a sparse block matrix to the entire sound, rather than the full DFT matrix[1]. You will be spared the details for actually splitting the sound file into blocks: you can find the function `playDFT(L, lower)` which performs this splitting, sets frequency components to 0 except the described $2L + 1$ frequency components, and plays the resulting sound. The second parameter `lower` states if the highest or the lowest frequency components should be kept. If you try this for $L = 7$ (i.e. we keep only 15 of the DFT coefficients) for the lower frequencies, the result sounds like this. You can hear the disturbance in the sound, but we have not lost that much even if more than half the DFT coefficients are dropped.

[1] We will shortly see, however, that efficient algorithms for the DFT exist, so that this problem is not so big after all.

If we instead try $L = 3$ the result will sound like this. The quality is much poorer now. However we can still recognize the song, and this suggests that most of the frequency information is contained in the lower frequencies. If we instead use `playDFT` to listen to the higher frequencies, for $L = 7$ the result now sounds like this, and for $L = 3$ the result sounds like this. Both sounds are quite unrecognizable, confirming that most information is contained in the lower frequencies.

Note that there may be a problem in the previous example: when we restrict to the values in a given block, we actually look at a different signal. The new signal repeats the values in the block in periods, while the old signal consists of one much bigger block. What are the differences in the frequency representations of the two signals?

Assume that the entire sound has length M. The frequency representation of this is computed as an M-point DFT (the signal is actually repeated with period M), and we write the sound samples as a sum of frequencies: $x_k = \frac{1}{M}\sum_{n=0}^{M-1} y_n e^{2\pi ikn/M}$. Let us consider the effect of restricting to a block for each of the contributing pure tones $e^{2\pi ikn_0/M}$, $0 \leq n_0 \leq M-1$. When we restrict this to a block of size N, we get the signal $\left\{e^{2\pi ikn_0/M}\right\}_{k=0}^{N-1}$. Depending on n_0, this may not be a Fourier basis vector! Its N-point DFT gives us its frequency representation, and the absolute value of this is

$$\begin{aligned} |y_n| &= \left|\sum_{k=0}^{N-1} e^{2\pi ikn_0/M} e^{-2\pi ikn/N}\right| = \left|\sum_{k=0}^{N-1} e^{2\pi ik(n_0/M - n/N)}\right| \\ &= \left|\frac{1-e^{2\pi iN(n_0/M-n/N)}}{1-e^{2\pi i(n_0/M-n/N)}}\right| = \left|\frac{\sin(\pi N(n_0/M-n/N))}{\sin(\pi(n_0/M-n/N))}\right|. \end{aligned} \tag{2.12}$$

If $n_0 = kM/N$, this gives $y_k = N$, and $y_n = 0$ when $n \neq k$. Thus, splitting the signal into blocks gives another pure tone when n_0 is a multiplum of M/N. When n_0 is different from this the situation is different. Let us set $M = 1000$, $n_0 = 1$, and experiment with different values of N. Figure 2.5 shows the y_n values for different values of N. We see that the frequency representation is now very different, and that many frequencies contribute.

The explanation is that the pure tone is not a pure tone when $N = 64$ and $N = 256$, since at this scale such frequencies are too high to be represented exactly. The closest pure tone in frequency is $n = 0$, and we see that this has the biggest contribution, but other frequencies also contribute. The other frequencies contribute much more when $N = 256$, as can be seen from the peak in the closest frequency $n = 0$. In conclusion, when we split into blocks, the frequency representation may change in an undesirable way. This is a common problem in signal processing theory, that one in practice needs to restrict to smaller segments of samples, but that this restriction may have undesired effects.

Another problem when we restrict to a shorter periodic signal is that we may obtain discontinuities at the boundaries between the new periods, even if there were no discontinuities in the original signal. And, as we know from the

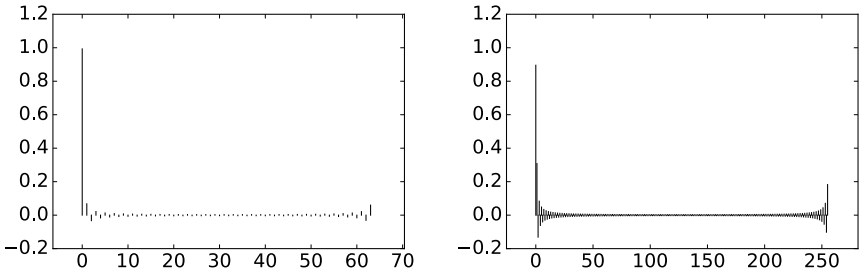

Figure 2.5: The frequency representation obtained when restricting to a block of size N of the signal, for $N = 64$ (left), and $N = 256$ (right)

square wave, discontinuities introduce undesired frequencies. We have already mentioned that symmetric extensions may be used to remedy this.

The MP3 standard also applies a DFT to the sound data . In its simplest form it applies a 512 point DFT. There are some differences to how this is done when compared to Example 2.27, however. In our example we split the sound into disjoint blocks, and applied a DFT to each of them. The MP3 standard actually splits the sound into blocks which overlap, as this creates a more continuous frequency representation. Another difference is that the MP3 standard applies a *window* to the sound samples, and the effect of this is that the new signal has a frequency representation which is closer to the original one, when compared to the signal obtained by using the block values unchanged as above. We will go into details on this in Section 3.3.1.

Example 2.28. *Compression by zeroing out DFT coefficients.*

We can achieve compression of a sound by setting small DFT coefficients which to zero. The idea is that frequencies with small values at the corresponding frequency indices contribute little to our perception of the sound, so that they can be discarded. As a result we obtain a sound with less frequency components, which is thus more suitable for compression. To test this in practice, we first need to set a threshold, which decides which frequencies to keep. The following code then sets frequencies below the threshold to zero:

```
threshold = 50;
y = fft(x);
y = (abs(y) >= threshold).*y;
newx = ifft(y);
```

In this code 1 represents a value of `true` in the logical expression which is evaluated, 0 represents false. The value is 1 if and only if the absolute value of the corresponding element is greater than or equal to `threshold`. As in the previous example, we can apply this code to small blocks of the signal at a time, and listen to the result by playing it. We have implemented a function `playDFTthreshold(threshold)` which splits our sample audio file into blocks of the same size as above, applies the code above with the given threshold, and

plays the result. The code also writes to the display how large percentage of the DFT indices were set to 0. If you run this function with `threshold` equal to 0.02, the result sounds like this, and the function says that about 74.1% of the DFT indices were set to zero. You can clearly hear the disturbance in the sound, but we have not lost that much. If we instead try `threshold` equal to 0.1, the result will sound like this, and the function says that about 93.5% of the DFT indices were set to zero. The quality is much poorer now, even if we still can recognize the song. This suggests that most of the frequency information is contained in frequencies with the highest values. In figure 2.6 we have illustrated this principle for compression.

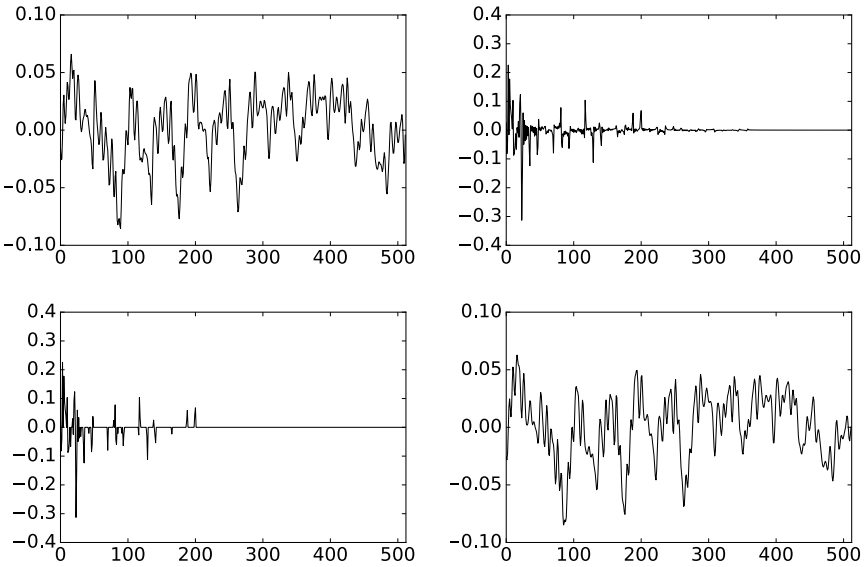

Figure 2.6: Experimenting with the DFT on a small part (512 sound samples) of a song. The upper two plots show the sound samples and their DCT. The lower plots show first what is left after the coefficients smaller than 0.02 have been set to 0 (52 values remain), and then the reconstructed sound.

The samples of the sound are shown in (a) and (the absolute value of) the DFT in (b). In (c) all values of the DFT with absolute value smaller than 0.02 have been set to zero. The sound is then reconstructed with the IDFT, and the result shown in (d). The two signals in (a) and (d) visually look almost the same even though the signal in (d) can be represented with less than 10 % of the information present in (a).

Note that using a neglection threshold in this way is too simple in practice: The neglection threshold in general should depend on the frequency, since the human auditory system is more sensitive to certain frequencies.

Example 2.29. *Compression by quantizing DFT coefficients.*

The previous example is a rather simple procedure to obtain compression. The disadvantage is that it only affects frequencies with low contribution. A more neutral way to obtain compression is to let each DFT index occupy a certain number of bits. This is also called *quantization*, and provides us with compression if the number of bits is less than what actually is used to represent the sound. This is closer to what modern audio standards do. Consider the following code:

```
n = 5;
y = fft(x);
y = y/2^n;
y = round(y);
y = y*2^n;
newx = ifft(y);
playerobj=audioplayer(newx, fs);
playblocking(playerobj);
```

The effect of the middle lines is that a number with bit representation

$$...d_2d_1d_0.d_{-1}d_{-2}d_{-3}...$$

is truncated so that the bits d_{n-1}, d_{n-2}, d_{n-2} are discarded. In other words, high values of n mean more rounding. We have implemented a function `playDFTquantized(n)` which executes this code and plays the result, in the same way as in the examples above. If you run this function with `n` equal to -3, the result sounds like this, with $n = -1$ the result sounds like this, and with $n = 1$ the result sounds like this. You can hear that the sound degrades further when n is increased.

In practice this quantization procedure is also too simple, since the human auditory system is more sensitive to certain frequency information, and should thus allocate a higher number of bits for such frequencies. Modern audio standards take this into account, but we will not go into details on this.

What you should have learned in this section.

- Translation between DFT index and frequency. In particular DFT indices for high and low frequencies.
- How one can use the DFT to adjust frequencies in sound.

Exercise 2.19: Comment code

Explain what the code below does, line by line:

```
[x, fs] = audioread('sounds/castanets.wav');
N = size(x, 1);
y = fft(x);
y((round(N/4)+1):(round(N/4)+N/2), :) = 0;
newx = abs(ifft(y));
```

```
newx = newx/max(max(newx));
playerobj = audioplayer(newx,fs);
playblocking(playerobj)
```

Comment in particular why we adjust the sound samples by dividing with the maximum value of the sound samples. What changes in the sound do you expect to hear?

Exercise 2.20: Which frequency is changed?

In the code from the previous exercise it turns out that $f_s = 44100$Hz. Which frequencies in the sound file will be changed on the line where we zero out some of the DFT coefficients?

Exercise 2.21: Implement interpolant

Implement code where you do the following:

- at the top you define the function $f(x) = \cos^6(x)$, and $M = 3$,
- compute the unique interpolant from $V_{M,T}$ (i.e. by taking $N = 2M + 1$ samples over one period), as guaranteed by Theorem 2.21,
- plot the interpolant against f over one period.

Finally run the code also for $M = 4$, $M = 5$, and $M = 6$. Explain why the plots coincide for $M = 6$, but not for $M < 6$. Does increasing M above $M = 6$ have any effect on the plots?

2.4 The Fast Fourier Transform (FFT)

The main application of the DFT is as a tool to compute frequency information in large datasets. Since this is so useful in many areas, it is of vital importance that the DFT can be computed with efficient algorithms. The straightforward implementation of the DFT with matrix multiplication we looked at is not efficient for large data sets. However, it turns out that the DFT matrix may be factored in a way that leads to much more efficient algorithms, and this is the topic of the present section. We will discuss the most widely used implementation of the DFT, usually referred to as the Fast Fourier Transform (FFT). The FFT has been stated as one of the ten most important inventions of the 20'th century, and its invention made the DFT computationally feasible in many fields. The FFT is for instance used much in real time processing, such as processing and compression of sound, images, and video. The MP3 standard uses the FFT to find frequency components in sound, and matches this information with a psychoachoustic model, in order to find the best way to compress the data.

Let us start with the most basic FFT algorithm, which applies for a general complex input vector $\boldsymbol{x}$, with length N being an even number.

Theorem 2.30. *FFT algorithm when N is even.*

Let $\boldsymbol{y} = \mathrm{DFT}_N \boldsymbol{x}$ be the N-point DFT of $\boldsymbol{x}$, with N an even number, and let $D_{N/2}$ be the $(N/2) \times (N/2)$-diagonal matrix with entries $(D_{N/2})_{n,n} = e^{-2\pi i n/N}$ for $0 \leq n < N/2$. Then we have that

$$(y_0, y_1, \ldots, y_{N/2-1}) = \mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)} + D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)} \tag{2.13}$$

$$(y_{N/2}, y_{N/2+1}, \ldots, y_{N-1}) = \mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)} - D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)} \tag{2.14}$$

where $\boldsymbol{x}^{(e)}, \boldsymbol{x}^{(o)} \in \mathbb{R}^{N/2}$ consist of the even- and odd-indexed entries of $\boldsymbol{x}$, respectively, i.e.

$$\boldsymbol{x}^{(e)} = (x_0, x_2, \ldots, x_{N-2}) \qquad \boldsymbol{x}^{(o)} = (x_1, x_3, \ldots, x_{N-1}).$$

Put differently, the formulas (2.13)-(2.14) reduce the computation of an N-point DFT to two $N/2$-point DFT's. It turns out that this is the basic fact which speeds up computations considerably. It is important to note that we first should compute that the same term $D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)}$ appears in both formulas above. It is thus important that this is computed only once, and then inserted in both equations. Let us first check that these formulas are correct.

Proof. Suppose first that $0 \leq n \leq N/2 - 1$. We start by splitting the sum in the expression for the DFT into even and odd indices,

$$\begin{aligned} y_n &= \sum_{k=0}^{N-1} x_k e^{-2\pi i n k/N} = \sum_{k=0}^{N/2-1} x_{2k} e^{-2\pi i n 2k/N} + \sum_{k=0}^{N/2-1} x_{2k+1} e^{-2\pi i n(2k+1)/N} \\ &= \sum_{k=0}^{N/2-1} x_{2k} e^{-2\pi i n k/(N/2)} + e^{-2\pi i n/N} \sum_{k=0}^{N/2-1} x_{2k+1} e^{-2\pi i n k/(N/2)} \\ &= \left(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)}\right)_n + e^{-2\pi i n/N} \left(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)}\right)_n, \end{aligned}$$

where we have substituted $\boldsymbol{x}^{(e)}$ and $\boldsymbol{x}^{(o)}$ as in the text of the theorem, and recognized the $N/2$-point DFT in two places. Assembling this for $0 \leq n < N/2$ we obtain Equation (2.13). For the second half of the DFT coefficients, i.e. $\{y_{N/2+n}\}_{0 \leq n \leq N/2-1}$, we similarly have

$$\begin{aligned} y_{N/2+n} &= \sum_{k=0}^{N-1} x_k e^{-2\pi i(N/2+n)k/N} = \sum_{k=0}^{N-1} x_k e^{-\pi ik} e^{-2\pi ink/N} \\ &= \sum_{k=0}^{N/2-1} x_{2k} e^{-2\pi in2k/N} - \sum_{k=0}^{N/2-1} x_{2k+1} e^{-2\pi in(2k+1)/N} \\ &= \sum_{k=0}^{N/2-1} x_{2k} e^{-2\pi ink/(N/2)} - e^{-2\pi in/N} \sum_{k=0}^{N/2-1} x_{2k+1} e^{-2\pi ink/(N/2)} \\ &= \left(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)}\right)_n - e^{-2\pi in/N}\left(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)}\right)_n. \end{aligned}$$

Equation (2.14) now follows similarly. □

Note that an algorithm for the IDFT can be deduced in exactly the same way. All we need to change is the sign in the exponents of the Fourier matrix. In addition we need to divide by $1/N$ at the end. If we do this we get the following result, which we call the IFFT algorithm. Recall that we use the notation $\overline{A}$ for the matrix where all the elements of A have been conjugated.

Theorem 2.31. *IFFT algorithm when N is even.*
Let N be an even number and let $\tilde{\boldsymbol{x}} = \overline{\mathrm{DFT}_N}\boldsymbol{y}$. Then we have that

$$(\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{N/2-1}) = \overline{\mathrm{DFT}_{N/2}}\boldsymbol{y}^{(e)} + \overline{D_{N/2}\mathrm{DFT}_{N/2}})\boldsymbol{y}^{(o)} \tag{2.15}$$

$$(\tilde{x}_{N/2}, \tilde{x}_{N/2+1}, \ldots, \tilde{x}_{N-1}) = \overline{\mathrm{DFT}_{N/2}}\boldsymbol{y}^{(e)} - \overline{D_{N/2}\mathrm{DFT}_{N/2}})\boldsymbol{y}^{(o)} \tag{2.16}$$

where $\boldsymbol{y}^{(e)}, \boldsymbol{y}^{(o)} \in \mathbb{R}^{N/2}$ are the vectors

$$\boldsymbol{y}^{(e)} = (y_0, y_2, \ldots, y_{N-2}) \qquad \boldsymbol{y}^{(o)} = (y_1, y_3, \ldots, y_{N-1}).$$

Moreover, $\boldsymbol{x} = \mathrm{IDFT}_N\boldsymbol{y}$ can be computed from $\boldsymbol{x} = \tilde{\boldsymbol{x}}/N = \overline{\mathrm{DFT}_N}\boldsymbol{y}/N$

It turns out that these theorems can be interpreted as matrix factorizations. For this we need to define the concept of a block matrix.

Definition 2.32. *Block matrix.*
Let $m_0, \ldots, m_{r-1}$ and $n_0, \ldots, n_{s-1}$ be integers, and let $A^{(i,j)}$ be an $m_i \times n_j$-matrix for $i = 0, \ldots, r-1$ and $j = 0, \ldots, s-1$. The notation

$$A = \begin{pmatrix} A^{(0,0)} & A^{(0,1)} & \cdots & A^{(0,s-1)} \\ A^{(1,0)} & A^{(1,1)} & \cdots & A^{(1,s-1)} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(r-1,0)} & A^{(r-1,1)} & \cdots & A^{(r-1,s-1)} \end{pmatrix}$$

denotes the $(m_0 + m_1 + \ldots + m_{r-1}) \times (n_0 + n_1 + \ldots + n_{s-1})$-matrix where the matrix entries occur as in the $A^{(i,j)}$ matrices, in the way they are ordered. When A is written in this way it is referred to as a block matrix .

Clearly, using equations (2.13)-(2.14), the DFT matrix can be factorized using block matrix notation as

$$(y_0, y_1, \ldots, y_{N/2-1}) = \begin{pmatrix}\mathrm{DFT}_{N/2} & D_{N/2}\mathrm{DFT}_{N/2}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(e)}\\ \boldsymbol{x}^{(o)}\end{pmatrix}$$
$$(y_{N/2}, y_{N/2+1}, \ldots, y_{N-1}) = \begin{pmatrix}\mathrm{DFT}_{N/2} & -D_{N/2}\mathrm{DFT}_{N/2}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(e)}\\ \boldsymbol{x}^{(o)}\end{pmatrix}.$$

Combining these, noting that

$$\begin{pmatrix}\mathrm{DFT}_{N/2} & D_{N/2}\mathrm{DFT}_{N/2}\\ \mathrm{DFT}_{N/2} & -D_{N/2}\mathrm{DFT}_{N/2}\end{pmatrix} = \begin{pmatrix}I & D_{N/2}\\ I & -D_{N/2}\end{pmatrix}\begin{pmatrix}\mathrm{DFT}_{N/2} & \mathbf{0}\\ \mathbf{0} & \mathrm{DFT}_{N/2}\end{pmatrix},$$

we obtain the following factorisations:

Theorem 2.33. *DFT and IDFT matrix factorizations.*
We have that

$$\mathrm{DFT}_N\boldsymbol{x} = \begin{pmatrix}I & D_{N/2}\\ I & -D_{N/2}\end{pmatrix}\begin{pmatrix}\mathrm{DFT}_{N/2} & \mathbf{0}\\ \mathbf{0} & \mathrm{DFT}_{N/2}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(e)}\\ \boldsymbol{x}^{(o)}\end{pmatrix}$$
$$\mathrm{IDFT}_N\boldsymbol{y} = \frac{1}{N}\overline{\begin{pmatrix}I & D_{N/2}\\ I & -D_{N/2}\end{pmatrix}}\begin{pmatrix}\overline{\mathrm{DFT}_{N/2}} & \mathbf{0}\\ \mathbf{0} & \overline{\mathrm{DFT}_{N/2}}\end{pmatrix}\begin{pmatrix}\boldsymbol{y}^{(e)}\\ \boldsymbol{y}^{(o)}\end{pmatrix} \tag{2.17}$$

We will shortly see why these factorizations reduce the number of arithmetic operations we need to do, but first let us consider how to implement them. First of all, note that we can apply the FFT factorizations again to $F_{N/2}$ to obtain

$$\mathrm{DFT}_N\boldsymbol{x} = \begin{pmatrix}I & D_{N/2}\\ I & -D_{N/2}\end{pmatrix}\begin{pmatrix}I & D_{N/4} & \mathbf{0} & \mathbf{0}\\ I & -D_{N/4} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & I & D_{N/4}\\ \mathbf{0} & \mathbf{0} & I & -D_{N/4}\end{pmatrix}\times$$
$$\begin{pmatrix}\mathrm{DFT}_{N/4} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathrm{DFT}_{N/4} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathrm{DFT}_{N/4} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathrm{DFT}_{N/4}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(ee)}\\ \boldsymbol{x}^{(eo)}\\ \boldsymbol{x}^{(oe)}\\ \boldsymbol{x}^{(oo)}\end{pmatrix}$$

where the vectors $\boldsymbol{x}^{(e)}$ and $\boldsymbol{x}^{(o)}$ have been further split into even- and odd-indexed entries. Clearly, if this factorization is repeated, we obtain a factorization

$$\mathrm{DFT}_N = \prod_{k=1}^{\log_2 N} \begin{pmatrix} I & D_{N/2^k} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ I & -D_{N/2^k} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I & D_{N/2^k} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I & -D_{N/2^k} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & I & D_{N/2^k} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & I & -D_{N/2^k} \end{pmatrix} P. \quad (2.18)$$

The factorization has been repated until we have a final diagonal matrix with DFT_1 on the diagonal, but clearly $\mathrm{DFT}_1 = 1$, so we do not need any DFT-matrices in the final factor. Note that all matrices in this factorization are sparse. A factorization into a product of sparse matrices is the key to many efficient algorithms in linear algebra, such as the computation of eigenvalues and eigenvectors. When we later compute the number of arithmetic operations in this factorization, we will see that this is the case also here.

In Equation (2.18), P is a permutation matrix which secures that the even-indexed entries come first. Since the even-indexed entries have 0 as the last bit, this is the same as letting the last bit become the first bit. Since we here recursively place even-indexed entries first, it is not too difficult to see that P permutes the elements of $\boldsymbol{x}$ by performing a *bit-reversal* of the indices, i.e.

$$P(\boldsymbol{e}_i) = \boldsymbol{e}_j \qquad i = d_1 d_2 \dots d_n \qquad j = d_n d_{n-1} \dots d_1,$$

where we have used the bit representations of i and j. Since $P^2 = I$, a bit-reversal can be computed very efficiently, and performed *in-place*, i.e. so that the result ends up in same vector $\boldsymbol{x}$, so that we do not need to allocate any memory in this operation. We will use an existing function called `bitreverse` to perfom in-place bit-reversal. In exercise 2.30 we will go through this implementation.

Matrix multiplication is usually not done in-place, i.e. when we compute $\boldsymbol{y} = A\boldsymbol{x}$, different memory is allocated for $\boldsymbol{x}$ and $\boldsymbol{y}$. For certain simple matrices, however, matrix multiplication can also be done in-place, so that the output can be written into the same memory ($\boldsymbol{x}$) used by the input. It turns out that the matrices in factorization (2.18) are of this kind, so that the entire FFT can be computed in-place . We will have more to say on this in the exercises.

In a practical algorithm, it is smart to perform the bit-reversal first, since the matrices in the factorization (2.18) are block diagonal, so that the different blocks in each matrix can be applied in parallel to $P\boldsymbol{x}$ (the bit-reversed version of $\boldsymbol{x}$). We can thus exploit the parallel processing capabilities of the computer. It turns out that this bit-reversal is useful for other similar factorizations of the DFT as well. We will also look at other such factorizations, and we will therefore split the computation of the DFT as follows: First a general function is applied, which is responsible for the bit-reversal of the input vector $\boldsymbol{x}$. Then the matrices in the factorization (2.18) is applied in a "kernel FFT function" (and we will

have many such kernels), which assumes that the input has been bit-reversed. A simple implementation of the general function can be as follows.

```
function y = FFTImpl(x, FFTKernel)
    x = bitreverse(x);
    y = FFTKernel(x);
```

A simple implementation of the kernel FFT function, based on the first FFT algorithm we stated, can be as follows.

```
function y = FFTKernelStandard(x)
    N = size(x, 1);
    if N == 1
        y = x;
    else
        xe = FFTKernelStandard(x(1:(N/2)));
        xo = FFTKernelStandard(x((N/2+1):N));
        D = exp(-2*pi*1j*(0:(N/2-1))'/N);
        xo = xo.*D;
        y = [ xe + xo; xe - xo];
    end
```

Note that, although computations can be performed in-place, this Matlab implementation does not, since return values and parameters to functions are copied in Matlab. In exercise 2.22 we will extend these to the general implementations we will use later. We can now run the FFT by combining the general function and the kernel as follows:

```
y = FFTImpl(x, @FFTKernelStandard);
```

Note that `FFTKernelStandard` is recursive; it calls itself. If this is your first encounter with a recursive program, it is worth running through the code manually for a given value of N, such as $N = 4$.

Immediately we see from factorization (2.18) two possible implementations for a kernel. First, as we did, we can apply the FFT recursively. A second way is to, instead of using recursive function calls, use a for-loop where we at each stage in the loop compute the product with one matrix in factorization (2.18), from right to left. Inside this loop there must be another for-loop, where the different blocks in this matrix are applied. We will establish this non-recursive implementation in exercise 2.28, and see that this leads to a more efficient algorithm.

Matlab has built-in functions for computing the DFT and the IDFT using the FFT algorithm. The functions are called `fft` and `ifft`. These functions make no assumption about the length of the vector, i.e. it may not be of even length. The implementation may however check if the length of the vector is 2^r, and in those cases variants of the algorithm discussed here can be used. In general, fast algorithms exist when the vector length N can be factored as a product of small integers.

2.4.1 Reduction in the number of multiplications with the FFT

Now we will explain why the FFT and IFFT factorizations reduce the number of arithmetic operations when compared to direct DFT and IDFT implementations. We will assume that $\boldsymbol{x} \in \mathbb{R}^N$ with N a power of 2, so that the FFT algorithm can be used recursively, all the way down to vectors of length 1. In many settings this power of 2 assumption can be done. As an example, in compression of sound, one restricts processing to a certain block of the sound data, since the entire sound is too big to be processed in one piece. One then has a freedom to how big these blocks are made, and for optimal speed one often uses blocks of length 2^r with r some integer in the range 5–10. At the end of this section we will explain how the more general FFT can be computed when N is not a power of 2.

We first need some terminology for how we count the number of operations of a given type in an algorithm. In particular we are interested in the limiting behaviour when N becomes large, which is the motivation for the following definition.

Definition 2.34. *Order of an algorithm.*

Let R_N be the number of operations of a given type (such as multiplication or addition) in an algorithm, where N describes the dimension of the data (such as the size of the matrix or length of the vector), and let f be a positive function. The algorithm is said to be of order N, also written $O(f(N))$, if the number of operations grows as $f(N)$ for large N, or more precisely, if

$$\lim_{N\to\infty} \frac{R_N}{f(N)} = 1.$$

In some situations we may count the number of operations exactly, but we will also see that it may be easier to obtain the order of the algorithm, since the number of operations may have a simpler expression in the limit. Let us see how we can use this terminology to describe the complexity of the FFT algorithm. Let M_N and A_N denote the number of real multiplications and real additions, respectively, required by the FFT algorithm. Once the FFT's of order $N/2$ have been computed ($M_{N/2}$ real multiplications and $A_{N/2}$ real additions are needed for each), it is clear from equations (2.13)-(2.14) that an additional N complex additions, and an additional $N/2$ complex multiplications, are required. Since one complex multiplication requires 4 real multiplications and 2 real additions, and one complex addition requires two real additions, we see that we require an additional $2N$ real multiplications, and $2N + N = 3N$ real additions. This means that we have the difference equations

$$M_N = 2M_{N/2} + 2N \qquad\qquad A_N = 2A_{N/2} + 3N. \tag{2.19}$$

Note that $e^{-2\pi i/N}$ may be computed once and for all and outside the algorithm, and this is the reason why we have not counted these operations.

The following example shows how the difference equations (2.19) can be solved. It is not too difficult to argue that $M_N = O(2N \log_2 N)$ and $A_N = O(3N \log_2)$, by noting that there are $\log_2 N$ levels in the FFT, with $2N$ real multiplications and real $3N$ additions at each level. But for $N = 2$ and $N = 4$ we may actually avoid some multiplications, so we should solve these equations by stating initial conditions carefully, in order to obtain exact operation counts. In practice, and as we will see later, one often has more involved equations than (2.19), for which the solution can not be seen directly, so that one needs to apply systematic mathematical methods instead, such as in the example below.

Example 2.35. *Solving for the number of operations.*

To use standard solution methods for difference equations to equations (2.19), we first need to write them in a standard form. Assuming that A_N and M_N are powers of 2, we set $N = 2^r$ and $x_r = M_{2^r}$, or $x_r = A_{2^r}$. The difference equations can then be rewritten as $x_r = 2x_{r-1} + 2 \cdot 2^r$ for multiplications, and $x_r = 2x_{r-1} + 3 \cdot 2^r$ for additions, and again be rewritten in the standard forms

$$x_{r+1} - 2x_r = 4 \cdot 2^r \qquad\qquad x_{r+1} - 2x_r = 6 \cdot 2^r.$$

The homogeneous equation $x_{r+1} - 2x_r = 0$ has the general solution $x_r^h = C2^r$. Since the base in the power on the right hand side equals the root in the homogeneous equation, we should in each case guess for a particular solution on the form $(x_p)_r = Ar2^r$. If we do this we find that the first equation has particular solution $(x_p)_r = 2r2^r$, while the second has particular solution $(x_p)_r = 3r2^r$. The general solutions are thus on the form $x_r = 2r2^r + C2^r$, for multiplications, and $x_r = 3r2^r + C2^r$ for additions.

Now let us state initial conditions for the number of additions and multiplications. Example 2.16 showed that floating point multiplication can be avoided completely for $N = 4$. We can therefore use $M_4 = x_2 = 0$ as an initial value. This gives, $x_r = 2r2^r - 4 \cdot 2^r$, so that $M_N = 2N \log_2 N - 4N$.

For additions we can use $A_2 = x_1 = 4$ as initial value (since $\text{DFT}_2(x_1, x_2) = (x_1 + x_2, x_1 - x_2)$), which gives $x_r = 3r2^r$, so that $A_N = 3N \log_2 N - N$. Our FFT algorithm thus requires slightly more additions than multiplications. FFT algorithms are often characterized by their *operation count*, i.e. the total number of real additions and real multiplications, i.e. $R_N = M_N + A_N$. We see that $R_N = 5N \log_2 N - 5N$. The order of the operation count of our algorithm can thus be written as $O(5N \log_2 N)$, since $\lim_{N\to\infty} \frac{5N \log_2 N - 4N}{5N \log_2 N} = 1$.

In practice one can reduce the number of multiplications further, since $e^{-2\pi in/N}$ take the simple values $1, -1, -i, i$ for some n. One can also use that $e^{-2\pi in/N}$ can take the simple values $\pm 1/\sqrt{2} \pm 1/\sqrt{2}i = 1/\sqrt{2}(\pm 1 \pm i)$, which also saves some floating point multiplication, due to that we can factor out $1/\sqrt{2}$. These observations do not give big reductions in the arithmetic complexity, however, and one can show that the operation count is still $O(5N \log_2 N)$ after using these observations.

It is straightforward to show that the IFFT implementation requires the same operation count as the FFT algorithm.

In contrast, the direct implementation of the DFT requires N^2 complex multiplications and $N(N-1)$ complex additions. This results in $4N^2$ real multiplications and $2N^2 + 2N(N-1) = 4N^2 - 2N$ real additions. The total operation count is thus $8N^2 - 2N$. In other words, the FFT and IFFT significantly reduce the number of arithmetic operations. In Exercise 2.29 we present another algorithm, called the Split-radix algorithm, which reduces the number of operations even further. We will see, however, the reduction obtained with the split-radix algorithm is about 20%. Let us summarize our findings as follows.

Theorem 2.36. *Number of operations in the FFT and IFFT algorithms.*

The N-point FFT and IFFT algorithms we have gone through both require $O(2N \log_2 N)$ real multiplications and $O(3N \log_2 N)$ real additions. In comparison, the number of real multiplications and real additions required by direct implementations of the N-point DFT and IDFT are $O(8N^2)$.

Often we apply the DFT for real data, so we would like to have FFT-algorithms tailored to this, with reduced complexity (since real data has half the dimension of general complex data). By some it has been argued that one can find improved FFT algorithms when one assumes that the data is real. In exercise 2.27 we address this issue, and conclude that there is little to gain from assuming real input: The general algorithm for complex input can be tailored for real input so that it uses half the number of operations, which harmonizes with the fact that real data has half the dimension of complex data.

Another reason why the FFT is efficient is that, since the FFT splits the calculation of the DFT into computing two DFT's of half the size, the FFT is well suited for parallel computing: the two smaller FFT's can be performed independently of one another, for instance in two different computing cores on the same computer. Besides reducing the number of arithmetic operations, FFT implementation can also apply several programming tricks to speed up computation, see for instance http://cnx.org/content/m12021/latest/ for an overview.

2.4.2 The FFT when $N = N_1 N_2$

Applying an FFT to a vector of length 2^n is by far the most common thing to do. It turns out, however, that the idea behind the algorithm easily carries over to the case when N is any composite number, i.e. when $N = N_1 N_2$. This make the FFT useful also in settings where we have a dictated number of elements in $\boldsymbol{x}$, which is not an even number. The approach we will present in this section will help us as long as N is not a prime number. The case when N is a prime number needs other techniques.

So, assume that $N = N_1 N_2$. Any time-index k can be written uniquely on the form $N_1 k + p$, with $0 \leq k < N_2$, and $0 \leq p < N_1$. We will make the following definition.

Definition 2.37. *Polyphase components of a vector.*

Let $\boldsymbol{x} \in \mathbb{R}^{N_1 N_2}$. We denote by $\boldsymbol{x}^{(p)}$ the vector in $\mathbb{R}^{N_2}$ with entries $(\boldsymbol{x}^{(p)})_k = x_{N_1 k + p}$. $\boldsymbol{x}^{(p)}$ is also called the p'th *polyphase component* of $\boldsymbol{x}$.

The previous vectors $\boldsymbol{x}^{(e)}$ and $\boldsymbol{x}^{(o)}$ can be seen as special cases of polyphase components. Polyphase components will also be useful later (see Chapter 8). Using the polyphase notation, we can write

$$\begin{aligned}\mathrm{DFT}_N\boldsymbol{x} &= \sum_{k=0}^{N-1} x_k e^{-2\pi ink/N} = \sum_{p=0}^{N_1-1}\sum_{k=0}^{N_2-1} (\boldsymbol{x}^{(p)})_k e^{-2\pi in(N_1k+p)/N} \\ &= \sum_{p=0}^{N_1-1} e^{-2\pi inp/N} \sum_{k=0}^{N_2-1} (\boldsymbol{x}^{(p)})_k e^{-2\pi ink/N_2}\end{aligned}$$

Similarly, any frequency index n can be written uniquely on the form N_2q+n, with $0 \leq q < N_1$, and $0 \leq n < N_2$, so that the DFT can also be written as

$$\begin{aligned}&\sum_{p=0}^{N_1-1} e^{-2\pi i(N_2q+n)p/N} \sum_{k=0}^{N_2-1} (\boldsymbol{x}^{(p)})_k e^{-2\pi i(N_2q+n)k/N_2} \\ &= \sum_{p=0}^{N_1-1} e^{-2\pi iqp/N_1} e^{-2\pi inp/N} \sum_{k=0}^{N_2-1} (\boldsymbol{x}^{(p)})_k e^{-2\pi ink/N_2}.\end{aligned}$$

Now, if X is the $N_2 \times N_1$-matrix X where the p'th column is $\boldsymbol{x}^{(p)}$, we recognize the inner sum $\sum_{k=0}^{N_2-1}(\boldsymbol{x}^{(p)})_k e^{-2\pi ink/N_2}$ as matrix multiplication with DFT_{N_2} and X, so that this can be written as $(\mathrm{DFT}_{N_2}X)_{n,p}$. The entire sum can thus be written as

$$\sum_{p=0}^{N_1-1} e^{-2\pi iqp/N_1} e^{-2\pi inp/N} (\mathrm{DFT}_{N_2}X)_{n,p}.$$

Now, define Y as the matrix where X is multiplied component-wise with the matrix with (n,p)-component $e^{-2\pi inp/N}$. The entire sum can then be written as

$$\sum_{p=0}^{N_1-1} e^{-2\pi iqp/N_1} Y_{n,p} = (YF_{N_1})_{n,q}$$

This means that the sum can be written as component (n,q) in the matrix YF_{N_1}. Clearly YF_{N_1} is the matrix where the DFT is applied to all rows of Y. We have thus shown that component N_2q+n of $F_N\boldsymbol{x}$ equals $(YF_{N_1})_{n,q}$. This means that $F_N\boldsymbol{x}$ can be obtained by stacking the columns of YF_{N_1} on top of one-another. We can thus summarize our procedure as follows, which gives a recipe for splitting an FFT into smaller FFT's when N is not a prime number.

Theorem 2.38. *FFT algorithm when N is composite.*

When $N = N_1N_2$, the FFT of a vector $\boldsymbol{x}$ can be computed as follows

- Form the $N_2 \times N_1$-matrix X, where the p'th column is $\boldsymbol{x}^{(p)}$.

- Perform the DFT on all the columns in X, i.e. compute $F_{N_2}X$.
- Multiply element (n, p) in the resulting matrix with $e^{-2\pi inp/N}$ (these are called *twiddle factors*), to obtain matrix Y.
- Perform the DFT on all the rows in the resulting matrix, i.e. compute YF_{N_1}.
- Form the vector where the columns of the resulting matrix are stacked on top of one-another.

From the algorithm one easily deduces how the IDFT can be computed also: All steps are invertible, and can be performed by IFFT or multiplication. We thus only need to perform the inverse steps in reverse order.

But what about the case when N is a prime number? Rader's algorithm [29] handles this case by expressing a DFT with N a prime number in terms of DFT's of length $N-1$ (which is not a prime number). Our previous scenario can then be followed, but stops quickly again if $N-1$ has prime factors of high order. Since there are some computational penalties in applying Rader's algorithm, it may be inefficient some cases. Winograd's FFT algorithm [39] extends Rader's algorithm to work for the case when $N = p^r$. This algorithm tends to reduce the number of multiplications, at the price of an increased number of additions. It is difficult to program, and is rarely used in practice.

What you should have learned in this section.

- How the FFT algorithm works by splitting into two FFT's of half the length.
- Simple FFT implementation.
- Reduction in the number of operations with the FFT.

Exercise 2.22: Extend implementation

Recall that, in exercise 2.16, we extended the direct DFT implementation so that it accepted a second parameter telling us if the forward or reverse transform should be applied. Extend the general function and the standard kernel in the same way. Again, the forward transform should be used if the `forward` parameter is not present. Assume also that the kernel accepts only one-dimensional data, and that the general function applies the kernel to each column in the input if the input is two-dimensional (so that the FFT can be applied to all channels in a sound with only one call). The signatures for our methods should thus be changed as follows:

```
function y = FFTImpl(x, FFTKernel, forward)
function y = FFTKernelStandard(x, forward)
```

It should be straightforward to make the modifications for the reverse transform by consulting the second part of Theorem 2.33. For simplicity, let `FFTImpl` take care of the additional division with N we need to do in case of the IDFT. In the following we will assume these signatures for the FFT implementation and the corresponding kernels.

Exercise 2.23: Compare execution time

In this exercise we will compare execution times for the different methods for computing the DFT.

a) Write code which compares the execution times for an N-point DFT for the following three cases: Direct implementation of the DFT (as in Example 2.17), the FFT implementation used in this chapter, and the built-in `fft`-function. Your code should use the sample audio file `castanets.wav`, apply the different DFT implementations to the first $N = 2^r$ samples of the file for $r = 3$ to $r = 15$, store the execution times in a vector, and plot these. You can use the functions `tic` and `toc` to measure the execution time.

b) A problem for large N is that there is such a big difference in the execution times between the two implementations. We can address this by using a loglog-plot instead. Plot N against execution times using the function `loglog`. How should the fact that the number of arithmetic operations are $8N^2$ and $5N \log_2 N$ be reflected in the plot?

c) It seems that the built-in FFT is much faster than our own FFT implementation, even though they may use similar algorithms. Try to explain what can be the cause of this.

Exercise 2.24: Combine two FFT's

Let $\boldsymbol{x}_1 = (1, 3, 5, 7)$ and $\boldsymbol{x}_2 = (2, 4, 6, 8)$. Compute $\text{DFT}_4\boldsymbol{x}_1$ and $\text{DFT}_4\boldsymbol{x}_2$. Explain how you can compute $\text{DFT}_8(1, 2, 3, 4, 5, 6, 7, 8)$ based on these computations (you don't need to perform the actual computation). What are the benefits of this approach?

Exercise 2.25: Composite FFT

When N is composite, there are a couple of results we can state regarding polyphase components.

a) Assume that $N = N_1N_2$, and that $\boldsymbol{x} \in \mathbb{R}^N$ satisfies $x_{k+rN_1} = x_k$ for all k, r, i.e. $\boldsymbol{x}$ has period N_1. Show that $y_n = 0$ for all n which are not a multiplum of N_2.

b) Assume that $N = N_1 N_2$, and that $\boldsymbol{x}^{(p)} = \mathbf{0}$ for $p \neq 0$. Show that the polyphase components $\boldsymbol{y}^{(p)}$ of $\boldsymbol{y} = \mathrm{DFT}_N \boldsymbol{x}$ are constant vectors for all p.

Exercise 2.26: FFT operation count

When we wrote down the difference equation for the number of multiplications in the FFT algorithm, you could argue that some multiplications were not counted. Which multiplications in the FFT algorithm were not counted when writing down this difference equation? Do you have a suggestion to why these multiplications were not counted?

Exercise 2.27: Adapting the FFT algorithm to real data

In this exercise we will look at an approach to how we can adapt an FFT algorithm to real input $\boldsymbol{x}$. We will now instead rewrite Equation (2.13) for indices n and $N/2 - n$ as

$$\begin{aligned} y_n &= (\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n + e^{-2\pi i n/N}(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n \\ y_{N/2-n} &= (\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_{N/2-n} + e^{-2\pi i (N/2-n)/N}(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_{N/2-n} \\ &= (\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_{N/2-n} - e^{2\pi i n/N}\overline{(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n} \\ &= \overline{(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n} - \overline{e^{-2\pi i n/N}(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n}. \end{aligned}$$

We see here that, if we have computed the terms in y_n (which needs an additional 4 real multiplications, since $e^{-2\pi i n/N}$ and $(\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n$ are complex), no further multiplications are needed in order to compute $y_{N/2-n}$, since its compression simply conjugates these terms before adding them. Again $y_{N/2}$ must be handled explicitly with this approach. For this we can use the formula

$$y_{N/2} = (\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_0 - (D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_0$$

instead.

a) Conclude from this that an FFT algorithm adapted to real data at each step requires $N/4$ complex additions and $N/2$ additions. Conclude from this as before that an algorithm based on real data requires $M_N = O(N \log_2 N)$ multiplications and $A_N = O\left(\frac{3}{2} N \log_2 N\right)$ additions (i.e. again we obtain half the operation count of complex input).

b) Find an IFFT algorithm adapted to vectors $\boldsymbol{y}$ which have conjugate symmetry, which has the same operation count we found above.

Hint. Consider the vectors $y_n + \overline{y_{N/2-n}}$ and $e^{2\pi i n/N}(y_n - \overline{y_{N/2-n}})$. From the equations above, how can these be used in an IFFT?

Exercise 2.28: Non-recursive FFT algorithm

Use the factorization in Equation (2.18) to write a kernel function `FFTKernelNonrec` for a non-recursive FFT implementation. In your code, perform the matrix multiplications in Equation (2.18) from right to left in an (outer) for-loop. For each matrix loop through the different blocks on the diagonal in an (inner) for-loop. Make sure you have the right number of blocks on the diagonal, each block being on the form

$$\begin{pmatrix} I & D_{N/2^k} \\ I & -D_{N/2^k} \end{pmatrix}.$$

It may be a good idea to start by implementing multiplication with such a simple matrix first as these are the building blocks in the algorithm (also attempt to do this so that everything is computed in-place). Also compare the execution times with our original FFT algorithm, as we did in Exercise 2.23, and try to explain what you see in this comparison.

Exercise 2.29: The Split-radix FFT algorithm

In this exercise we will develop a variant of the FFT algorithm called the *split-radix FFT algorithm*, which until recently held the record for the lowest operation count for any FFT algorithm.

We start by splitting the rightmost $\text{DFT}_{N/2}$ in Equation (2.17) by using Equation (2.17) again, to obtain

$$\text{DFT}_N \boldsymbol{x} = \begin{pmatrix} \text{DFT}_{N/2} & D_{N/2} \begin{pmatrix} \text{DFT}_{N/4} & D_{N/4}\text{DFT}_{N/4} \\ \text{DFT}_{N/4} & -D_{N/4}\text{DFT}_{N/4} \end{pmatrix} \\ \text{DFT}_{N/2} & -D_{N/2} \begin{pmatrix} \text{DFT}_{N/4} & D_{N/4}\text{DFT}_{N/4} \\ \text{DFT}_{N/4} & -D_{N/4}\text{DFT}_{N/4} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(e)} \\ \boldsymbol{x}^{(oe)} \\ \boldsymbol{x}^{(oo)} \end{pmatrix}. \tag{2.20}$$

The term radix describes how an FFT is split into FFT's of smaller sizes, i.e. how the sum in an FFT is split into smaller sums. The FFT algorithm we started this section with is called a radix 2 algorithm, since it splits an FFT of length N into FFT's of length $N/2$. If an algorithm instead splits into FFT's of length $N/4$, it is called a radix 4 FFT algorithm. The algorithm we go through here is called the split radix algorithm, since it uses FFT's of both length $N/2$ and $N/4$.

a) Let $G_{N/4}$ be the $(N/4)\times(N/4)$ diagonal matrix with $e^{-2\pi i n/N}$ on the diagonal. Show that $D_{N/2} = \begin{pmatrix} G_{N/4} & \mathbf{0} \\ \mathbf{0} & -iG_{N/4} \end{pmatrix}$.

b) Let $H_{N/4}$ be the $(N/4) \times (N/4)$ diagonal matrix $G_{D/4}D_{N/4}$. Verify the following rewriting of Equation (2.20):

$$
\begin{aligned}
\mathrm{DFT}_N \boldsymbol{x} &= \begin{pmatrix} \mathrm{DFT}_{N/2} & \begin{pmatrix} G_{N/4}\mathrm{DFT}_{N/4} & H_{N/4}\mathrm{DFT}_{N/4} \\ -iG_{N/4}\mathrm{DFT}_{N/4} & iH_{N/4}\mathrm{DFT}_{N/4} \end{pmatrix} \\ \mathrm{DFT}_{N/2} & \begin{pmatrix} -G_{N/4}\mathrm{DFT}_{N/4} & -H_{N/4}\mathrm{DFT}_{N/4} \\ iG_{N/4}\mathrm{DFT}_{N/4} & -iH_{N/4}\mathrm{DFT}_{N/4} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(e)} \\ \boldsymbol{x}^{(oe)} \\ \boldsymbol{x}^{(oo)} \end{pmatrix} \\
&= \begin{pmatrix} I & \mathbf{0} & G_{N/4} & H_{N/4} \\ \mathbf{0} & I & -iG_{N/4} & iH_{N/4} \\ I & \mathbf{0} & -G_{N/4} & -H_{N/4} \\ \mathbf{0} & I & iG_{N/4} & -iH_{N/4} \end{pmatrix} \begin{pmatrix} \mathrm{DFT}_{N/2} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathrm{DFT}_{N/4} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathrm{DFT}_{N/4} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(e)} \\ \boldsymbol{x}^{(oe)} \\ \boldsymbol{x}^{(oo)} \end{pmatrix} \\
&= \begin{pmatrix} I & \begin{pmatrix} G_{N/4} & H_{N/4} \\ -iG_{N/4} & iH_{N/4} \end{pmatrix} \\ I & -\begin{pmatrix} G_{N/4} & H_{N/4} \\ -iG_{N/4} & iH_{N/4} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)} \\ \mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)} \\ \mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)} \end{pmatrix} \\
&= \begin{pmatrix} \mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)} + \begin{pmatrix} G_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)} + H_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)} \\ -i\left(G_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)} - H_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)}\right) \end{pmatrix} \\ \mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)} - \begin{pmatrix} G_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)} + H_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)} \\ -i\left(G_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)} - H_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)}\right) \end{pmatrix} \end{pmatrix}
\end{aligned}
$$

c) Explain from the above expression why, once the three FFT's above have been computed, the rest can be computed with $N/2$ complex multiplications, and $2 \times N/4 + N = 3N/2$ complex additions. This is equivalent to $2N$ real multiplications and $N + 3N = 4N$ real additions.

Hint. It is important that $G_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oe)}$ and $H_{N/4}\mathrm{DFT}_{N/4}\boldsymbol{x}^{(oo)}$ are computed first, and the sum and difference of these two afterwards.

d) Due to what we just showed, our new algorithm leads to real multiplication and addition counts which satisfy

$$M_N = M_{N/2} + 2M_{N/4} + 2N \qquad A_N = A_{N/2} + 2A_{N/4} + 4N$$

Find the general solutions to these difference equations and conclude from these that $M_N = O\left(\frac{4}{3}N\log_2 N\right)$, and $A_N = O\left(\frac{8}{3}N\log_2 N\right)$. The operation count is thus $O\left(4N\log_2 N\right)$, which is a reduction of $N\log_2 N$ from the FFT algorithm.

e) Write an FFT kernel function `FFTKernelSplitradix` for the split-radix algorithm (again this should handle both the forward and reverse transforms). Are there more or less recursive function calls in this function than in the original FFT algorithm? Also compare the execution times with our original FFT algorithm, as we did in Exercise 2.23. Try to explain what you see in this comparison.

By carefully examining the algorithm we have developed, one can reduce the operation count to $4N\log_2 N - 6N + 8$. This does not reduce the order of the algorithm, but for small N (which often is the case in applications) this

reduces the number of operations considerably, since $6N$ is large compared to $4N \log_2 N$ for small N. In addition to having a lower number of operations than the FFT algorithm of Theorem 2.31, a bigger percentage of the operations are additions for our new algorithm: there are now twice as many additions than multiplications. Since multiplications may be more time-consuming than additions (depending on how the CPU computes floating-point arithmetic), this can be a big advantage.

Exercise 2.30: Bit-reversal

In this exercise we will make some considerations which will help us explain the code for bit-reversal. This is perhaps not a mathematically challenging exercise, but nevertheless a good exercise in how to think when developing an efficient algorithm. We will use the notation i for an index, and j for its bit-reverse. If we bit-reverse k bits, we will write $N = 2^k$ for the number of possible indices.

a) Consider the following code

```
j = 0;
for i = 0:(N-1)
    j
    m = N/2;
    while (m >= 1 && j >= m)
        j = j - m;
        m = m/2;
    end
    j = j + m;
end
```

Explain that the code prints all numbers in $[0, N-1]$ in bit-reversed order (i.e. j). Verify this by running the program, and writing down the bits for all numbers for, say $N = 16$. In particular explain the decrements and increments made to the variable j. The code above thus produces pairs of numbers (i, j), where j is the bit-reverse of i. As can be seen, `bitreverse` applies similar code, and then swaps the values x_i and x_j in $\boldsymbol{x}$, as it should.

Since bit-reverse is its own inverse (i.e. $P^2 = I$), it can be performed by swapping elements i and j. One way to secure that bit-reverse is done only once, is to perform it only when $j > i$. You see that `bitreverse` includes this check.

b) Explain that $N - j - 1$ is the bit-reverse of $N - i - 1$. Due to this, when $i, j < N/2$, we have that $N - i - 1, N - j - l \geq N/2$, and that `bitreversal` can swap them. Moreover, all swaps where $i, j \geq N/2$ can be performed immediately when pairs where $i, j < N/2$ are encountered. Explain also that $j < N/2$ if and only if i is even. In the code you can see that the swaps (i, j) and $(N - i - 1, N - j - 1)$ are performed together when i is even, due to this.

c) Assume that $i < N/2$ is odd. Explain that $j \geq N/2$, so that $j > i$. This says that when $i < N/2$ is odd, we can always swap i and j (this is the last swap performed in the code). All swaps where $0 \leq j < N/2$ and $N/2 \leq j < N$ can be performed in this way.

In `bitreversal`, you can see that the bit-reversal of $2r$ and $2r+1$ are handled together (i.e. i is increased with 2 in the `for`-loop). The effect of this is that the number of `if`-tests can be reduced, due to the observations from b) and c).

2.5 Summary

We defined digital sound, and demonstrated how we could perform simple operations on digital sound such as adding noise, playing at different rates e.t.c.. Digital sound could be obtained by sampling the sounds from the previous chapter. We considered the analog of Fourier series for digital sound, which is called the Discrete Fourier Transform, and looked at its properties and its relation to Fourier series. We also saw that the sampling theorem guaranteed that there is no loss in considering the samples of a function, as long as the sampling rate is high enough compared to the highest frequency in the sound.

We obtained an implementation of the DFT, called the FFT, which is more efficient in terms of the number of arithmetic operations than a direct implementation of the DFT. The FFT has been cited as one of the ten most important algorithms of the 20'th century [3]. The original paper [6] by Cooley and Tukey dates back to 1965, and handles the case when N is composite. In the literature, one has been interested in the FFT algorithms where the number of (real) additions and multiplications (combined) is as low as possible. This number is also called the *flop count* . The presentation in this book thus differs from the literature in that we mostly count only the number of multiplications. The split-radix algorithm [40, 10], which we reviewed in Exercise 2.4. 2.29, held the record for the lowest flop count until quite recently. In [18], Frigo and Johnson showed that the operation count can be reduced to $O(34N\log_2(N)/9)$, which clearly is less than the $O(4N\log_2 N)$ we obatined for the split-radix algorithm. It may seem strange that the total number of additions and multiplications are considered: Aren't multiplications more time-consuming than additions? When you consider how this is done mechanically, this is certainly the case: In fact, floating point multiplication can be considered as a combination of many floating point additions. Due to this, one can find many places in the literature where expressions are rewritten so that the multiplication count is reduced, at the cost of a higher addition count. Winograd's algorithm [39] is an example of this, where the number of additions is much higher than the number of multiplications. However, most modern CPU's have more complex hardware dedicated to computing multiplications, which can result in that one floating point multiplication can be performed in one cycle, just as one addition can. Another thing is that modern CPU's typically can perform many additions and multiplications in parallel, and the higher complexity in the multiplication hardware may result in that the CPU can run less multiplications in parallel, compared to additions. In other words, if we run test program on a computer, it may be difficult to detect any differences in performance between addition and multiplication, even though complex big-scale computing should in theory show some differences. There are also other important aspects of the FFT, besides

the flop count. Another is memory use. It is possible to implement the FFT so that the output is computed into the same memory as the input, so that the FFT algorithm does not require extra memory besides the input buffer. Clearly, one should bit-reverse the input buffer in order to achieve this.

We have now defined two types of transforms to the frequency domain: Fourier series for continuous, periodic functions, and the DFT, for periodic vectors. In the literature there are in two other transforms also: The Continuous time Fourier transform (CTFT) we have already mentioned at the end of Chapter 1. We also have the Discrete time Fourier transform (DTFT)) for vectors which are not periodic [28]. In this book we will deliberately avoid the DTFT as well, since it assumes that the signal to transform is of infinite duration, while we in practice analyze signals with a limited time scope.

The sampling theorem is also one of the most important results of the last century. It was discovered by Harry Nyquist and Claude Shannon [31], but also by others independently. One can show that the sampling theorem holds also for functions which are not periodic, as long as we have the same bound on the highest frequency. This is more common in the literature. In fact, the proof seen here where we restrict to periodic functions is not common. The advantage of the proof seen here is that we remain in a finite dimensional setting, and that we only need the DFT. More generally, proofs of the sampling theorem in the literature use the DTFT and the CTFT.